Baishakhi Ray
http://rayb.info Email: rayb@virginia.edu

Research Statement

In today's world, almost every aspect of our lives is controlled by software. Unfortunately, most software tends to be buggy, even the most safety- and security-critical software is not free from bugs. According to a recent report[1], software bugs cost the worldwide economy around USD$1.1 trillion in 2016 alone. My research addresses this problem---I devise automated techniques to find and fix bugs in large-scale real-world software and thus improve software reliability and security. In particular, I develop novel program analysis techniques to extract different abstract properties of code and apply advanced machine learning models to learn from those properties. Such models help to identify and fix anomalous properties that may lead to bugs.

A key insight behind my research is that the way regular developers write code, including the bugs they inadvertently introduce, are repetitive, predictable [Gabel'10, Hindle'12], and thus, amenable to automation. Developers not only introduce structurally similar code elements within and across software [MSR'17], but they also make similar mistakes [MSR'15]. A significant portion of the time and effort in writing code involves performing such repetitive task [FSE'12, PhDThesis], which is often tedious and error-prone [ASE'13]. I argue that a large amount of these repetitive coding jobs can be automated---we can autogenerate boilerplate code templates [MSR'15], detect and fix both pre-release and in production bugs without any manual effort [FSE'17, ASE'16, UsenixSec'16, ICSE'16, ASE'13], and test the security properties of existing code automatically [S&P'14]. To date, my research has been able to detect and fix more than a hundred bugs and security issues in large-scale software including 5 CVEs (Common Vulnerabilities and Exposures)[2].

My work has won many best paper awards: my paper on automatically detecting error handling bugs and fixes was awarded the best paper award in FSE'17, and my paper on analyzing cross-project clones across GitHub won the best paper award in MSR'17. My original FSE'14 paper on analyzing effects of programming language on code quality was invited for CACM journal. This work was also covered by media including SlashDot, The Register, Reddit, InfoWorld, Hacker News, etc. Previously, my PhD work on copy-paste errors was nominated for a distinguished paper award in ASE'13. My work on differential testing of certificate validation in SSL/TLS implementations won the best practical paper award in IEEE Security and Privacy Symposium (Oakland), 2014 [S&P'14], and the testing framework was later adopted by Mozilla.

Detecting & Fixing Bugs Using Code Similarity

It has been shown that developers write not only similar code but also make similar mistakes and fix them similarly. Thus, bugs and the corresponding corrected code often show repetitive patterns. I design algorithms and build tools that automatically detect and repair code using such repetitive patterns. In particular, I studied the following bug categories:

Error Handling Bugs in C. Low-level languages like C often do not support any error handling primitives and leave it up to developers to create their own mechanisms for error propagation and handling. However, developers often make mistakes while writing the repetitive and tedious error handling code and inadvertently introduce bugs. For example, a large number of security flaws in SSL/TLS implementations result from incorrect error handling (e.g., CVE-2014-0092, CVE-2015-0208, CVE-2015-0288, CVE-2015-0285, CVE-2015-0292, etc.). These bugs are often hard to detect and localize using conventional automated bug-finding techniques because they do not display any obviously erroneous behaviors (e.g., crashes or assertion failures) but instead cause subtle inaccuracies that violate the intended security and privacy guarantees of SSL/TLS. Therefore, it is extremely important to detect and fix these bugs at an earlier stage of the development cycle, to minimize their disastrous side effects when unknowingly released to production. To understand the nature of error handling bugs that occur in widely used C programs, I conducted a comprehensive study of real-world post-production error handling bugs and their eventual fixes. Leveraging this knowledge, I then designed, implemented, and evaluated a toolchain [Usenix'16, FSE'17, ASE'16] based on under-constrained symbolic execution and AST analysis, that not only detects and characterizes different types of error handling bugs but also automatically fixes them. Several of these bug fixes were adopted by OpenSSL developers and resulted in one CVE. This work received the best paper award in FSE'17.

