Improving Software Quality with Static Analysis [*]

Jeffrey S. Foster Michael W. Hicks William Pugh
University of Maryland, College Park
{jfoster,mwh,pugh}@cs.umd.edu

Abstract

At the University of Maryland, we have been working to improve the reliability and security of software by developing new, effective static analysis tools. These tools scan software for bug patterns or show that the software is free from a particular class of defects. There are two themes common to our different projects:

1. Our ultimate focus is on *utility*: can a programmer actually improve the quality of his or her software using an analysis tool? The important first step toward answering this question is to engineer tools so that they can analyze existing, nontrivial programs, and to carefully report the results of such analyses experimentally. The desire to better understand a more human-centered notion of utility underlies much of our future work.
2. We release all of our tools open source.[1] This allows other researchers to verify our results, and to reuse some or all of our implementations, which often required significant effort to engineer. We believe that releasing source code is important for accelerating the pace of research results software quality, and just as importantly allows feedback from the wider community.

In this research group presentation, we summarize some recent work and sketch future directions.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.2.4 [*Software/Program Verification*]: Reliability

General Terms Algorithms, Reliability, Security

Keywords Bugs, bug patterns, FFIs, network protocols, data races, modularity, C, Java, software quality

1. Recent Work

Our focus has been to develop tools that find problems in existing systems, with a low burden on the programmer (e.g., no or few annotations, being fully automatic, producing few false positives, etc.). Among others, we have developed the following five tools.

FindBugs FindBugs [4] is a static analysis tool that finds coding mistakes or defects in Java programs. The approach taken by the FindBugs project is to start with real bugs in real software, abstract a bug pattern from those bugs, and devise the simplest possible detector that can effectively find that bug pattern, as evaluated by trying the proposed detector on test cases for that bug pattern as well as tens of millions of lines of production software. FindBugs now finds more than 250 bug patterns. Some bug detectors, such as those reporting null pointer errors [5], reflect lots of work over a long period of time, and find lots of errors. But there is also a long tail of bug detectors/patterns: many patterns only occur a few times per million lines of code, but in total find a substantial number of defects. Such detectors are frequently conceived, implemented, evaluated and tuned within the space of a few hours.

FindBugs can be run within multiple environments, and also supports historical tracking [8] so that the persistence of defects across successive builds of a software project can be captured.

FindBugs has been downloaded more than 300,000 times, and is now in use by many large companies and open source efforts. This provides an opportunity to study open code bases and see how the defects identified by FindBugs change over time, both in cases where FindBugs is in use and in cases where it is not. FindBugs also provides an excellent delivery vehicle for allowing new defect detection research ideas to be quickly disseminated.

Saffire Many software systems are written in multiple languages, and yet there has been little research to date in reasoning about such programs. Most multi-lingual programs are built using *foreign function interfaces* (FFIs), which define what must be done to translate between native and foreign data representations and how the foreign code should safely interact with the native code. These requirements are easy to get wrong, and mistakes often lead to subtle, hard-to-find bugs. To prevent such problems, we developed Saffire [1, 2], which checks type safety across an FFI. Saffire operates on the OCaml-to-C FFI and the Java-to-C FFI (called the Java Native Interface or JNI).

Saffire works by performing type inference on C "glue code," which in these FFIs performs most of the work mediating between the languages. Saffire infers *representational types*, which model C's low-level view of OCaml and Java values, and Saffire uses singleton types to track integers, memory offsets, type tags, and strings through C. J-Saffire, our Java system, uses a polymorphic, flow-insensitive, unification-based analysis, while O-Saffire, our OCaml system, uses a monomorphic, flow-sensitive analysis. Our experiments using Saffire are very encouraging. Across 11 programs that use the OCaml FFI, O-Saffire found 24 bugs and 22 bad coding practices (things that are not actually errors, but are not a good idea), and across 12 programs that use the JNI, J-Saffire found 156 bugs and 124 bad coding practices. The analysis is very fast, typically taking only seconds, with a low false positive rate.

Pistachio Today's software systems communicate over the Internet using standard protocols that have been heavily scrutinized, providing some assurance of resistance to malicious attacks and general robustness. However, the software that implements those protocols may still contain mistakes, and an incorrect implementation could lead to vulnerabilities even in the most well-understood protocol. We have developed a tool called Pistachio [10] that tries

[*] This research was supported in part by NSF CCF-0346982, CCF-0430118, CCF-0346989, IIS-0613601, and CCF-0541036, and gifts from Fortify Software, Sun Microsystems and Google.

[1] `http://www.cs.umd.edu/projects/PL/`

Copyright is held by the author/owner(s).
PASTE'07 June 13–14, 2007, San Diego, California, USA.
ACM 978-1-59593-595-3/07/0006.

to close this gap by checking that a C implementation of a protocol matches its description in an RFC or similar standards document.

The first component of Pistachio is a rule-based specification language that is tuned to describing network protocols. Our rule language is designed to naturally encode the kinds of requirements in protocol specifications. As an example, our specification for part of SSH2 consists of 96 rules, which took about 7 hours to develop. The second component of Pistachio is a symbolic evaluation engine that simulates the execution of program source code. Using a fully automatic theorem prover, Pistachio checks that whenever it encounters a statement that triggers a rule (typically a *receive()* call), on every path the conclusion of the rule is eventually satisfied (typically, there is a *send()* that transmits the right data). The analysis is not guaranteed sound, but in practice has few false negatives. We applied Pistachio to implementations of SSH2 and RCP, and compared its output against the projects' bug database. Overall, Pistachio found many bugs, and had approximately a 38% false positive rate and only a 5% false negative rate.

