Computer systems have evolved significantly along two lines: (1) **cloud-scale computing**, moving into large data centers and scaling out to process a massive amount of data while providing high-availability, and (2) **open-platform mobile computing**, moving into small mobile devices and hosting a wide variety of third-party applications while managing constrained resources. These advances foster products that are pervasive in society, *e.g.*, Google, Facebook, AWS, iOS and Android. However, developing and operating these systems have also become considerably complex. This complexity gives rise to new challenges in ensuring the quality of systems. Many of the challenges could not be well addressed by existing techniques.

My research goal is to enable building reliable, efficient, and defensible systems in new computing platforms like cloud and mobile. A meta challenge in achieving this goal lies in identifying and understanding emerging problems of quality control for new-platform systems. To take on this meta challenge, I adopt a pragmatic approach and sift through many real-world issues. For example, I studied service outages in industrial–strength cloud systems [4]; I analyzed thousands of posts about user-reported disruptive mobile app behaviors [7, 8]; and I interviewed professional performance analysts in a commercial data warehousing system [5, 6]. The understandings gained from immersing in real-world issues provide me with insights to solve the pressing problems in new-platform systems.

In my dissertation work, I focused on addressing the reliability challenges in cloud services. I first analyzed the distinctive characteristics of failures in commercial cloud systems and sought out answers for why failures were not tolerated and what could be improved. I then tackled misconfiguration, a prevalent fault in cloud, by designing a specification language and a framework for practitioners to efficiently validate configuration. The work generated a lot of discussion within Microsoft. Besides my dissertation work, I am also interested in solving other problems that match my research goal. For example, I recently study disruptive app behaviors and explore making mobile OS defensive to prevent the impact of bad apps.

For a particular type of system problem, I attack it from two ends. First, I seek proactive methods to prevent certain system faults in the first place, as opposed to adopting reactive break-fix methods. Second, for some defects, I accept that they are inevitable in systems and resort to mitigating the impact of defects with runtime solutions. In pursuing such methodology throughout my research projects, I drew upon diverse techniques including program analysis [ICSE ’14, SOSP ’13, OSDI ’12], language design [EuroSys ’15], machine learning [UCSD TR1014, NSDI ’13], OS [In Submission], and distributed systems [OSDI ’14].

One distinguishing characteristic of my work is that I have close collaboration with industry including Microsoft, Facebook and Teradata. The collaboration not only enabled me to work on important research problems that matter in industry settings, but also provided means for me to evaluate my solutions in the real world and achieve broad impact. In addition, I made most of the software artifacts and datasets in my work publicly available because I believe that open-source stimulates collaboration and enhances impact.

### Fault and fault tolerance in cloud services

Cloud-scale systems are built with hundreds of thousands of commodity servers. Operating at this scale, faults are inevitable. For example, a rare race condition in Linux 3.10 was hit 500 times per day in Facebook [2]. To cope with this fact of life, cloud designers harden the systems with various fault handling techniques to detect, tolerate and recover from faults as well as using careful software engineering, testing to catch faults before they manifest as failures. Nevertheless, all cloud-scale systems continue to experience failures. For example, AWS experienced a multi-day outage that impacted many of its customers [1].

This reality motivates me to work on the cloud failure analysis project [4]. Obviously, real failure data is indispensable for the project. There are a few public posts about service disruptions in major cloud vendors. However, they are insufficient to gain deep understanding about the problems. Fortunately, Microsoft is willing to collaborate with us and provide us with a snapshot of a thirteen-month period of
service disruption data. I spent a summer in Microsoft Research and the Microsoft Azure team to work with experts and decode the internals of the failure cases in the dataset. Many of the failure cases are eye-opening. After the internship, I continued working on digesting the data to extract insights.

Analyzing failures for systems at this scale is challenging. There are a number of prior work studying failures in small-scale systems. The typical methodology is to categorize the failure root causes. Because failure is a norm in cloud services, simply adopting this methodology for our project would not produce much insight for cloud designers to reduce the prevalence of these failures in the future. I took a new approach to understand cloud failures by seeking answers for three key questions: a) Why do failures occur even in systems that are designed for fault tolerance and equipped with many fault handling techniques? b) What kind of faults are especially hard to tolerate? c) Which parts are still lacking and need to be improved?

I devised a novel framework from the above angles to anatomize the failures for fault-tolerant systems like cloud services. The framework also looks into aspects such as whether a fault is contained and what propagation patterns lead to the eventual failure. I further zoomed into misconfiguration in the data since it is a dominant fault that is hard to tolerate. Our study triggered a wide discussion within Microsoft Azure. The result was summarized in a research paper [4] that was accepted by OSDI 2014 with high review scores. Unfortunately, in the end, Microsoft Azure upper management requested the paper to be withdrawn from publication due to concerns about data sensitivity.

