# Nathaniel Ayewah

**RESEARCH STATEMENT**

My research focuses on understanding the impact of static analysis tools in practice. Static analysis tools can identify defects in software applications without running them, but using these tools effectively presents some challenges. Through my research, I am learning how real organizations deal (sometimes unsuccessfully) with these challenges. I study developers directly by observing, interviewing or surveying them, and indirectly by examining artifacts such as code repositories, bug databases or documentation. These studies have led me to develop "best practices" for software teams and identify ways to improve tools. In particular, I am developing a novel approach to teach a static analysis tool to find more defects by providing examples and counterexamples. The goal is to make it simple for non-expert developers to extend tools and find custom bug patterns. In the future, I hope to apply lessons from this research to other programming language technologies. For example, I would like to study domain specific languages (DSLs) in practice to identify common problems in their design and usage, and to learn from the experiences of organizations that have adopted this technology.

---

Static analysis tools offer a lot of promise. They can quickly find problems anywhere in code without needing to execute it. Their search is exhaustive for some classes of problems and they build on the wisdom of experts to identify problems developers may not be aware of. Many tools can be extended to find new classes of bugs and some tools analyze binaries, which is useful when source code is unavailable.

Despite this promise, static analysis tools can be challenging to use in practice. Sometimes assumptions made by the analysis are incorrect, leading to false positives. Sometimes warnings are found in code that is owned by someone else and the analyst may not understand the context of the warning. Unlike unit testing, some mistakes found by static analysis do not cause incorrect behavior. For example, a potential problem may be masked by surrounding code, or may occur in dead code, or may even have the unintended effect of making the program function correctly! Even when a warning identifies a real defect, it may not represent a quality dimension an organization is interested in. For example, internationalization warnings may not be relevant for applications that only expect to run in one locale.

As part of my research, I have sought to understand the overall *value* of static analysis tools. What proportion of analysis warnings actually signal incorrect behavior in practice and can these be found by other quality assurance methods at comparable cost? I also study the bugs that occur in practice, and the choices developers make about which ones to fix. In other words, *which warnings matter*? Ultimately, I wish to identify best practices that increase successful adoption of tools. In some ways this research is similar to efforts to improve spell checkers in word processors or spreadsheets by studying the habits of users. Unlike spell checkers though, static analysis tools demand more human investment and infrastructure to identify and remediate significant warnings.

My research practice alternates between direct interactions with users which yield mostly qualitative and anecdotal information, and substantial studies of code artifacts and bug reports which yield mostly quantitative data. Analyzing large software artifacts reveals significant trends that we can generalize, but tells us little about why these trends are observed. We fill this gap by directly interacting with users and organizations. Conversely, lab studies and user interviews help us generate hypotheses which we can then investigate quantitatively.

Most of my studies have been conducted using FindBugs, an open source static analysis tool for Java from the University of Maryland, which has been downloaded more than a million times and is used by companies such as Google, EBay, Amazon, Sun, and Oracle. I have surveyed about a thousand FindBugs users, visited organizations that use FindBugs, interviewed several dozen developers, and conducted lab studies with students. I have also manually inspected hundreds of warnings in various code bases, developed techniques to automatically mine software repositories and bug reports, and made technical contributions to FindBugs' analysis. Given FindBugs' strong focus on defects associated with code quality, I have rounded out my research by working with static analysis tools that have a stronger focus on code security, including tools from Fortify Software and Coverity.

In my studies, I observed that static analysis tools are not adopted fully and used consistently unless they are run automatically as part of the software development process. Many FindBugs users have not yet established the infrastructure to do this, and hence are not retaining its full value. The main challenges cited by users are resolving the large number of initial warnings, integrating multiple static analysis tools into a consistent interface, and customizing tools to filter out undesired warning patterns and add project specific ones. Many commercial tools enable users to review warnings communally, track issues in bug databases, and better integrate tools into an existing process. FindBugs has recently introduced some of these features.

I have also observed that various quality assurance techniques eventually find most of the consequential quality problems flagged by static analysis tools. The value of static analysis is that it can catch these significant problems *early* in the development cycle when they are cheaper to fix. This finding does not apply to certain classes of security and concurrency warnings which tend elude other quality assurance methods. In addition, this finding may change the paradigm of some developers who perceive static analysis tools primarily as a way to find any problems they missed. Efficient static analysis could save them time by finding problems sooner.

