Saikat Chakraborty
https://saikatc.info Email: saikatc@cs.columbia.edu

Research Statement

Software developers around the world spend hundreds of hours writing and maintaining code. Unsurprisingly, in their work cycle, they do a lot of repetitive works. My research is motivated by building tools and techniques that reduce software engineering practitioners' burden and increase developer productivity. The "Naturalness" conjecture suggests source code artifacts (e.g., source code[Hindle'16], code changes[Nguyen'10, Nguyen'13, Ray'15], bugs[Ray'16]) are repetitive, i.e., follow similar patterns across different programmers, repositories, and organizations. My research area is *Programming Language Processing (PLP)* – a coalescence between Software Engineering (SE) and Machine Learning (ML) aimed at developing methods for analyzing such patterns. My research philosophy is to improve developer productivity by conducting multi-disciplinary collaborative research combining Software Engineering, Natural Language Processing, Program Analysis, Security, and HCI. I believe that my research background in Software Engineering and extensive collaboration with NLP and Security make me suitable for embracing such a research challenge.

I divide my research focus into two orthogonal perspectives - (i.) source code understanding and (ii.) source code generation. My research efforts focus on developing ML models specifically catered to programming languages. While Programming Languages (PL) exhibit strictly defined syntax and semantics, processing PLs (i.e., source code written by programmers) bring specific challenges. First, a slight change in a source code can drastically change the functionality of the source code. For instance, in a recent CVE[1] (CVE-2021-38094), using datatype "*int*" instead of "*float*" made the code susceptible to *integer overflow* vulnerability. In fact, a wide number of known security bugs can be fixed with very small changes in source code. Furthermore, source code can be arbitrarily large with virtually infinite possible identifier names. The dependencies between code snippets could be highly hierarchical through function, class, package, project, even third-party libraries. Thus, while developing automation tools to assist developers, we should consider these properties, and design tools accordingly.

In my Ph.D. career, I have been pursuing the development of AI-driven source code analysis tools. I have developed tools for Vulnerability Detection [TSE'21], Automated Code Change/Program Repair [TSE'20, ASE'21], Program Comprehension [ACL'20], Code Generation[ACL'18, NAACL'21, EMNLP'21], Code Translation across PLs [NAACL'21], Code Search [EMNLP'21, SCAM'19] etc. My research on learning general-purpose source code representation [NAACL'21] has been on the leaderboard of the Microsoft CodeXGLUE challenge on several SE automation tasks (i.e., Vulnerability detection, Code translation, Code refinement, etc.)[2]. My future research interest is to push the boundary of machine understanding of source code and building tools for improving the software development pipeline (i.e., building better developer assisting tools) with Machine Learning and Deep Learning.

Learning to Understand Source Code

Automated reasoning about source code requires a profound (machine)-understanding of source code. The success of automation in various software engineering tasks hinges on such an understanding of code. A machine that understands source code syntax and semantics can be deployed for different tasks to reduce developers' burden. For instance, such a machine can learn vulnerable code patterns from existing vulnerabilities - acting as an extra set of eyes for ensuring the safety and reliability of software. My research aims at developing robust models for source code understanding. In order for doing so, I focus on developing new ML models and techniques that account for PL properties. Here are some of my research illustrating the need to solve the problem of source code understanding.

*** Vulnerability detection in source code.** With the advent of machine learning in source code analysis, there has been a paradigm shift towards handing over the responsibility to identify repetitive bugs and vulnerabilities to data-driven ML systems. Several recent Deep Learning (DL) based studies have demonstrated promising results achieving an accuracy of up to 95% at detecting vulnerabilities. Such development begs an answer to the question, "*How well do the state-of-the-art DL-based techniques perform in a real-world vulnerability prediction scenario?*" To our surprise, we find that their performance drops by more than 50% when we applied to identify vulnerabilities in two large-scale software (i.e., Chromium and Debian). Further investigation revealed that if we present source code as a sequence of tokens to an ML model, there is no way for the model to reason about the syntax and semantics. In our TSE'21 paper (ReVeal) [TSE'21], we proposed a novel graph-based source code representation model equipped with the capacity to reason about PL properties. With in-depth investigation, we showed that

[1] http://cve.mitre.org/
[2] https://microsoft.github.io/CodeXGLUE/

understanding the core properties of source code is a *must* for building a usable ML technique for vulnerability detection. These findings in this paper can be extended to general-purpose bug detection.