**Configuration management in cloud services**

The cloud failure analysis project reveals several areas that need improvement to enhance cloud system reliability. Among them, configuration management is one that interests me most since we find that misconfiguration is responsible for the majority of untolerable faults, high-severity incidents, and overall unavailability. Therefore, I did a second internship in Microsoft Research and the Microsoft Azure configuration team. The goal is to develop solutions to improve configuration management. This leads to the ConfValley project [3].

Prior to ConfValley, I worked on the SPEX project [10] that used static analysis to extract configuration constraints from source code and test software resilience for misconfiguration using the constraints. At the beginning of the ConfValley project, I explored whether it is possible to adopt a similar approach. I then realized that in cloud environment, static analysis faces scalability challenges as well as fragmentation challenges: the system consists of many components that are written in different languages, maintained by different teams who use configurations differently.

I talked with operators to seek other solutions and found an under-explored area in configuration management: **configuration validation**. Configuration validation checks if configurations satisfy rules such as a parameter should be a unique IP address. It is a proactive way to prevent misconfiguration. The pre-defined rule is also useful to diagnose misconfiguration. However, the current practice of configuration validation is inefficient and *ad hoc*. It uses bulky validation code that is hard to maintain and reuse.

I have the vision that, with right tools, configuration validation can be made an ordinary deployment process to prevent misconfiguration. Towards this end, I developed a declarative specification language, CPL, for practitioners to easily describe various validation requirements. The language decouples core validation logic from implementation details, allowing the specifications to be described independently of underlying configuration representations. In addition to the new language, ConfValley also includes an inference component to automatically generate specifications based on history and configuration instances.

Our new language can express the existing imperative validation code used inside Microsoft Azure and two open-source cloud systems with more than 10X reduction in lines of code. The new validation code is easier to read and reuse. ConfValley detects a number of errors in production configuration data in Azure.

**Defensible mobile system**

Mobile devices have transformed from a closed-platform with proprietary applications to an open-platform hosting an enormous amount of third-party applications. Many people including teenagers [9] set about
developing mobile apps. Unfortunately, due to developer inexperience and a lack of development resources, mobile apps are typically weaker in terms of quality compared to traditional desktop and server applications. Consequently, many mobile users encounter frustrating experiences due to app misbehaviors, such as fast battery drain and excessive cellular data consumption. As a smartphone user, I believe that the mobile system should be made defensible enough to take care of these app misbehaviors. The system should protect users from these negative experiences.

**eDoctor:** In the eDoctor project [8], we address the Abnormal Battery Drain (ABD) problem, which is among the top complaints of negative smartphone user experiences. ABD refers to an abnormal draining of battery that is not caused by regular resource usage. From a user’s point of view, the device used to have reasonable battery life, but at some point the battery runs out unexpectedly faster than usual.

eDoctor is designed to automatically diagnose ABD issues for users and suggest repair solutions. To do so, eDoctor leverages a concept of execution phases to capture an app’s time-varying behavior, which can then be used to identify suspicious apps. In addition, eDoctor tracks important events such as app upgrading or user configuration change, so that once it identifies suspicious apps it can narrow down the causing events to suggest appropriate repair solutions to users. In a controlled in-lab experiment and user study, eDoctor successfully diagnosed 47 out of 50 issues and imposed no more than 1.5% power overhead.

**DefDroid:** eDoctor is effective in diagnosing ABD issues. But some problems are still unsolved. Specifically, eDoctor still needs constant user involvement in the diagnosis and is often too late to prevent the damage. In addition, ABD is in fact part of a more generic class of mobile app misbehavior – Disruptive App Behavior (DAB). DAB refers to apps acting immaturely to its hosting ecosystem and adversely affecting other apps running on the same device. For example, an app may keep the device awake for too long, forget to unregister GPS after use, and download large data over cellular networks without user consent. These disruptive behaviors waste or overuse resources without considering other apps, and cause significant battery drain, excessive consumption of bandwidth or storage, etc.

I designed DefDroid [7] to attack DAB issues from the OS level and proactively protect users from the bad effect of DAB while minimizing user involvement. My insight is that a lot of the DAB issues arise because the current mobile OS is too liberal. I believe that in an open-platform of third-party applications, some of which are written by amateur developers, the OS should be much more defensive than a traditional OS. The key idea of DefDroid is to use a runtime defense service controlling multiple defense modules that perform fine-grained defense actions such as releasing long-held wakelocks to misbehaving apps when necessary. In evaluation, DefDroid successfully limited the negative impact of 125 out of 128 real-world DAB issues. To further evaluate how DefDroid performs in the field, I have deployed DefDroid to more than 180 real users. Interestingly, DefDroid so far has found 6 new unknown DAB issues in the field.

