Residual Investigation: Predictive and Precise Bug Detection

Kaituo Li,
Christoph Reichenbach*
Computer Science Dept.
University of Massachusetts
Amherst, MA 01003, USA
kaituo@cs.umass.edu,
creichen@gmail.com

Christoph Csallner
Department of Computer Science and Engineering
University of Texas at Arlington,
Arlington, TX 76019, USA
csallner@uta.edu

Yannis Smaragdakis
Dept. of Informatics,
U. of Athens
Athens 15784, Greece
and Comp. Sci.,
U. Massachusetts
Amherst, MA 01003, USA
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 pertinent to the statically reported error. That is, good residual investigations encode dynamic conditions that, when taken in conjunction with the static error report, increase confidence in the existence of an error, as well as its severity, without needing to directly observe a fault resulting from the error.

We enhance the static analyzer FindBugs with several residual investigations, appropriately tuned to the static error patterns in FindBugs, and apply it to 7 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.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Design, Reliability, Verification

Keywords

False warnings, existing test cases, RFBI

*Current affiliation: Google Inc., Mountain View, CA, USA

ISSTA '12, July 15–20, 2012, Minneapolis, MN, USA

1. INTRODUCTION AND MOTIVATION

False error reports are the bane of automatic bug detection—this experience is perhaps the most often-reported in the program analysis research literature [1, 21, 22, 27, 28]. Programmers are quickly frustrated and much less likely to trust an automatic tool if they observe that reported errors are often not real errors, or are largely irrelevant in the given context. This is in contrast to error detection at early stages of program development, where guarantees of detecting all errors of a certain class (e.g., type soundness guarantees) are desirable. Programmers typically welcome conservative sanity checking while the code is actively being developed, but prefer later warnings (which have a high cost of investigation) to be with much higher confidence, even at the expense of possibly missing errors.

The need to reduce false or low-value warnings raises difficulties especially for static tools, which, by nature, overapproximate program behavior. This has led researchers to devise combinations of static analyses and dynamic observation of faults (e.g., [6, 9, 10, 12, 17, 25, 26]) in order to achieve higher certainty than purely static approaches.

In this paper we present a new kind of combination of static and dynamic analyses that we term *residual investigation*. A residual investigation is a dynamic analysis that serves as the run-time agent of a static analysis and to discern with higher certainty whether the error identified by the static analysis is likely true. In other words, one can see the dynamic analysis as the "residual" of the static analysis at a subsequent stage: that of program execution. The distinguishing feature of a residual investigation, compared to past static-dynamic combinations, is that the residual investigation does not intend to report the error only if it actually occurs, but to identify *general* conditions that confirm the statically detected error. That is, a residual investigation is a *predictive* dynamic analysis, predicting errors in executions not actually observed.

Consider as an example the "equal objects must have equal hashcodes" analysis (codenamed HE) in the FindBugs static error detector for Java [1, 14, 16]. The HE analysis emits a warning whenever a class overrides the method `equals(Object)` (originally defined in the `Object` class, the ancestor of all Java classes) without overriding the `hashCode()` method (or vice versa). The idea of the analysis is that the hash code value of an object should serve as an equality signature, so a class should not give a new meaning to equality without updating the meaning of `hashCode()`. An actual fault may occur if, e.g., two objects with distinct hash code values are equal as far as the `equals` method is concerned, and are used in the same hash table. This error will be hard to trace, as it can lead to a violation of fundamental program invariants. The programmer

may validly object to the error warning, however: objects of this particular class may never be used in hash tables in the current program. Our residual investigation consists of determining whether (during the execution of the usual test suite of the program) objects of the suspect class are ever used inside a hash table data structure, or otherwise have their `hashCode` method ever invoked. (The former is a strong indication of an error, the latter a slightly weaker one.) Note that this will likely not cause a failure of the current test execution: all objects inserted in the hash table may have distinct hash code values, or object identity in the hash table may not matter for the end-to-end program correctness. Yet, the fact that objects of a suspect type are used in a suspicious way is a very strong indication that the program will likely exhibit a fault for different inputs. In this way the residual investigation is both more general than mere testing, as well as more precise than static analysis.

We have designed and implemented residual investigations for several of the static analyses/bug patterns in the FindBugs system. The result is a practical static-dynamic analysis prototype tool, RFBI (for *Residual FindBugs Investigator*). Our implementation uses standard techniques for dynamic introspection and code interposition, such as bytecode rewriting and customized AspectJ aspects [18]. The addition of extra analyses is typically hindered only by engineering (i.e., implementation) overheads. Designing the residual investigation to complement a specific static pattern requires some thought, but it is typically quite feasible, by following the residual investigation guidelines outlined earlier: the analysis should be significantly more general than mere testing while also offering a strong indication that the statically predicted fault may indeed occur.