In addition to ReVeal, our recent work in collaboration with the IBM research on a robust understanding of code aims at understanding the functional properties of source code. Executing every piece of code to understand its functionality is challenging. We solve this problem by drawing inspiration from mutation testing and delta debugging. We realize that a slight change in a code could make an immense difference in the source code. For instance, changing a conditional operator from ">" to "<" could drastically change the functionality of code; it could even trigger a vulnerable backdoor in the source code. On the other hand, completely different (both textually and structurally) different codes could have the same functionality. For example, the same code can be implemented both recursively and iteratively, resulting in different structures yet the same functionality. We proposed a robust technique to learn from such functional contrast. Such source code understanding model showed great promise in different code understanding tasks, e.g., vulnerability detection, clone detection, etc. The paper is currently under review, and the preprint is available in arxiv[3].

*** Code Summarization.** Writing (or updating) source code documentation is tedious work developers have to undergo to ensure the proper maintainability of source code. To ease the burden on the developers, we aimed at building an automated tool for code summarization. Our research collaboration with Dr. Kai-Wei Chang from UCLA revealed that developers' written summaries mainly depend on the identifiers and APIs used in the code. Developers write meaningful API/identifiers, often consisting of many meaningful subtokens. Our ACL'20 paper showed that deconstruction of identifiers into subtokens can significantly improve automated code summarization performance. In addition, we also showed that the interpretation of an identifier depends on the local context of the source code. Thus our model leans toward identifier representation emphasizing the location context.

Learning to Generate/Edit Source Code

Self-correcting software or self-modifying software is one of the key stepping stones for total automation in software. A software that can – a) identify and repair any potential flaw, or b) change part of its source code following the previous code-change pattern from the developers. Previous research efforts showed that changes made by real developers are repetitive. Such repetitive changes enable us to build an intelligent machine that can learn those code change patterns and apply them wherever necessary. The automatic Code Change task can be formulated as a translation task, where the intelligent machine will translate a piece of code to a changed code. Consequently, Machine Translator tools such as Neural Machine Translators (NMT) are a natural fit for this task. At the heart of such an NMT system, there is an encoder that analyzes the code before editing, and there is a Code Generator (decoder) that generates the edited code.

In my research, I offer a thorough investigation into the usage of NMT in the Automated Code Change task. Some key insights guide my studies and subsequent tool development efforts – 1. The tool must understand what the input code is. The encoder of the NMT *must not* solely rely on the code lexicons. It should be able to reason (albeit implicitly) about the input code's syntax and semantics. 2. The Code Generator *must* generate syntactically correct code. Any generated code that is not syntactically correct is entirely unusable from a fully automated tool's point of view. 3. The Code Generator *must* make sure that the generated code is contextually correct. The code must – a) use appropriate names for appropriate identifiers, b) maintain proper usage of APIs, and c) exhibit developers' and organizational coding practices and rules. Fulfilling this requirement will embellish an automated code editor in the actual development environment. In light of these insights, we first developed CODIT [TSE'20], collaborating with Dr. Miltos Allamanis from Microsoft Research. We investigated the syntactic correctness guarantee of generated code through Context-Free Grammar (CFG) of the programming language. Using a tree-based model, the decoder in CODIT samples from the CFG and generates the code's syntax tree. In doing so, it ensures the syntactic guarantee of the generated code. CODIT shows significant promise in automated code change and automated program repair.

While CODIT successfully generates syntactically correct code, it was not designed to provide any provision to learn about contextual correctness. Generating code that adheres to such patterns is essential for a fully-automated system. For instance, consider an *Exception Handling* code scenario - one context may demand handling the exception locally, others may pass it to the caller. While both the codes are syntactically correct from CODIT 's perspective, the ultimate correct code depends on the context. We cannot deterministically say which one CODIT will prefer to generate over the other. We solve the contextual correctness problem with critical insight into how any machine learning model works. Conceptually, in any ML model, there are two parts: understanding the input code's contextual semantics and reasoning about the task at hand. Similarly, in the case of NMT based code change, the

[3] https://arxiv.org/pdf/2110.03868.pdf

tool should learn – a) understand the code contextual semantics and generate syntactically and contextually correct code, and b) learn to apply the change patterns. Since the first task does not necessarily depend on the code changes, we can pre-train an NMT model with raw code samples. Such training aims to train the encoder to understand the code semantics and train the decoder to generate syntactically and contextually correct code. With all these insights, we developed PLBART [NAACL'21] in collaboration with Dr. Kai-Wei Chang from UCLA. We leveraged denoising auto-encoding to pre-train the encoder and decoder. PLBART showed great promise in a wide variety of downstream software engineering tasks, including Automated Code Change.

