# RESEARCH STATEMENT

Baishakhi Ray

Over the last few decades, software industry has grown enormously. According to a recent survey, worldwide software industry has grown to $407.3 Billion in 2013, 4.8% more than the previous year. For maintaining such growth, we must ensure the quality of the software as well as the efficiency of the software development process. My research focuses on achieving these goals. In particular, I work on **Empirical Software Engineering** in order to understand the current software engineering process by statistically analyzing large-scale software projects. Then, leveraging this data-driven knowledge, I develop novel **program analysis** algorithms and **testing** techniques to improve software quality, reliability, as well as developer productivity.

A key insight behind my research is that similarities within and across software systems can be leveraged to improve both software quality and developer productivity. In my doctoral thesis, I empirically demonstrated that developers spend a significant amount of time and effort in introducing similar features and bug-fixes in and across different projects [FSE'12]. This involves a lot of repetitive work, which is often tedious and error-prone. To facilitate developers in this process, I developed a change recommendation system [MSRTR'14], a bug detection tool, and an automatic testing framework [ASE'13]. I also observed that multiple applications often implement the same features in different manner. Based on this fact, I developed a differential testing framework to test the SSL certificate validation routines across different SSL implementations [S&P'14]. To date, my research has been able to *detect more than a hundred bugs and security issues* in large scale software. In addition, I have extensively collaborated with systems researchers to apply existing software engineering techniques for building mobile and heterogeneous computing framework [SOSP'11; IEEE'08].

So far, my research has been well accepted by the community. My work on the quality of repetitive edits [ASE'13] was *nominated for distinguished paper award* in the International Conference on Automated Software Engineering conference, and was invited for a special issue in the journal of Automated Software Engineering. The work on differential testing of certificate validation in SSL/TLS implementations [S&P'14] won *best paper award* in IEEE Security and Privacy Symposium (Oakland), 2014, and the testing framework was later adopted by Mozilla. Our empirical study of supplementary bug fixes was invited to the Special Issue of Journal of Empirical Software Engineering (EMSE) [MSR'12]. My study on the impact of programming languages on software quality [FSE'14] was *covered by the media* including SlashDot, Reddit, Twitter, Register, Infoworld, *etc*.

The following is a summary of my research contributions that address some of the fundamental problems of software engineering.

## Repetitive changes in software

Changes in software development come in many forms. Some changes are frequent, idiomatic, or repetitive (*e.g.,* adding null checks or logging important values), while others are unique. Though existing literature has shown repetitiveness is pervasive in source code, the actual nature and cost required to introduce such repetitive changes are yet unknown. As part of my PhD thesis, I investigated different aspects (*e.g.,* extent, cost, quality, *etc.*) of repetitive changes across different open source and proprietary software including the BSDs, Linux, and multiple Microsoft projects.

***Estimating overhead of repeated work.*** In software engineering literature, a developer's work is often measured by the lines of code changed by the developer. Thus, to estimate the amount of repeated work, I computed the amount of repetitive changes during a project's evolution. I designed *Repertoire*, an automatic change analysis tool, that compares the edit contents and the corresponding operations of program patches to identify similar changes, with 94% precision and 84% recall [FSE-Demo'12]. Using *Repertoire*, I found that developers introduce a significant amount of repeated changes both within and across projects. Most notably, repetitive changes among forked projects (different variants of an existing project, *e.g.,* FreeBSD, NetBSD and OpenBSD) incur significant duplicate work. In each BSD release, on average, more than twelve thousand lines are ported from peer projects, and more than 25% of active developers participate in cross-system porting in each release [FSE'12]. Similar trends were also observed for intra-system repetitive changes in Linux and the Microsoft projects. I also noticed a non-trivial amount of repetitive changes in supplementary bug-fixing patches [MSR'12]. To implement repetitive changes, developers usually copy code from an existing implementation and then paste and adapt it to fit the new context. An incorrect adaptation often leads to an *copy-paste error*. Such errors are quite common in practice–in last 3 years 182 copy-paste errors were reported in Linux. These results confirmed that the overhead of repetitive changes is non-trivial and suggests that new, automated tools would be helpful to facilitate the process.

***Recommending changes.*** As a first step towards automating the repetitive changes, I built a change recommendation system. Learning from previous code changes, the recommendation system suggests relevant changes in two phases: (1) *change suggestion:* when developers select a code fragment to modify, it recommends possible changes that have previously been applied to similar code. (2) *change completion:* when developers introduce a repetitive change, this step recommends other patches that co-occurred with the change in the past. On average, the recommendation system suggests repetitive changes with 55% precision (20 percentage point better than its predecessor) and recommends change completion with 40% precision [MSRTR'14].