We believe that the ability to easily define such analyses is testament to the value of the concept of residual investigation. Predictive dynamic analyses are usually hard to invent. To our knowledge, there is only a small number of predictive dynamic analyses that have appeared in the research literature. (A standard example of a predictive dynamic analysis is the Eraser race detection algorithm [23]: its analysis predicts races based on inconsistent locking, even when no races have appeared in the observed execution.) In contrast, we defined seven predictive analyses in a matter of days, by merely examining the FindBugs list of bug patterns under the lens of residual investigation.

In summary, the main contributions of this work are:

- We introduce residual investigation as a general concept and illustrate its principles.
- We implement residual investigations for several of the most common analyses in the FindBugs system, such as "cloneable not implemented correctly", "dropped exception", "read return should be checked", and several more. This yields a concrete result of our work, in the form of the Residual FindBugs Investigator (RFBI) tool.
- The real experimental validation of our work consists of the ability to invent residual investigations easily. However, we also validate our expectation that the resulting analyses are useful by applying them to 7 open-source applications (including large systems, such as JBoss, Tomcat, NetBeans, and more) using their native test suites. We find that residual investigation produces numerous (31) warnings that do not correspond to test suite failures and are overwhelmingly bugs.

2. RESIDUAL INVESTIGATION

Residual investigation is a simple concept—it is a vehicle that facilitates communication rather than a technical construction with a strict definition. We next discuss its features and present the example analyses we have defined.

2.1 Preliminary Discussion

We consider a dynamic check that is tied to a static analysis to be a residual investigation if it satisfies the informal conditions outlined in the Introduction:

- The check has to identify with very high confidence that the statically predicted behavior (typically a fault[1]) is valid and relevant for actual program executions. A residual investigation should substantially reduce the number of false (or low-value) error reports of the static analysis.
- The analysis has to be predictive: it should be generalizing significantly over the observed execution. A residual investigation should recognize highly suspicious behaviors, not just executions with faults.

A bit more systematically, we can define the following predicates over a program p:

- $B(p)$, for "p has a bug", i.e., the program text contains a possible error of a kind we are concerned with (e.g., class overrides `equals` but not `hashcode`) and there is some execution e_p of p for which this error leads to a fault.
- $S(p)$, for "p induces a static error report", i.e., the program text contains a possible error that the static analysis reports.
- $T(p)$, for "p causes a test case fault", when executed with p's test suite.
- $R(p)$, for "p causes a residual investigation report", when executed with p's test suite.

We assume that our static analysis is complete for the kinds of errors we consider, and the dynamic testing is sound for the execution it examines.[2] Thus, we have:

$$\forall p : B(p) \Rightarrow S(p)$$

(by completeness of static analysis)

$$\forall p : T(p) \Rightarrow B(p)$$

(by soundness of dynamic testing).

The requirements for having a valid and useful residual investigation then become:

[1]The computing literature is remarkably inconsistent in the use of the terms "error", "fault", "failure", etc. In plain English "error" and "fault" are dictionary synonyms. Mainstream Software Engineering books offer contradicting definitions (some treat an "error" as the cause and a "fault" as the observed symptom, most do the opposite). Standard systems parlance refers indiscriminately to "bus errors" and "segmentation faults", both of which are quite similar program failures. In this paper we try to consistently treat "error" (as well as "bug" and "defect") as the cause (in the program text) of unexpected state deviation, and "fault" as the dynamic occurrence that exhibits the consequences of an error. That is we think of *programming* errors, and *execution* faults. It should also be possible for the reader to treat the two terms as synonyms.

[2]This assumption holds only for the purposes of the formalism in this sub-section. The notion of residual investigation holds perfectly well for incomplete bug finders. We view all analyses as bug detectors, not as correctness provers. Therefore soundness means that warning about an error implies it is a true error, and completeness means that having an error implies it will be reported. For a correctness prover the two notions would be exactly inverse.

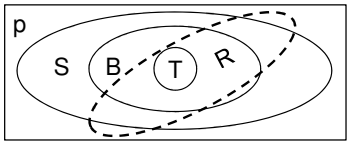

Figure 1: The goal of residual investigation (R) is to provide a filter for the static bug warnings (S), such that R and S combined (i.e., the intersection of R and S) better approximates the set of true bugs (B) than static analysis and dynamic testing (T).