**Efficient performance testing**

Software today is evolving rapidly. Some code changes, while preserving software functionality, can significantly degrade performance. Performance testing is an effective way to reveal such issues in early stages. However, performance testing faces two main challenges: 1). testing is too costly to be conducted on every commit; 2). testing can produce a large volume of results that are time-consuming to analyze.

**PerfScope:** The PerfScope project [5] is designed to address the first issue. My insight to tackle this problem is that not all commits warrant performance testing, e.g., commits touching cold path. But existing practices all treat the testing target—code commits—as a black-box, and blindly carry out testing even for commits that are unlikely to introduce performance regression. I believe it is useful to examine the content of testing targets and exploit the information to prioritize performance testing on risky commits first. PerfScope implements this new testing target prioritization approach with a lightweight performance risk analysis (PRA) that statically analyzes source code change and estimates the risk of this revision in introducing performance regression. Based on such estimation, PerfScope prioritizes performance testing to conduct comprehensive tests for high-risk revisions while lowering the testing cost for low-risk ones.

**CPAoracle:** The CPAoracle project [6] tackles the second issue of analyzing the large volume of perfor-
mance testing data. In practice, performance data is often manually analyzed in the context of a baseline measurement. I leveraged machine learning to build an automated analysis engine. It compares a given set of performance data with a stable baseline and judges whether the performance data is abnormal or not. An interesting part of the project was that when I initially applied standard learning algorithms on the performance data from a commercial system in Teradata, I met with discouraging results. The state-of-the-art classifiers have an especially high false negative rate, meaning many performance regression defects would be missed. I revisited the standard solutions and the data itself, and found several issues that were not addressed before. After addressing these issues, the analysis engine achieved a high balanced accuracy.

Future Research Directions

I am excited to continue working on systems, in particular on pursuing my goal of improving system quality in new computing platforms. This is an area that embraces many fundamental challenges. To take on these challenges, I would build on my past experiences, leverage my close collaboration with industry, and seek practical solutions that have broad impact.

**New abstractions and tooling support for configurations.** Compared to source code, the language support for configuration seriously lags behind. Developers have many ease-of-use programming languages to choose from when writing code. But there are few languages for configuration, and most of them are too low-level (e.g., imperative router settings). This is the root cause for many misconfigurations. A good configuration language should make it easy to compose, as well as make it hard to misuse. I would like to explore new language abstractions for configurations. The initial success of CPL in our ConfValley project [3] is encouraging for me to pursue this direction.

Additionally, the tooling support for configuration is poor. When writing code, developers have a powerful tool chain at their disposal: IDEs like Eclipse, debuggers like gdb, and build tools like CMake. These tools make it possible to write large programs. But few tools exist for configurations. For example, even for editing configurations, operators often only have Notepad/Vim. I intend to build a complete tool chain around new configuration languages for operators to write, validate, refactor and debug configurations.

**Reducing security-related misconfiguration.** Large systems are facing increasing threats. Even if the code is free from vulnerabilities, improper configurations can also lead to security holes. For example, misconfigured certificates, firewall rules, and ACL in cloud service can cause the systems and user data to be compromised. There has been extensive research on detecting vulnerabilities at the source code level. But the misconfiguration aspect of security threat is under-explored. I intend to investigate this problem domain and develop practical solutions to reduce misconfiguration-induced security issues. One particular direction that I’m interested is providing new language abstractions to make it easier for operators to write security policies as well as providing strong security guarantees.

**Better APIs for developers to write well-behaved apps.** As mobile developer population grows rapidly and embraces developers from a diverse background, ensuring rigorous training in mobile development becomes challenging. This has led to a rise of inexperienced developers who are writing apps. In the eDoctor and DefDroid project, I feel the struggle from inexperienced developers. In part, I think the struggle is due to the interfaces in the mobile development framework itself. For example, the interfaces require releasing wake lock for each acquiring of wake lock. An inexperienced developer can easily forget to release the lock in some code branch or release the lock too late. I am interested in creating new mobile app interfaces and high-level libraries to make the APIs less susceptible to developer mistakes.

**Defensible systems against malicious apps.** With abundant third-party apps and inexperienced developers, open-platforms like smartphones harbor many app misbehaviors. While prior solutions have explored using tools to detect app misbehaviors, I envision that more system support would be added to harden the last level of defense and prevent user frustrations. In DefDroid, I explored enhancing the mobile OS to address disruptive app behaviors. The main target was immature apps caused by programming mistakes. In the future, I intend to take a step further to look into hardening the OS to mitigate security and privacy related misbehaviors caused by malicious apps, which is a growing concern for open-platforms.
References


