Residual Investigation: Predictive and Precise Bug Detection

Kaituo Li[1], Christoph Reichenbach[2], Christoph Csallner[3], Yannis Smaragdakis[4]

Dept. of Comp. Science[1] & FB 12/Informatik[2] &
Dept. of CS/Eng.[3] & Dept. of Informatics & Telecommunication[4]
University of Massachusetts, Amherst[1] & Goethe University Frankfurt[2] &
University of Texas at Arlington[3] & University of Athens[4]
140 Governors Drive[1] & Robert-Mayer-Str. 11–15[2] &
Eng. Research Building, Room 640, Box 19015[3] & Panepistimiopolis, Ilisia[4]
01300 Amherst, MA[1] & 60054 Frankfurt[2] & 76010 Arlington, TX[3] & 157 84 Athens[4]
kaituo@cs.umass.edu
reichenbach@cs.uni-frankfurt.de
csallner@exchange.uta.edu
smaragd@di.uoa.gr

Abstract: We introduce the concept of "residual investigation" for program analysis. A residual investigation is a dynamic check installed as a result of running a static analysis that reports a possible program error. The purpose is to observe conditions that indicate whether the statically predicted program fault is likely to be realizable and relevant. The key feature of a residual investigation is that it has to be much more precise (i.e., with fewer false warnings) than the static analysis alone, yet significantly more general (i.e., reporting more errors) than the dynamic tests in the program's test suite that are pertinent to the statically reported error. That is, good residual investigations encode dynamic conditions that, when considered in conjunction with the static error report, increase confidence in the existence or severity of an error without needing to directly observe a fault resulting from the error.

We enhance the static analyser FindBugs with several residual investigations, appropriately tuned to the static error patterns in FindBugs, and apply it to 9 large open-source systems and their native test suites. The result is an analysis with a low occurrence of false warnings ("false positives") while reporting several actual errors that would not have been detected by mere execution of a program's test suite.

Static analysis can be invaluable for detecting software bugs. For example, consider the following method:

```
public byte getTenthByte(java.io.InputStream istream) {
  byte[] a = new byte[10];
  istream.read(a);   // try to read 10 bytes
  return a[9];       // return last byte
}
```

This method reads ten bytes from an instance of Java's `java.io.InputStream` class and returns the last byte read. Here, `istream.read(a)` will try to read as many bytes as

it can fit into `a` and return the number of bytes actually read—possibly less than 10. If we are near the end of the input stream, `istream.read(a)` may return a smaller number, and `a[9]` will contain garbage. Fortunately, static checkerers such as FindBugs [HP04] can detect such bugs (marked '*Read Return Should Be Checked*', in FindBugs).

Unfortunately, static analyses tend to suffer from *false positives* [HP04], i.e., they issue warnings that do not reflect reality. This is also true in our particular example: the Eclipse system provides a subclass of `java.io.InputStream` whose `read` method always returns the size of the array as return value, causing many spurious FindBugs warnings.

To eliminate such warnings, we propose *residual investigation*. A residual investigation is a dynamic analysis that serves as the run-time agent of a static analysis. Its purpose is to determine with higher certainty whether the error identified by the static analysis is likely true. The dynamic analysis serves as 'residual' of the static analysis at a subsequent stage: that of program execution. Unlike past static-dynamic combinations, residual investigation does not intend to report the error only if it actually occurs, but to identify general conditions that reinforce statically detected errors. A residual investigation is thus a predictive dynamic analysis, predicting errors in executions not actually observed. This permits us to increase confidence in error reports even with relatively small test suites .

In our example, the residual investigation consists of two checks: (a) a dynamic check that checks for each subclass C of `java.io.InputStream` if that particular subclass' `read` method ever reads fewer than the maximum number of bytes, and (b) a dynamic check that determines, for each program location ℓ that triggers a `read` and ignores its return value, which dynamic types ℓ_C we observe for the input stream. We then reinforce the bug report at ℓ iff there exists a $C \in \ell_C$ that satisfies property (a). Note that this is not the same as observing an actual fault: we may have observed property (a) in an unrelated piece of code. This is what makes our analysis *predictive*: Knowing that C has property (a) justifies concern about the `read` return value, and knowing that $C \in \ell_C$ justifies warning the user that a fault here is more plausible than suggested by a purely static analysis.

Our full paper [LRCS15] explores residual investigation on six FindBugs patterns as well as on static race detection. We find our approach to be highly effective at eliminating false positives and highlighting actual bugs among static bug reports without needing to observe real faults. For example, FindBugs with residual investigation lowers the false positive rate from $\geq 90\%$ to $\leq 23\%$ in several large Open Source systems. We observe similarly dramatic improvements with race detection.

References

[HP04] David Hovemeyer und William Pugh. Finding bugs is easy. In *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 132–136. ACM, Oktober 2004.

[LRCS15] Kaituo Li, Christoph Reichenbach, Christoph Csallner und Yannis Smaragdakis. Predictive and Precise Bug Detection. *ACM Transactions on Sotftware Engineering and Methodology*, Seite to appear, 2015.