1. The static analysis is unsound (i.e., some warnings are false):

   $$\exists p : S(p) \wedge \neg B(p)$$

   (We are only interested in non-trivial and therefore undecidable program properties. As our static analysis is complete, it has to be unsound. Note that since T is sound, this implies $\exists p : S(p) \wedge \neg T(p)$ i.e., the dynamic analysis also does not flag this program that the static analysis flags.)

2. The dynamic testing (of a program's test suite) is incomplete (i.e., bugs are missed by testing):

   $$\exists p : B(p) \wedge \neg T(p)$$

   (Again, the undecidability of non-trivial program properties combined with the soundness of testing implies testing is incomplete.)

3. The residual investigation should be an appropriate bridge for the gap between the static analysis and the bug (see also Figure 1):

   $$\forall p : B(p) \cong (S(p) \wedge R(p))$$

   We use $\cong$ for "approximately equivalent". This is the only informal notion in the above. It is practically impossible to have exact equivalence for realistic programs and error conditions, since $R(p)$ examines a finite number of program executions. Note that (approximate) equivalence means both that $S(p) \wedge R(p)$ (likely) implies a bug and that, if there is a bug, $R(p)$ will (likely) be true, i.e., will detect it. In practice, we place a much greater weight on the former direction of the implication. That is, we are happy to give up on completeness (which is largely unattainable anyway) to achieve (near-)soundness of error warnings.

The question then becomes: how does one identify a good residual investigation? We have used some standard steps:

- Start with the static analysis and identify under what conditions it is *inaccurate* or *irrelevant*.
- Estimate how likely these conditions can be. In other words, is this static analysis likely to yield error reports that the programmer will object to, seeing them as false or of low-value?
- If so, is there a concise set of dynamic information (other than a simple fault) that can invalidate the programmer's objections? That is, can we determine based on observable dynamic data if the likely concerns of a programmer to the static warnings do not apply?

Recognizing such "likely objections of the programmer" has been the key part in our design. With this approach we proceeded to identify residual investigations for seven static analyses in the FindBugs system, including some of the analyses that issue the most common FindBugs warnings.

2.2 Catalog of Analyses

We next present the residual investigations defined in our RFBI (Residual FindBugs Investigator) tool, each tuned to a static analysis in the FindBugs system. We list each analysis (uniquely described by the corresponding FindBugs identifier) together with the likely user *objections* we identified and a description of *clues* that dynamic analysis can give us to counter such objections. To simplify the presentation, we detail our implementation at the same time.

2.2.1 Bad Covariant Definition of Equals (Eq)

The `equals(Object)` method is defined in the `Object` class (`java.lang.Object`) and can be overridden by any Java class to supply a user-defined version of object value equality. A common mistake is that programmers write `equals` methods that accept a parameter of type other than `Object`. The typical case is that of a covariant re-definition of `equals`, where the parameter is a subtype of `Object`, as in the example class `Pixel`:

```
class Pixel {
  int x;
  int y;
  int intensity;

  boolean equals(Pixel p2)
  { return x==p2.x && y==p2.y; }
}
```

This `equals` method does not override the `equals` method in `Object` but instead redefines it for arguments of the appropriate, more specific, static type. As a result, unexpected behavior will occur at runtime, especially when an object of the class type is entered in a Collections-based data structure (e.g., `Set`, `List`). For example, if one of the instances of `Pixel` is put into an instance of a class implementing interface `Container`, then when the `equals` method is needed, `Object.equals()` will get invoked at runtime, not the version defined in `Pixel`. One of the common instances of this scenario involves invoking the `Container.contains(Object)` method. A common skeleton for `Container.contains(Object)` is:

```
boolean contains(Object newObj) {
  Iterator cur = this.iterator();
  while(cur.hasNext()) {
    if(cur.next().equals(newObj))
      return true;
  }
  return false;
}
```

Here, `contains(Object)` will use `Object.equals`, which does not perform an appropriate comparison: it compares references, not values. Therefore, objects of type `Pixel` are not compared in the way that was likely intended.

Possible programmer objections to static warnings. FindBugs issues an error report for each occurrence of a covariant definition of `equals`. Although the covariant definition of `equals` is very likely an error, it is also possible that no error will ever arise in the program. This may be an accidental artifact of the program structure, or even a result of the programmer's calculation that for objects of the suspect class the dynamic type will always be equal to the static type, for every invocation of `equals`. For instance, the redefined `equals(Pixel)` method may be used only inside class `Pixel`, with arguments that are always instances of subtypes of

`Pixel`, and the programmer may have chosen the covariant definition because it is more appropriate and convenient (e.g., obviates the need for casts).