[1]https://raygun.com/blog/cost-of-software-errors/

[2]https://www.cvedetails.com/cve-help.php (CVE stands for Common Vulnerability and Exposure, which is a list of critical security vulnerabilities maintained by Department of Homeland Security)

Copy-paste Bugs. While implementing new functionality similar to existing functionality, developers often copy and paste code from a source to a target location. While copy-pasting, they need to adapt the pasted code to the target context; a failure to properly adapt the code is known as copy-paste errors and can have disastrous consequences. In order to automatically detect copy-paste errors, I investigated: (1) What are the common types of copy-paste errors? (2) How can they be automatically detected [ASE'13]? By analyzing the version histories of FreeBSD and Linux, I found five common types of copy-paste errors and then, leveraging this categorization, I designed a static analysis technique to detect and characterize copy-paste errors. My key insight was that all copied instances of a piece of code should behave similarly except for the differences caused by their contexts; the behavioral similarity was captured by program dependence graphs. Incorporating such code semantics in the analysis technique helped me to detect real copy-paste bugs in widely popular software like FreeBSD, Eclipse, Mozilla, etc. I collaborated with researchers from NASA for this work, which was nominated for a distinguished paper award in ASE'13.

In the above techniques, I focused on some particular types of bugs. However, a significant number of bugs may not fit any of the well-defined bug categories or violate specific well-known rules. Yet they may result in errors by violating the implicit rules of common coding practices [Kremenek'06]. I aim to develop novel static and dynamic program analysis techniques that can infer such implicit program rules so code elements that violate these rules can be automatically marked as potential bugs.

Generic Bugs (using NLP models). Researchers have demonstrated that the predictability of source code written by ordinary developers to solve real-world problems is similar to that of natural language [Hindle'12]. Such naturalness/predictability have been successfully captured by statistical language models and used to build suggestion engines, porting tools, coding standards checkers, and idiom miners. This suggests that code that appears improbable, or surprising, to a good statistical language model is "unnatural" in some sense, and thus possibly suspicious, i.e., buggy. I investigated this hypothesis in work published at ICSE'16: I considered a large corpus of bug fix commits, from 10 popular widely-used Java projects, and built a language model of the source code and evaluated the naturalness (i.e., entropy) of buggy code and the corresponding fixes as computed by the model. I found that code with bugs tends to be more entropic (i.e., unnatural), becoming less so as bugs are fixed. This approach can be used as an effective code inspection tool by focusing on highly entropic lines. I also showed that such entropy-based techniques could complement generic static bug finders (PMD, FindBugs).

Differential Testing. Nowadays in the open software market, multiple pieces of software are available to users that provide similar functionality. For example, there is a pool of popular SSL/TLS libraries (e.g., OpenSSL, GnuTLS, NSS, CyaSSL, GnuTLS, PolarSSL, MatrixSSL, etc.) for securing network connections from man-in-the-middle attacks. Such code may not be structurally similar but implements similar functionalities. For example, certificate validation is a crucial part of SSL/TLS connection setup. Though implemented differently, the certificate validation logic of these different libraries should serve the same purpose, following the SSL/TLS protocol: for a given certificate, all of the libraries should either accept it as valid or reject it as invalid. In collaboration with security researchers at the University of Texas at Austin, we designed the first large-scale framework for testing certificate validation logic in SSL/TLS implementations. First, we generated millions of synthetic certificates by randomly mutating parts of real certificates and thus induced unusual combinations of extensions and constraints. A valid SSL implementation should be able to detect and reject the unusual mutants. Next, using a differential testing framework, we checked whether one SSL/TLS implementation accepts a certificate while another rejects the same certificate. We used such discrepancies as an oracle for finding flaws in individual implementations. We uncovered 208 discrepancies between popular SSL/TLS implementations, many of them are caused by serious security vulnerabilities. This paper received the best student paper award at Oakland [S&P'14].