We learned a key lesson from the experiences with CODIT and PLBART. While high-level code change patterns are repetitive, individual code changes are usually much more diverse. Concrete changes that happen to a piece of code depend on the development circumstances. For a developer, such a piece of information about the circumstances can be bug/issue reports, test cases, code review comments, etc. We thus take another input (summary of the edit) to automate code editing. Consequently, we designed MODIT [ASE'21] to incorporate such an auxiliary source of information, where we showed for editing a piece of code (i) the context is vital to edit a code successfully – often the edited code collects existing components (e.g., variables, APIs) from the context, and (ii) a description summarizing the edit could significantly narrow down the search space for code edit generation. We are further enhancing MODIT by guiding it to generate syntactically and semantically correct code. Currently, I am developing a model with supervised training to generate syntactically and semantically correct code in collaboration with Dr. Premkumar Devanbu from UC Davis. In this work, we are proposing to reinforce such correctness while training. In particular, I propose rewarding the model when it generates syntactically and semantically correct code, penalizing otherwise. We train the model in such a way that maximizes such expected rewards and minimizes the expected penalty. Training the model in such a way will both (a) take advantage of pre-training from a large code corpus and (b) take advantage of auxiliary reward based on generated code quality.

Code Search Augmented Learning

An industry-scale case study [Sadowski et al. '15] showed that developers often use code-search for various purposes, including understanding the code behavior, solving problems, etc. I divide my research objective in code-search into two different agendas - (i) developing search techniques for source code and (ii) using search for solving different developer assistance tasks. Our research found that code search is different from searching a text; thus, off-the-shelf ranking methods used in traditional search result in sub-optimal solutions in code search. We presented the empirical study results in an ICSE'18 poster and SCAM'19 paper. We also designed a framework for automatically selecting the most favorable ranking metric for different software engineering tasks following the empirical findings [SCAM'19]. In a recent paper [REDCODER, EMNLP-findings'21], we proposed a more generic deep learning approach for code search given a text query. In particular, we proposed to learn the alignment between a vector representation of a query text and corresponding source code with supervised learning.

In addition to inventing effective ways to search source code, I also focus on using code search to increase developer productivity. For instance, when a developer searches for some code in StackOverflow, they often have to modify the search result to use in their particular need. As such, in our EMNLP paper, we proposed a search augmented code generation technique, REDCODER. Given a natural language query, REDCODER first searches for relevant code using the search tool engineered explicitly for code searching. We then use the search result in conjunction with the query to generate source code for the programmer. Our investigation showed that, for small functions and non-trivial code generations, REDCODER archives outstanding performance. We hypothesize that if we can hand over trivial programming-related tasks to the machine, developers could focus their primary attention on solving challenging problems, thus improving productivity. Currently, I am collaborating with RISElab from UC Berkeley to extend the code search to find semantic clones across different programming languages.

Future Plan and Long Term Goal

Program Generation/Synthesis. Automated program generation/synthesis can help developers to a great extent to increase productivity. The ever-changing landscape of APIs and libraries in modern API-driven software development demands a steep learning curve for programmers. To each such burden, an automated program synthesis/generation tool may help programmers. Such an automated tool can generate programs from a set of I/O examples, a description of the task, etc. The current SOTA for machine generation of programs is far from perfect. On the one hand, there is *precise program synthesis* with provable correctness. However, these programs are often tailored towards the problem domain and do not scale well for general-purpose programs. On the other hand, *general-purpose source code generation* often generates syntactically and semantically incorrect codes, hindering their integration in automated programming. My future research goal is to bridge the gap between these two ends of

the spectrum. In particular, I aim at building tools that can benefit from both ends of the spectrum. I aim at collaborating with experts from the PL community to integrate provable syntactic and semantic correctness into model-based general-purpose program generation. At the same time, I also aim at applying my experience and expertise in modeling source code to build a holistic solution for general-purpose source code generation with a provable correctness guarantee. In the longer term, I envision building IDE support with which developers, especially novice developers, can take great advantage of auto-generated code.

Refining Cross-Lingual search with semantic matching. Similar to the evolving nature of libraries, programming languages are also evolving rapidly. With the advent of newer and richer programming languages, cross-lingual code search is becoming a significant challenge to solve [Matthews et al. '21]. In the presence of drastic differences between syntax and semantics of different PLs, a very straightforward way to check cross-lingual similarity is to compare them against a set of I/O examples. However, such an approach comes with several significant hurdles. First, not all the pieces of code will be executable. Thus we cannot compare it for I/O behavior. Second, running a set of I/O for every search could be very expensive. I plan on working towards learning the alignment between code syntax and semantics across different programming languages to solve these problems. I aim at developing models where such alignment learning can be reinforced by the I/O behavior of part of training data. In particular, I aim to learn syntactic and semantic alignment to maximize the I/O similarity. When fully trained, knowledge about I/O behavior will be embedded in the model and thus will not require executing code in runtime. I aim to collaborate with experts from Machine Learning, Information Retrieval, and Programming Languages to design such a system.