Dynamic clues that reinforce static warnings. Our residual investigation consists of simply checking whether the ancestral `equals` method, `Object.equals(Object)`, is called on an instance of a class that has a covariant definition of `equals`. The implementation first enters suspect classes into a blacklist and then instruments all call sites of `Object.equals(Object)` to check whether the dynamic type of the receiver object is in the blacklist.

Implementation. We transform the application bytecode, using the ASM Java bytecode engineering library. Generally, for all our analyses, we instrument incrementally (i.e., when classes are loaded), except in applications that perform their own bytecode rewriting which may conflict with load-time instrumentation. In the latter case, we pre-instrument the entire code base in advance (build time).

2.2.2 Cloneable Not Implemented Correctly (CN)

Java is a language without direct memory access, hence generic object copying is done only via the convention of supplying a `clone()` method and implementing the `Cloneable` interface. Additionally, the `clone()` method has to return an object of the right dynamic type: the dynamic type of the returned object should be the same as the dynamic type of the receiver of the `clone` method and *not* the (super)class in which the executed method `clone` happened to be defined. This is supported via a user convention: any definition of `clone()` in a class `S` has to call `super.clone()` (i.e., the corresponding method in `S`'s superclass). The end result is that the (special) `clone()` method in the `java.lang.Object` class is called, and produces an object of the right dynamic type.

Possible programmer objections to static warnings. FindBugs statically detects violations of the above convention and reports an error whenever a class implements the `Cloneable` interface, but does not directly invoke `super.clone()` in its `clone` method (typically because it merely creates a new object by calling a constructor). Although this condition may at first appear to be quite accurate, in practice it often results in false error reports because the static analysis is not inter-procedural. The `clone` method may actually call `super.clone()` by means of invoking a different intermediate method that calls `super.clone()` and returns the resulting object.

Dynamic clues that reinforce static warnings. A dynamic check that determines whether a `clone` method definition is correct consists of calling `clone` on a subclass of the suspect class `S` and checking the return type (e.g., by casting and possibly receiving a `ClassCastException`). Our residual investigation introduces a fresh subclass `C` of `S` defined and used (in a minimal test case) via the general pattern:

```
class C extends S {
  public Object clone()
  { return (C) super.clone(); }
}
... ((new C()).clone()) // Exception
```

(If `S` does not have a no-argument constructor, we statically replicate in `C` all constructors with arguments and dynamically propagate the actual values of arguments used for construction of `S` objects, as observed at runtime.)

If the test case results in a `ClassCastException` then the definition of `clone` in class `S` is indeed violating convention. Conversely, if `S` implements the `clone` convention correctly (i.e., indirectly calls `super.clone()`) no exception is thrown. This test code is executed the first time an object of class `S` is instantiated. In this way, if class `S` does not get used at all in the current test suite, no error is reported.

The above residual investigation provides a very strong indication of a problem that will appear in an actual execution of the program, without needing to observe the problem itself. Indeed, the current version of the program may not even have any subclasses of `S`, but a serious error is lurking for future extensions.

Implementation. Our implementation of this analysis uses AspectJ to introduce the extra class and code. In the case of complex constructors, we retrieve those with Java reflection and use AspectJ's constructor joinpoints instead of generating customized calls. A subtle point is that if the superclass, `S`, only has private constructors, the residual investigation does not apply. This is appropriate, since the absence of any externally visible constructor suggests this class is not to be subclassed. Similarly, the generated code needs to be in the same package as the original class `S`, in order to be able to access package-protected constructors.

2.2.3 Dropped Exception (DE)

Java has *checked exceptions*: any exception that may be thrown by a method needs to either be caught or declared to be thrown in the method's signature, so that the same obligation is transferred to method callers. To circumvent this static check, programmers may catch an exception and "drop it on the floor", i.e., leave empty the `catch` part in a `try-catch` block. FindBugs statically detects dropped exceptions and reports them.

Possible programmer objections to static warnings. Detecting all dropped exceptions may be a good practice, but is also likely to frustrate the programmer or to be considered a low-priority error report. After all, the type system has already performed a check for exceptions and the programmer has explicitly disabled that check by dropping the exception. The programmer may be legitimately certain that the exception will never be thrown in the given setting (a common case—especially for I/O classes—is that of a general method that may indeed throw an exception being overridden by an implementation that never does).