***Detecting copy-paste errors.*** 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 two-stage analysis technique to detect and characterize copy-paste errors. The first stage of the analysis, *SPA*, detects and categorizes inconsistencies in repetitive changes based on a static control and data dependence analysis. *SPA* successfully identifies copy-paste errors with 65% to 73% precision, an improvement by 14 to 17 percentage points *w.r.t.* previous tools. The second stage of the analysis, *SPA++*, uses the inconsistencies computed by *SPA* to direct symbolic execution in order to generate program behaviors that are impacted by the inconsistencies. *SPA++* further compares these program behaviors in the reference and target program versions and generates test inputs that exercise program paths containing the reported inconsistencies. A case study shows how *SPA++* can refine the results reported by *SPA*, and how the results of *SPA++* can help developers analyze inconsistencies in the repetitive changes.

## Analytical support for improving quality & productivity

Thanks to the large number of diverse open source projects available in software forges such as GitHub, it becomes possible to evaluate some long-standing questions about software engineering practices. Each of these project repositories hosts source code along with entire evolution history, description, mailing lists, bug database, *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, random networks, visualization, and regression analysis techniques, I shed some empirical light on how to improve software quality and developers' productivity.

- **Effect of programming languages on software quality.** This question has been a topic of much debate for a very long time. To answer, I gathered a very large data set from GitHub (728 projects, 63 Million SLOC, 29,000 authors, 1.5 million commits, in 17 languages) [FSE'14]. Using a mixed-methods approach, combining multiple regression modeling with visualization and text analytics, I studied the effect of language features such as static *v.s.* dynamic typing, strong *v.s.* weak typing on software quality. By triangulating findings from different methods, and controlling for confounding effects such as code size, project age, and contributors, I observed that a language design choice does have a significant, but modest effect on software quality. This work received significant media coverage including Slashdot, Infoworld, Register, Reddit, Twitter *etc*.

- **API Stability and Adoption in the Android Ecosystem.** In today's software ecosystem, which is primarily governed by web, cloud, and mobile technologies, APIs perform a key role to connect disparate software. Big players like Google, FaceBook, Microsoft aggressively publish new APIs to accommodate new feature requests, bugs fixes, and performance improvement. However, application programs adopt the new APIs rather slowly. Our recent study on Android API evolution shows that Android is evolving fast at a rate of 115 API updates per month on average [ICSM'13]. However, client adoption of the new APIs is slow, with a median lagging period of 16 months. Furthermore, client code with new APIs is typically more defect prone than the one without API adaptation. This study indicates that application software developers are hesitant to adopt fast evolving, unstable APIs. To the best of my knowledge, this is the first work studying API adoption, and suggests how to promote API adoption and how to facilitate growth of the overall ecosystems.

- **Other GitHub Analyses.** Assertions in a program are believed to improve software quality. I conducted a large scale study on how developers typically use assertions in C and C++ code and showed that they play positive role in improving code quality. I further characterized assertion usage along different process and product metrics. Such detailed characterization of assertions will help to predict relevant locations of useful assertions and eventually will improve code quality [UCTRa'14]. In another study, I and my colleagues studied gender and tenure diversity in online programming teams and found that both gender and tenure diversity are positive and significant predictors of productivity. These results can inform decision-making on all levels, leading to better outcomes in recruiting and performance [UCTRb'14].

## Automated adversarial testing

Nowadays in open software market, multiple software are available to users that provide similar functionality. For example, there exist 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. Certificate validation is a crucial part of SSL/TLS connection set up. Though implemented differently, the certificate validation logic of these different libraries should serve the same purpose, following the SSL/TLS protocol, *i.e.* for a given certificate, all of the libraries should either accept or reject it. Based on this philosophy, I and my colleagues designed, implemented, and applied the first large-scale framework for testing certificate validation logic in SSL/TLS implementations. First, we generate a millions of synthetic certificates by randomly mutating parts of real certificates and thus induce 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 check whether one SSL/TLS implementation accepts a certificate while another rejects the same certificate. We use 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 work won the best practical paper award in IEEE Symposium on Security and Privacy 2014 [S&P'14].

### Building secure and scalable software systems

I started my research career in developing software systems that are robust, reliable, and performance sensitive. I also have spent five years working as a professional software engineer, implementing various embedded systems and network protocol stacks. In fact, my systems building expertise from those early years motivated me to develop techniques and tools that would help building better software. Following is a brief summary of some research prototype systems that I designed and implemented.

- **Operating System Abstractions To Manage GPUs as Compute Devices.** Currently, GPUs are typically used for high-performance rendering or batch-oriented computations, but not as general purpose compute-intensive task, such as gesture recognition, brain-computer interfaces, or file system encryption. This is because current OS does not treat GPUs as a shared computational resource, like a CPU, but rather as an I/O device. In this work, we propose a new set of OS abstractions, collectively called the PTask API, to support GPUs as general purpose computing resources. As part of this work, I ported EncFS, a FUSE based encrypted file system for Linux, to CUDA framework such that it can use GPU for AES encryption and decryption. However, on Linux, this overrode kernels scheduling priority, because with several processes periodically contending for the GPU, the one invoking longer GPU kernels effectively monopolized the GPU regardless of its OS priority. PTask solved this problem. Using PTask's GPU scheduling mechanism I showed that running EncFS on GPU over CPU made a sequential read and write of a 200MB file 17% and 28% faster [SOSP'11].
- **A Context Aware Secure Ecosystem for Mobile Social Network.** With the rising popularity of social networks, people have started accessing social networking sites from anywhere, anytime, and from a variety of devices. Exploiting this ubiquity, we designed a context-aware framework that couples social networking data of an individual with her geographical location [IEEE'08]. However, the existing infrastructure to share this personalized data forces users to compromise their privacy. I designed and implemented a secure framework, which allowed interaction of social network information with location-based services, without compromising user privacy and security. Through exchanging an encrypted nonce ID associated with a verified user location, the framework allowed location-based services to query its vicinity for relevant information without disclosing user identity [BookChapter'12].