A consequence of this observation is that when we analyze software that has been in production for some time, we often find real bugs that are mostly harmless. For example, we recently conducted a large FindBugs review at Google in which hundreds of engineers reviewed thousands of warnings. The feedback was positive, the managers were impressed, and many defects were fixed, but we did not observe many issues that caused serious problems in production. In one case, a bug detected by FindBugs caused some internal runs to fail, but the change that introduced the bug was automatically rolled back. If the responsible engineers had used FindBugs, they might have avoided the delay and inconvenience, but the real impact of the bug was limited.

Through the Google review and some lab studies, I have observed that developers tend to perceive *noisy* warnings (that manifest as exceptions or crashes) to be more severe than *silent* warnings (that quietly compute incorrect values). But noisy warnings in mature software tend to occur in dead code or infeasible scenarios and hence are often impotent. Silent issues may indicate a logic error or contradicting invariants, and reviewing them may lead to discovery of deeper bugs. For example, a *Redundant Comparison to Null* warning occurs when a value is compared to null moments after it is dereferenced, with no intervening assignments. If the value was null, then the dereference would have caused an exception, so the check is redundant. The developer is effectively asserting contradicting invariants by assuming the value cannot be null at the dereference site, but then assuming it may be null soon after. Many times the value cannot be null, and the comparison is defensive and harmless, but sometimes we observe errors including accidentally using the wrong variable in the comparison. This finding is useful to teams needing to review thousands of warnings – they may get more bang for their buck by addressing some silent issues, even though they look unimportant. FindBugs has updated its rankings to factor this in.

I have also observed that many null pointer errors are not caused by mishandled null values, but by deeper logic errors which eventually manifest as null pointer exceptions. This finding is relevant when static analysis reports the dereference of a potentially null value. The analysis may suggest that the developer has erred by dereferencing this value without a guard but the developer may implicitly assume that the value will never be null. For example, some operations on a Java File instance may lead to a null pointer exception if the instance does not represent a directory. Some static analysis tools report this warning and push developers to introduce guards around these operations. But using a guard is redundant if the developer believes the instance will always be a directory. If there is a fault in the code, it must be an unrelated logic error that causes the developer's belief to be violated. Often the developer's only recourse is to throw a different exception and this leads to code that is harder to read and maintain.

---

My current research activity is identifying ways to extend FindBugs, and is motivated in part by observations from our studies. I have observed that many survey participants had never tried to extend FindBugs to find custom bug patterns. But I have also observed that many projects and libraries have *project-specific* defects that recur often. Usually these are mistakes that are easy to make due to unspecified or loosely specified API rules. In addition, within an organization, a team may want to deploy custom bug detectors that are run locally and not across the organization. These observations have led me to explore a novel way to specify bug detectors by giving examples and counterexamples. FindBugs parses these examples and refines a bug detector so that it flags all the "bug" examples, but does not flag any of the "not bug" counterexamples. The goal is to encourage more non-expert users to create bug detectors that will only be used within their teams or organizations.

Going forward, I hope to study other programming language technologies, looking for ways to make them better for their users. I am particularly interested in Domain Specific Languages (DSLs) which enable users to program at a higher abstraction level. I would like to observe the language creation

process, documenting in detail its activities and challenges. I would also like to observe the process of learning and using a new language to understand how much effort is involved and how this compares to regular languages. I can also examine bug reports from projects that use DSLs to discover the sorts of bugs that occur, determine if they may be caused by poor language design or incorrect usage, and search for common bug patterns that cut across multiple DSLs or multiple programs written in one DSL. If a particular bug pattern recurs often, it may be possible and beneficial to create a higher level DSL construct that eliminates the problem.

I am also interested in working with *end user programmers*: non expert users who do light programming tasks to serve some personal need. This includes spreadsheet authors and simple "web mashup" creators. These users are not inclined to use rigorous software engineering methodologies and may benefit from lightweight tools that make their task less error prone. I hope to discover the sorts of mistakes these users make and look for recurring bug patterns within a given API or language, or across multiple platforms.

Through these studies and interaction with real users, I am learning how to make programming language technologies more useful. But I am not just interested in user studies and mining user data. I hope to use insights from these studies to innovate new technologies or components. My activities will build on my interdisciplinary research background which includes projects on unit testing concurrent abstractions, information visualization, noise reduction in hearing aids, machine learning and data mining. Ultimately, I want to participate in projects that create or refine technologies to make users more creative and productive.