Dynamic clues that reinforce static warnings. Our residual investigation consists of examining which methods end up being dynamically invoked in the suspect code block and watching whether the same methods ever throw the dropped exception when called from *anywhere* in the program. For instance, the following code snippet shows a method `meth1` whose `catch` block is empty. In the `try` block of `meth1`, first `foo1` is executed, then `foo2` (possibly called from `foo1`), then `foo3`, and so on:

```
void meth1() {
  try {
    foo1();
    //Call-graph foo1()->foo2()->...->fooN()
  } catch(XException e) { }  //empty
}
```

The residual investigation will report an error if there is any other method, `methX`, calling some `foo`i where `foo`i throws an `XException` during that invocation (regardless of whether that exception is handled or not):

```
void methX {
  try { ...
    //Call-graph ...->fooN()->...
  } catch(XException e) {
    ...  // handled
  }
}
```

In this case the user should be made aware of the possible threat. If `foo`$_i$ can indeed throw an exception, it is likely to throw it in any calling context. By locating the offending instance, we prove to programmers that the exception can occur. Although the warning may still be invalid, this is a much less likely case than in the purely static analysis.

Implementation. The implementation of this residual investigation uses both the ASM library for bytecode transformation and AspectJ, for ease of manipulation.

We execute the program's test suite twice. During the first pass, we instrument the beginning and end of each empty `try-catch` block with ASM, then apply an AspectJ aspect to find all methods executed in the dynamic scope of the `try-catch` block (i.e., transitively called in the block) that may throw the exception being caught.[3] (We also check that there is no intermediate method that handles the exception, by analyzing the signatures of parent methods on the call stack.) In the first pass we collect all such methods and generate custom AspectJ aspects for the second pass. During the second pass, we then track the execution of all methods we identified in the first pass and identify thrown exceptions of the right type. For any such exception we issue an RFBI error report.

2.2.4 *Equals Method Overrides Equals in Superclass and May Not Be Symmetric* *(EQ_OVERRIDING_EQUALS_NOT_SYMMETRIC)*

Part of the conventions of comparing for value equality (via the `equals` method) in Java is that the method has to be symmetric: the truth value of `o1.equals(o2)` has to be the same as that of `o2.equals(o1)` for every `o1` and `o2`. FindBugs has a bug pattern "equals method override equals in super class and may not be symmetric", which emits a warning if both the overriding equals method in the subclass and the overridden equals method in the superclass use `instanceof` in the determination of whether two objects are equal. The rationale is that it is common for programmers to begin equality checks with a check of type equality for the argument and the receiver object. If, however, both the overridden and the overriding `equals` methods use this format the result will likely be asymmetric because, in the case of a superclass, `S`, of a class `C`, the `instanceof S` check will be true for a `C` object but not vice versa.

Possible programmer objections to static warnings. The above static check is a blunt instrument. The programmer may be well aware of the convention and might be using `instanceof` quite legitimately, instead of merely in the naive pattern that the FindBugs analysis assumes. For instance, the code of the JBoss system has some such correct `equals` methods that happen to use `instanceof` and are erroneously flagged by FindBugs.

Dynamic clues that reinforce static warnings. Our residual investigation tries to establish some confidence before it reports the potential error. We checked this pattern dynamically by calling both `equals` methods whenever we observe a comparison involving a contentious object and test if the results match (this double-calling is safe as long as there are no relevant side effects). If the two `equals` methods ever disagree (i.e., one test is true, one is false) we emit an error report.

Implementation. We implemented this residual investigation using AspectJ to intercept calls to the `equals` method and perform the dual check in addition to the original.

[3]To apply the combination of ASM and AspectJ at load time, we had to make two one-line changes to the source code of AspectJ. The first allows aspects to apply to ASM-transformed code, while the second allows AspectJ-instrumented code to be re-transformed.

2.2.5 *Equal Objects Must Have Equal Hashcodes (HE)*

As mentioned in the Introduction, FindBugs reports an error when a class overrides the `equals(Object)` method but not the `hashCode()` method, or vice versa. All Java objects support these two methods, since they are defined at the root of the Java class hierarchy, class `java.lang.Object`. Overriding only one of these methods violates the standard library conventions: an object's hash code should serve as an identity signature, hence it needs to be consistent with the notion of object value-equality.

Possible programmer objections to static warnings. This warning can easily be low-priority or irrelevant in a given context. Developers may think that objects of the suspect type are never stored in hashed data structures or otherwise have their hash code used for equality comparisons in the course of application execution. Furthermore, the warning may be cryptic for programmers who may not see how exactly this invariant affects their program or what the real problem is.