## Future work

With the continuing growth of the software industry with many different applications, programming paradigm, and platforms flooding the software ecosystem every day, numerous interesting research questions on improving software quality and developer productivity remained open. Here, I summarize some of the specific research directions that I would like to investigate in the future.

### Naturalness in software

From my previous work on repetitive changes in software, I realized that software, in general, lacks uniqueness. Researchers in my field have further observed that real software, the kind programmers produce to solve real-world problems, tends to be "natural" like speech or natural languages [NLS]. In fact, the average entropy of source code is much less than the average entropy of a plain English text, proving the high degree of predictability and amenable to large-sample statistical methods. In the future, I would like to explore such naturalness of programming languages to understand their properties. Similar to widespread applications of Natural Language Processing, I vision a plethora of software engineering applications exploiting the naturalness of software: code suggestion engines, porting tools, bug finding tools, tool to aid code learning, *etc*; few of them have been investigated by existing researchers. I am particularly interested in building a fault-localization tool—how naturalness of code can be exploited to locate bugs. Since a source code is in general natural, it also inherently suggests that code that is surprising, or unpredictable is suspicious; indeed, syntax errors have been found to be unnatural when using standard entropy measures. I am planning to focus specifically on the language statistics of a large corpus of bug fix commits and check whether bugginess of code can be explained from inherent language property. This work will have two important implications: 1) entropy may be a reasonable (language-independent, simpler) alternative way to draw programmers attention to a problematic code *w.r.t.* light weight static analysis tools like PMD and FindBug [SFB]; 2) search-based bug-fixing methods may benefit from using entropy both for fault-localization and searching for fixes.

### Software ecosystem

My previous work on forked software and Andriod API evolution led me to believe that the way the peer projects interact in a software ecosystem is not efficient [FSE'12; ICSM'13]. There is significant latency and redundancy involved in the process. I would like to investigate how the overall growth of an ecosystem can be achieved. For example, how can we automatically port relevant patches from peer projects in a timely and reliable fashion? How can we facilitate a timely and risk-free adoption of new APIs? I would further like to understand how similar are these peer projects in terms of socio-economic dynamics. For instance, why certain application in Android ecosystem succeed while others fail? Whether two similar projects face similar issues, email discussion, *etc*. Finally, I would like to understand how the peer projects can be benefited from each other.

In general, I will continue working towards better quality software and more productive developers. I would like to collaborate with researchers from systems, security, and natural language processing towards achieving this goal. I believe my software engineering background and multi-disciplinary collaborative experience will help me to achieve this goal.

## References


[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 20th International Symposium on the Foundations of Software Engineering*. FSE 2014.

[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.

[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.

[FSE-Demo'12] B. Ray, C. Wiley, and M. Kim. "Repertoire: A cross-system porting analysis tool for forked software projects". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE 2012.

[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.

[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.

[SOSP'11] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. "PTask: Operating system abstractions to manage GPUs as compute devices". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 233–248.

[MSRTR'14] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. "The Uniqueness of Changes: Characteristics and Applications". In: *Microsoft Research Technical Report*. 2014.

[IEEE'08] A. Beach, M. Gartrell, S. Akkala, J. Elston, J. Kelley, K. Nishimoto, B. Ray, S. Razgulin, K. Sundaresan, B. Surendar, et al. "Whozthat? evolving an ecosystem for context-aware mobile social networks". In: *Network, IEEE* 22.4 (2008), pp. 50–55.

[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.

[UCTRa'14] C. Casey, D. Prem, O. Abilio, F. Vladimir, and B. Ray. *Assert Use in GitHub Projects*. Tech. rep. University of California, Davis, 2014.

[UCTRb'14] V. Bogdan, D. Posnett, B. Ray, M. v. d. Brand, Filkov, A. Serebrenik, D. Premkumar, and V. Filkov. *Gender and Tenure Diversity in GitHub Teams*. Tech. rep. University of California, Davis, 2014.

[BookChapter'12] B. Ray and R. Han. "SecureWear: A Framework for Securing Mobile Social Networks". In: *Advances in Computer Science and Information Technology. Computer Science and Engineering*. Vol. 85. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2012, pp. 515–524.

[NLS] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. "On the naturalness of software". In: *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE. 2012, pp. 837–847.

[SFB] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. "Using static analysis to find bugs". In: *IEEE Software* 25.5 (2008), pp. 22–29.