Analytical support for improving software reliability

Despite 50+ years effort from the Software Engineering community to improve software reliability, we are still seeing a rapid increase in the number of bugs---in 2016 alone software bugs at 363 companies affected 4.4 billion customers and caused more than 315 years of lost time. Thus, it is important to investigate why such bugs are still occurring, and how the existing methodologies intended to prevent the bugs are used in practice. Thanks to the large number of diverse open-source projects available in software forges like GitHub, it has become possible to perform such software forensics. Each of these project repositories hosts source code along with entire evolution history, descriptions, mailing lists, bug databases, etc. I implemented a number of code analysis and text analysis tools to gather different metrics from GitHub project repositories. Then applying a series of advanced data analysis methods from machine learning, visualization, and regression analysis techniques, I investigated what is going wrong and how to correct it.

Effect of programming languages on software quality. To investigate whether the choice of programming language has any effect on program quality (bug proneness in particular), I gathered a very large data set from GitHub (728 projects, 63M lines of code, 29K authors, 1.5M commits, in 17 most popular languages) [CACM'17, FSE'14]. Using a mixed-methods approach, combining multiple regression modeling with visualization and text analytics, I studied the effect of modern programming language features such as static v.s. dynamic typing and strong v.s. weak typing on software quality. By triangulating the findings from different methods, and controlling for confounding effects such as code size, project age, and contributors, I observed that most bugs stem from logic flaws orthogonal to the choice of programming language. I argue that developers need better tool support for preventing and detecting logic bugs, which history has shown will not come from improving programming languages.

API stability and adoption in the Android Ecosystem. In today's software ecosystems, which are primarily governed by Web, cloud, and mobile technologies, APIs (Application Programming Interfaces) perform a key role in connecting disparate software and sharing common software infrastructure. Big players like Google, Facebook, and Microsoft aggressively publish new APIs to accommodate new feature requests, bugs fixes, and performance improvements while expecting developers will utilize these APIs in their own application programs to update that software immediately. I investigated how such fast-paced API evolution affects the overall software ecosystem. My study on Android API evolution showed that application developers are hesitant to adopt fast evolving, unstable APIs. For instance, while there are on average 115 Android API updates per month, clients adopt the new APIs rather slowly, with a median lag period of 16 months [ICSM'13]. Furthermore, client code using new APIs is typically more defect prone than code without recent API adaptations. These results show that aggressive API release cycles are inadvertently hurting the quality of the Android ecosystem rather than improving it. To the best of my knowledge, this is the first work studying API adoption in a large software ecosystem, and the study suggests how to better promote API adoption and how to facilitate the growth of an overall ecosystem.

Ongoing & Future Work

With the continuing growth of the software industry with many different kinds of new applications, programming paradigms, and platforms are seemingly emerging every day, numerous interesting research questions on improving software reliability and developer productivity remain open. Due to fundamentally different programming paradigms like MapReduce, Deep Neural Network (DNN), Android and the Internet of Things (IoT), the traditional Software Engineering problems including testing/debugging/patching need to be revisited. One of my main ongoing and future research agendas is to develop domain-specific techniques for improving reliability, security, and performance of such emerging software systems, as summarized below:

Testing Self-Driving Cars. DNN-based autonomous cars, using sensors like camera, LiDAR, etc., can drive without human intervention. Many major manufacturers including Tesla, GM, Ford, BMW, and Waymo/Google are working on building and testing different types of autonomous vehicles. However, despite their spectacular progress, DNNs, just like traditional software, often demonstrate incorrect or unexpected corner-case behaviors -- bugs -- that can lead to potentially fatal collisions. Several real-world accidents involving autonomous cars have already happened including one that resulted in a fatality. Most existing testing techniques for DNN-driven vehicles are heavily dependent on the manual collection of test data under different driving conditions, which becomes prohibitively expensive as the number of test conditions increases. In collaboration with Suman Jana and Junfeng Yang from Columbia University and Yinzhi Cao from Lehigh University, we are designing, implementing, and evaluating a systematic testing tool for automatically detecting erroneous behaviors of DNN-driven vehicles. Our tool is designed to automatically generate test cases leveraging real-world changes in driving conditions like rain, fog, lighting conditions, etc. We systematically explore different parts of the DNN logic by generating test inputs that maximize the numbers of activated neurons. So far we have found thousands of erroneous behaviors under different realistic driving conditions, many of which could lead to potentially fatal crashes, in the three top performing DNNs of the Udacity self-driving car challenge [Tian'17].