Dynamic clues that reinforce static warnings. Our Residual FindBugs Investigator installs dynamic checks for the following cases:

- `Object.hashCode()` is called on an object of a class that redefines `equals(Object)` and inherits the implementation of `hashCode()`.
- `Object.equals(Object)` is called on a class that redefines `hashCode()` and inherits the implementation of `equals(Object)`.

Meeting either of these conditions is a strong indication that the inconsistent overriding is likely to matter in actual program executions. Of course, the error may not exhibit as a fault in the current (or any other) execution.

Implementation. Our detector is implemented using the ASM Java bytecode engineering library. First, we create a blacklist containing the classes that only redefine one of `Object.equals(Object)` and `Object.hashCode()` in a coordinated manner. Then we introduce our own implementations of the missing methods in the blacklisted classes. The result is to intercept every call to either `Object.equals(Object)` or `Object.hashCode()` in instances of blacklisted classes.

2.2.6 *Non-Short-Circuit Boolean Operator (NS)*

Programmers may mistakenly use non-short-circuiting & and `|` where they intend to use short circuiting boolean operators && and `||`. This could introduce bugs if the first argument suffices to determine the value of the expression and the second argument contains side-effects (e.g., may throw exceptions for situations like a null-pointer dereference or division by zero). Therefore, FindBugs issues warnings for uses of & and `|` inside the conditions of an `if` statement.

Possible programmer objections to static warnings. Such warnings can clearly be invalid or irrelevant, e.g. if the programmer used the operators intentionally or if they don't affect program behavior. FindBugs can sometimes identify the latter case through static analysis, but such analysis must be conservative (e.g., FindBugs considers any method call on the right hand side of an && or `||` to be side-effecting). Therefore the error reports are often false.

Dynamic clues that reinforce static warnings. Using a residual investigation we can check for actual side-effects on the right-hand side of a non-short-circuiting boolean operator. It is expensive to perform a full dynamic check for side-effects, therefore we check instead for several common cases. These include dynamically thrown exceptions (directly or in transitively called methods, as long as they propagate to the current method), writes to

any field of the current class, writes to local variables of the current method, and calls to well-known (library) I/O methods. Since the residual investigation can miss some side-effects, it can also miss actual bugs. Additionally, the residual investigation will often fail to generalize: there are common patterns for which it will report an error only if the error actually occurs in the current execution. For instance, in the following example an exception is thrown only when the left-hand side of the boolean expression should have short-circuited:

```
if (ref == null | ref.isEmpty()) ...
```

Still, the residual investigation generally avoids the too-conservative approach of FindBugs, while reporting dynamic behavior that would normally go unnoticed by plain testing.

Implementation. The implementation of this residual investigation is one of the most complex (and costly) in the Residual FindBugs Investigator arsenal. We rewrite boolean conditions with the ASM bytecode rewriting framework to mark a region of code (the right-hand side of the operator) for an AspectJ aspect to apply, using a "conditional check pointcut". The aspect then identifies side-effects that occur in this code region, by instrumenting field writes, installing an exception handler, and detecting method calls to I/O methods over files, network streams, the GUI, etc. Additionally, we use ASM to detect local variable writes (in the current method only) in the right-hand side of a boolean condition.

2.2.7 Read Return Should Be Checked (RR)

The `java.io.InputStream` class in the Java standard library provides two `read` methods that return the number of bytes actually read from the stream object (or an end-of-file condition). It is common for programmers to ignore this return value. FindBugs reports an error in this case. At first glance this check looks to be foolproof: the code should always check the stream status and received number of bytes against the requested number of bytes. If the return value from `read` is ignored, we may read uninitialized/stale elements of the result array or end up at an unexpected position in the input stream.

Possible programmer objections to static warnings. Perhaps surprisingly, this FindBugs check is the source of many false positives. Although the original `java.io.InputStream` class can indeed read fewer bytes than requested, the class is not `final` and can be extended. Its subclasses have to maintain the same method signature (i.e., return a number) when overriding either of the two `read` methods, yet may guarantee to always return as many bytes as requested (Notably, the Eclipse system defines such a subclass and suffers from several spurious FindBugs error reports.)

Dynamic clues that reinforce static warnings. Our residual investigation first examines whether the read method is called on a subclass of `java.io.InputStream` that overrides the `read` method. If so, we wait until we see a `read` method on an object of the suspect subclass return fewer bytes than requested (even for a call that *does* check the return value). Only then we report *all* use sites that do not check the return value of `read`, as long as they are reached in the current execution and the receiver object of `read` has the suspect dynamic type.