Locksmith aims to prove that multi-threaded C programs are free from data races. Locksmith statically enforces the the well-known "guarded-by" pattern [7, 3], in which each thread that accesses a memory location must do so while holding the lock that "guards" that location. LOCKSMITH aims to be sound, so that if it produces no warnings, then the program has no data races.

To begin, LOCKSMITH performs a context-sensitive, inclusion-based points-to analysis to characterize the locks and locations in the program, and uses a sharing analysis to determine which locations might be shared between threads. Given this information, we developed a novel *correlation analysis* to check that a shared location is consistently correlated with some lock. One of the challenges of achieving this result is to properly handle dynamic allocation of locks and data structures, and to properly check when a data structure might have several locks for each of its constituent parts. We adapted ideas from existential types to allow individual elements of recursive data structures to be conflated by the points-to analysis, but still be treated separately from each other when considering their synchronization behavior [6].

We have used LOCKSMITH to analyze several modestly-sized C programs, including Linux device drivers, and found several data races with a low number of false alarms.

CMod is tool that provides a sound, backward-compatible module system for C [9]. While many languages have linguistic support for modules, C programs instead rely on the preprocessor: `.h` header files act as module interfaces and `.c` source files act as module implementations. While this basic idea is well-known, the details in getting it right are not, and furthermore there is no enforcement mechanism. The result is the potential for type errors and information hiding violations, which degrade programs' modular structure and complicate maintenance.

CMOD addresses this problem by enforcing a set of four rules that are based on principles of modular reasoning and on current programming practice. The first two rules ensure that headers properly enforce *information hiding* policies: one module may only link against another module's symbol, or make use of a type defined by that module, by referring to that symbol or type via the header that acts as the module's interface. These rules also enforce that programs are *type safe* at link time (i.e., the types of shared symbols match across modules). The third and fourth rules define acceptable preprocessor usage so that the first two rules are not circumvented. To our knowledge, CMOD is the first system to enforce both information hiding and type safety for standard C programs. We evaluated CMOD on a number of benchmarks and found that most programs obey CMOD's rules, or can be made to with minimal effort, while rule violations reveal brittle coding practices including numerous information hiding violations and occasional type errors.

2. Future Directions

We think of software development as a process involving both humans (developers) and software tools (type checkers, code generators, and analysis tools). Any attempt to improve software quality needs to understand the roles each party plays and their interaction. We plan to pursue several important future directions all aimed at improving the *utility* of static analysis tools:

- What do we need to do to make a tool useful for a developer? Much academic research has focused on the fundamental algorithmic challenges of such tools rather than their usability. We need to study, both scientifically and anecdotally, what design choices make tools better or worse in practice.
- How can we give programmers more ways to customize tools to their needs? Some frameworks exist (e.g., standard types, typestate or finite state properties, and type qualifiers, to name just a few), but they capture only part of the space.
- Are the defects reported by tools important? It could be some static analysis techniques identify additional "bugs", but only find ones that developers wouldn't ordinarily bother with. This is a thorny issue, but should be studied.
- How can we help programmers understand the output of a static analysis tool? We need better ways to present static analysis results in terms of the abstractions the developer has in mind. Just as we would like to have standard tool front-ends, can we develop standard back-ends for presenting analysis results?

We are working at Maryland within the context of a large software research group, with strengths in software engineering and human-computer interaction. We feel further attention to these areas is critical to making future tools successful.

References

[1] M. Furr and J. S. Foster. Checking Type Safety of Foreign Function Calls. In *PLDI'05*, pages 62–72, Chicago, Illinois, June 2005.

[2] M. Furr and J. S. Foster. Polymorphic Type Inference for the JNI. In *ESOP'06*, pages 309–324, Vienna, Austria, 2006.

[3] M. Hicks, J. S. Foster, and P. Pratikakis. Lock Inference for Atomic Sections. In *TRANSACT'06*, Ottawa, Canada, June 2006.

[4] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Onward!, OOPSLA'04*, Vancouver, BC, October 2004.

[5] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE'05*, pages 13–19, 2005.

[6] P. Pratikakis, J. S. Foster, and M. Hicks. Existential Label Flow Inference via CFL Reachability. In *SAS'06*, pages 88–106, Seoul, Korea, Aug. 2006.

[7] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI'06*, pages 320–331, Ottawa, Canada, June 2006.

[8] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *MSR'06*, pages 133–136, 2006.

[9] S. Srivastava, M. Hicks, and J. S. Foster. Modluary Information Hiding and Type-Safe Linking for C. In *TLDI'07*, pages 3–13, Nice, France, Jan. 2007.

[10] O. Udrea, C. Lumezanu, and J. S. Foster. Rule-Based Static Analysis of Network Protocol Implementations. In *USENIX Security'06*, pages 193–208, Vancouver, British Columbia, Canada, Aug. 2006.