Natural Language Models for Android Applications. Source code often follows code templates, i.e., frequently repeating code structures where the only differences are in parameters and variable names. This observation is, in particular, true for App development platforms like Android, where the apps are heavily dependent on the platform APIs. There are only a few valid ways of using these APIs, which result in repetitive code templates [Raychev'14]. I am investigating a novel language model for such API driven programming in collaboration with NLP expert Prof. Kai-Wei Chang of UCLA. We are in the process of designing a template-based neural language model that predicts the structural properties of the code by learning from the templates, while predicting local code properties by learning from sequential code structure. Android applications often undergo many code transformations, e.g., API update, obfuscation, etc. Such transformations are also repetitive and show similar patterns due to heavy API usage. For example, when an API is updated all the applications using the API eventually need to update their code accordingly. I am also leveraging machine translation models to learn from previous transformations. In this case, the models are trained with the parallel corpora of original and transformed code versions. I believe that such models have significant potential for a plethora of Software Engineering applications including automatic code generation, bug fix, feature update, deobfuscation, malware detection, etc. For instance, I am collaborating with Miltos Allamanis of Microsoft Research (an NLP expert) to develop NLP models that can automatically repair buggy Android applications. Further, I am working with Gail Kaiser and her students at Columbia University, and Jonathan Bell from George Mason to automatically deobfuscate obfuscated APIs.

Analyzing Big-Data Frameworks. Big data frameworks, e.g., MapReduce, Spark, etc. are often so complex that users don't know how to configure them to achieve their desired properties (safety, security, runtime performance, etc.). We are in the process of developing a meta-heuristic search technique for searching configuration space for combinatorial optimization of such qualities based on three main hypotheses: (1) scale-up: small big-data jobs can serve as low-cost proxies for evaluating objective functions with significant performance benefits for much larger jobs; (2) scale-out: high-performing configurations found for one class of jobs provide significant benefits for related classes; and (3) centaur effectiveness: humans can profitably guide algorithmic searches for high-performing configurations. Initials result for the Hadoop platform shows up to 70% performance improvement. We further plan to develop a domain-specific language and static analysis framework to express the constraints of configuration spaces. We believe these additional techniques will help to find bugs in a configuration space, and will further reduce the search space of the meta-heuristic technique to a great extent. I am collaborating with Prof. Kevin Sullivan of UVa in this project.

Longer Term Goals. My interests lie primarily in the downstream phases of the software lifecycle, specifically testing, debugging, and patching. I believe, one of the main goals of Software Engineering research is to improve quality, reliability, and security throughout the software lifecycle. But we are now seeing a paradigm shift in the software development lifecycle, where decision making for high-dimensional real-world input is increasingly shifted from hand-coded program logic to machine learning. Machine learning-based applications from companies like Google [Sculley'14], Tesla, etc. are increasingly facing all the traditional software engineering challenges. However, machine learning (ML), e.g., deep neural network models (DNN), are fundamentally different from traditional software and existing software testing/debugging techniques do not apply in any obvious way to ML models. For example, unlike traditional software where the program logic is manually written by human developers, ML/DNN-based software automatically learns its logic from a large amount of data with minimal human guidance. Moreover, the logic of a traditional program is expressed in terms of control flow statements while DNNs use weights for edges between different neurons and nonlinear activation functions for similar purposes. These differences make automated testing of ML/DNN-based software challenging and present several interesting and novel research problems. I believe building new testing/debugging/patching frameworks for ML/DNN-based software is an emerging and important Software Engineering problem, especially given their increasing deployment in safety-critical systems like self-driving cars, diagnosis from medical imaging, etc. Thus, my long-term goal is to investigate novel testing, debugging and patching techniques for machine learning software.