Implementation. The implementation of this analysis involves two computations, performed in the same execution. For the first computation, we instrument all `read` methods that override the one in `InputStream` (using AspectJ) to observe which ones return fewer bytes than requested. We collapse this information by dynamic object type, resulting in a list of all types implementing a `read` method that may return fewer bytes than requested; we always include `java.io.InputStream` in that list. For the second computation, we instrument all suspected `read` call sites with ASM, to determine the dynamic type of the receiver object. These dynamic types are the output of the second computation. At the end of the test suite execution, we cross-check the output of both passes. We then report any use of `read` without a result check on an object with a dynamic type for which we know that `read` may return fewer bytes than the maximum requested.

2.3 Discussion

The main purpose of a residual investigation is to provide increased soundness for bug reports: an error report should be likely valid and important. In this sense, a residual investigation is not in competition with its underlying static analysis, but instead complements it. In the case of RFBI, the point is not to obscure the output of FindBugs: since the static analysis is performed anyway, its results are available to the user for inspection. Instead, RFBI serves as a classifier and reinforcer of FindBugs reports: the RFBI error reports are classified as higher-certainty than the average FindBugs report.

This confidence filtering applies both positively and negatively. RFBI may confirm some FindBugs error reports, fail to confirm many because of lack of pertinent dynamic observations, but also fail to confirm some *despite* numerous pertinent dynamic observations. To see this, consider an analysis such as "dropped exception" (DE). RFBI will issue no error report if it never observes an exception thrown by a method dynamically called from a suspicious `try-catch` block. Nevertheless, it could be the case that the program's test suite never results in exceptions or (worse) that there are exceptions yet the suspicious `try-catch` block was never exercised, and hence the methods under its dynamic scope are unknown. In this case, RFBI has failed to confirm an error report due to lack of observations and not due to the observations not supporting the error. This difference is important for the interpretation of results. It is an interesting future work question how to report effectively to the user the two different kinds of negative outcomes (i.e., unexercised code vs. exercised code yet failure to confirm the static warning). In our experimental evaluation, we do this via a metric of the dynamic opportunities the residual investigation had to confirm an error report, as discussed in the next section.

3. EVALUATION

3.1 Subject applications

We evaluated RFBI on several large open-source systems: JBoss (v.6.0.0.Final), BCEL (v.5.2), NetBeans (v.6.9), Tomcat (7.0), JRuby (v.1.5.6), Apache Commons Collections (v.3.2.1), and Groovy (v.1.7.10). The advantage of using third-party systems for evaluation is that we get a representative view of what to expect quantitatively by the use of residual investigation. The disadvantage is that these systems are large, so great effort needs to be expended to confirm or disprove bugs by manual inspection.

It is common in practice to fail to confirm a static error report because of a lack of relevant dynamic observations. This is hardly a surprise since our dynamic observations are dependent on the examined program's test suite. For all systems, we used the test suite supplied by the system's creators, which in some cases was sparse (as will also be seen in our test running times).

3.2 Dynamic potential

In order to make the important distinction between "relevant code not exercised" and "relevant code exercised, yet no confirmation of the bug found", we use a *dynamic potential* metric. Gen-

erally, for each static error report, we automatically pick a related method (chosen according to the kind of bug reported) and measure how many times this method gets executed by the test suite. We found this metric to be invaluable, despite occasional arbitrariness in what we pick to measure. Generally, when multiple conditions need to be satisfied for the dynamic error to occur, we choose one of them arbitrarily. For instance, for the "equal objects must have equal hashcodes" analysis, we measure the number of times the overridden `equals` is called on objects of the suspect class that redefines `equals` and not `hashCode` (and vice versa). This is not directly tied to the opportunities to find the bug (which, recall, is reported if `hashCode` is ever called on such a suspect object) but it is a good indication of how much this part of the code is exercised.

Table 1: Summary of results: reports by FindBugs (S) vs. RFBI (R), the Dynamic Potential (DP) metric of how many of the static error reports had related methods that were exercised dynamically, and the number of original test cases (T) that reported an error.

Bug Pattern	S	R	DP	T
Bad Covariant Definition of Equals	5	0	0	0
Cloneable Not Implemented Correctly	41	4	5	0
Dropped Exception	128	0	7	0
Equals Method May Not Be Symmetric	8	1	1	0
Equal Objects Must Have Eq. Hashcode	211	25	28	0
Non-Short-Circuit Boolean Operator	18	0	1	0
Read Return Should Be Checked	25	1	1	0
Total	436	31	43	0

3.3 Volume of reports

Table 1 shows the number of static error reports (FindBugs), reports produced by residual investigation (RFBI), and dynamic potential metric. The difference between FindBugs and RFBI reports is roughly an order of magnitude: a total of 436 potential bugs are reported by FindBugs for our test subjects and, of these, RFBI produces reports for 31. Thus, it is certainly the case that residual investigation significantly narrows down the area of focus compared to static analysis. Similarly, none of the test cases in our subjects' test suites failed. Therefore, the 31 reports by RFBI do generalize observations significantly compared to mere testing. Of course, these numbers alone mean nothing about the *quality* of the reports—we examine this topic later.