Ensuring security and trustworthiness of software systems. Digitization and dependence on automation make the human race susceptible to security/privacy and trust violation. The distributed, independent, and layered architecture of modern-day systems may raise vulnerable emergent behavior across the system as a whole, even when each component in the system is independently tested for their respective service level agreements(SLA). For instance, consider an IoT system automated home; the sensors, actuators, routers, servers are all independent layers of a system. However, when such systems use shared resources (e.g., memory, network access), they may create security/privacy attack backdoors due to the inconsistencies between layers. The multi-vendor nature of different layers leaves room for confusion and makes it difficult to reach a consensus on the SLA. Such vulnerable emergent behaviors are often not anticipated layers assembler and often identified in post-deployment. I aim at developing ML-based systems to learn to anticipate such cross-layer vulnerabilities based on previously known vulnerabilities. I aim at collaborating with experts from systems, security, privacy to solve such problems. In addition to cross-layer vulnerability detection, I also aim at researching intra-layer vulnerability localization, drawing inspiration from debugging. A central direction of my future research agenda is to precisely localize the bug reasoning about the execution trace and program flow graphs. In particular, my goal is to draw inspiration from developers' way of reasoning about "*states*" in an execution trace. I aim at collaborating with experts from testing and debugging to attain such goals.

Design and development of IDE tools. My first-hand experience working/collaborating in the industry (Google, Facebook, Fujitsu, IBM research) gave me a unique opportunity to experience the developers' needs. More precisely, when used in an actual development environment, there are many things to consider. We first have to consider the scalability, latency, overhead of a technique. For instance, if a code recommender (perhaps with excellent accuracy) takes more time to predict an identifier than the developer would need, the developer might even turn off the feature. Thus, the human component/user experience is a significant consideration while building such tools. In my future research, I aim at researching to develop IDE tools for developers in close collaboration with the development team. I aim to survey developers' needs, how they behave while using specific features, and which part of the code/tool they pay attention to. I also aim to learn their experience using a tool, what makes them use it, what demotivates them, etc. With these motivations and lessons, I aim at building tools that developers could use in their day-to-day programming activities. I aim at collaborating with experts from HCI and UX to improve the programmers' experience using the ML-based systems I aim to build.

References

My papers

- **[TSE-'20]** CODIT: Code Edits with Tree-Based Machine Translation, S. Chakraborty, Y. Ding, M. Allamanis, B. Ray, in IEEE Transactions on Software Engineering, 2020.
- **[TSE '21]** Deep Learning-based Vulnerability Detection: Are We There Yet? S. Chakraborty, R. Krishna, Y. Ding, B. Ray, IEEE Transaction of Software Engineering, 2021.

- **[ASE '21]** On Multi-Modal Learning of Editing Source Code, S. Chakraborty, B. Ray, Accepted to be published in The 36th IEEE/ACM International Conference on Automated Software Engineering.
- **[NAACL '21]** A Unified Pre-training for Program Understanding and Generation, WU. Ahmad, S. Chakraborty, B. Ray, K. Chang, Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), 2021.
- **[EMNLP '21]** Retrieval Augmented Code Generation and Summarization, MDR. Parvez, WU. Ahmad, S. Chakraborty, B. Ray, K. Chang, Findings of The 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP-findings), 2021.
- **[ACL '20]** A transformer-based Approach for Source Code Summarization (short paper), WU. Ahmad, S. Chakraborty, B. Ray, K. Chang, 58th Annual Meeting of the Association for Computational Linguistics (ACL) 2020.
- **[ACL '18]** Building Language Models for Text with Named Entities, R. Parvez, S. Chakraborty, B. Ray, K. Chang, 56th Annual Meeting of the Association for Computational Linguistics (ACL) 2018.
- **[SCAM '19]** Toward Optimal Selection of Information Retrieval Models for Software Engineering Tasks, MM. Rahman, S Chakraborty, G. Kaiser, B. Ray, 19th International Working Conference on Source Code Analysis and Manipulation (SCAM) 2019.
- **[ICSE-Poster'18]** Which similarity metric to use for software documents?: a study on information retrieval based software engineering tasks. Md. Rahman, S. Chakraborty, and B. Ray, Poster at Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. ACM, 2018.
- **[Under Review]** Contrastive Learning for Source Code with Structural and Functional Properties, Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, S. Chakraborty, under review (arXiv preprint arXiv:2110.03868)

Related references

- [Hindle '16] Hindle, A., Barr, E. T., Gabel, M., Su, Z., & Devanbu, P. (2016). On the naturalness of software. Communications of the ACM, 59(5), 122-131.
- [Ray '16] Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., & Devanbu, P. (2016, May). On the "naturalness" of buggy code. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) (pp. 428-439). IEEE.
- [Nguyen '10] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, 2010, pp. 315–324.
- [Nguyen '13] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 2013, pp. 180–190.
- [Ray '15] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, "The uniqueness of changes: Characteristics and applications," ser. MSR '15. ACM, 2015.
- [Chen '19] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and MartinMonperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair.IEEE Transactions onSoftware Engineering(TSE), 2019.