References

My papers:

[FSE'17]	Tian, Y. and Ray, B., 2017, August. Automatically diagnosing and repairing error handling bugs in C. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 752-762). ACM.
[MSR'17]	Gharehyazie, M., Ray, B. and Filkov, V., 2017, May. Some from here, some from there: cross-project code reuse in GitHub. In Proceedings of the 14th International Conference on Mining Software Repositories (pp. 291-301). IEEE Press.
[ASE'16]	Kang, Y., Ray, B. and Jana, S., 2016, August. APEx: Automated inference of error specifications for C APIs. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (pp. 472-482). ACM.
[UsenixSec'16]	Jana, S., Kang, Y.J., Roth, S. and Ray, B., 2016, August. Automatically Detecting Error Handling Bugs Using Error Specifications. In USENIX Security Symposium (pp. 345-362).
[ICSE'16]	Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A. and Devanbu, P., 2016, May. On the naturalness of buggy code. In Proceedings of the 38th International Conference on Software Engineering (pp. 428-439). ACM.
[MSR'15]	Ray, B., Nagappan, M., Bird, C., Nagappan, N. and Zimmermann, T., 2015, May. The uniqueness of changes: Characteristics and applications. In Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on (pp. 34-44). IEEE.

[FSE’14]	B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu. “A large scale study of programming languages and code quality in github”.In: Proceedings of the ACM SIGSOFT 22nd International Symposium on the Foundations of Software Engineering. FSE 2014, pp. 155–165.
[S&P’14]	C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. “Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations”. In: IEEE Symposium on Security and Privacy 2014. S&P 2014, pp. 114–129.
[ASE’13]	B. Ray, M. Kim, S. Person, and N. Rungta. “Detecting and characterizing semantic inconsistencies in ported code”. In: Automated Software Engineering, 2013 IEEE/ACM 28th International Conference on. ASE 2013, pp. 367–377.
[ICSM’13]	T. McDonnell, B. Ray, and M. Kim. “An empirical study of API stability and adoption in the Android ecosystem”. In: Software Maintenance, 2013 29th IEEE International Conference on. ICSM 2013, pp. 70–79.
[FSE’12]	B. Ray and M. Kim. “A Case Study of Cross-system Porting in Forked Projects”. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE 2012, pp. 53.1–53.11.
[MSR’12]	J. Park, M. Kim, B. Ray, and D. H. Bae. “An empirical study of supplementary bug fixes”. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE. 2012, pp. 40–49.
[Tian’17]	Tian, Y., Pei, K., Jana, S. and Ray, B., 2017. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. *arXiv preprint arXiv:1708.08559*.
[PhDThesis]	B. Ray. “Analysis of cross-system porting and porting errors in software projects”. PhD thesis. The University of Texas at Austin, 2013.

Related papers:

[Gabel’10]	Gabel, M. and Su, Z., 2010, November. A study of the uniqueness of source code. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (pp. 147-156). ACM.
[Hindle’12]	Hindle, A., Barr, E.T., Su, Z., Gabel, M. and Devanbu, P., 2012, June. On the naturalness of software. In Software Engineering (ICSE), 2012 34th International Conference on(pp. 837-847). IEEE.
[Kremenek’06]	Kremenek, T., Twohey, P., Back, G., Ng, A. and Engler, D., 2006, November. From uncertainty to belief: Inferring the specification within. In Proceedings of the 7th symposium on Operating systems design and implementation (pp. 161-176). USENIX Association.
[Sculley’14]	Sculley, D., Phillips, T., Ebner, D., Chaudhary, V. and Young, M., 2014. Machine learning: The high-interest credit card of technical debt.
[Raychev’14]	Raychev, V., Vechev, M. and Yahav, E., 2014, June. Code completion with statistical language models. In ACM SIGPLAN Notices (Vol. 49, No. 6, pp. 419-428). ACM.