By examining the dynamic potential metric in Table 1, we see that much of the difference between the numbers of FindBugs and RFBI reports is due simply to the suspicious conditions not being exercised by the test suite. Most of the static bug reports are on types or methods that do not register in the dynamic metrics. This can be viewed as an indication of "arbitrariness": the dynamic analysis can only cover a small part of the static warnings, because of the shortcomings of the test suite. A different view, however, is to interpret this number as an indication of why static analysis suffers from the "slew of false positives" perception mentioned in the Introduction. Programmers are likely to consider static warnings to be irrelevant if the warnings do not concern code that is even touched by the program's test suite.

3.4 Quality of research reports (summary)

RFBI narrows the programmer's focus compared to FindBugs but the question is whether the quality of the RFBI reports is higher than that of FindBugs reports, and whether RFBI succeeds as a classifier (i.e., whether it classifies well the dynamically exercised reports into bugs and non-bugs).

Since our test subjects are large, third-party systems, we cannot manually inspect all (436) FindBugs reports and see which of them are true bugs. Instead, we inspected the 43 reports that are dynamically exercised (per the DP metric) as well as a sample of 10 other FindBugs reports that were never dynamically exercised (and, thus, never classified by RFBI as either bugs or non-bugs). The latter were chosen completely at random (blind, uniform random choice among the reports).

If we view RFBI as a classifier of the 43 dynamically exercised FindBugs reports, its classification quality is high. As seen in Table 1, RFBI classifies 31 of the 43 dynamically exercised reports as bugs (i.e., reinforces them), thus rejecting 12 reports. Table 2 shows which of these RFBI classifications correspond to true bugs vs. non-bugs. The number for the correct outcome for each row is shown in boldface.

Table 2: Quality of RFBI warnings on the 43 dynamically exercised FindBugs reports.

Dynamic reports	bug	non-bug	undetermined
31 reinforced	**24**	6	1
12 rejected	0	**11**	1
43 total	24	17	2

From Table 2, we have that the precision of RFBI is $\geq$ 77% (or that RFBI produces $<$ 23% false warnings) and that its recall is $\geq$ 96%, over the 43 dynamically exercised FindBugs reports.

For comparison, among the 10 non-exercised FindBugs reports that we sampled at random, only one is a true bug. Thus, the precision of FindBugs on this sample was 10%, which is a false warning rate of 90%. We see, therefore, that RFBI reports are of higher quality than FindBugs reports and the programmer should prioritize their inspection.

3.5 Detailed discussion of reports

We next discuss in detail the RFBI results that we inspected manually. This yields concrete examples of bug reports reinforced and rejected (both correctly and falsely) for the numbers seen above. Table 3 breaks down the reports dynamically exercised by test subject and analysis, as well as their dynamic potential. Note that in the rest of this section we are not concerned with FindBugs reports that are not exercised dynamically.

- RFBI correctly confirms four dynamically exercised instances of "Cloneable Not Implemented Correctly" and rejects one. This is a sharp distinction, and, we believe, correct. In three of the four instances (in Apache Commons) `clone` directly constructs a new object, rather than calling the parent `clone`. One bug in Groovy arises in an instance where a delegator violates the cloning protocol by returning a clone of its delegate instead of a clone of itself. The rejected bug report is a `clone` method for a singleton object that returned `this`, which is entirely typesafe.

- RFBI rejects seven "Dropped Exception" reports, of which our manual inspection found six to be accurate and one unclear The unclear case involved the NetBeans test harness ignoring an exception caused by backing store problems during synchronisation; we expect that such an exception is likely to trigger further bugs and hence unit test failures but argue that it might have been appropriate to log the exception

Table 3: Breakdown of all RFBI warnings as well as the dynamic potential metric for the warning. "a/b/c" means there were a RFBI warnings of this kind, b dynamic potential methods executed (zero typically means there was no opportunity for the residual investigation to observe an error of the statically predicted kind), and each of them was observed to execute c times (on average). Thus, the sum of all as is the number of RFBI reinforced reports (31) and the sum of all bs is the number of total dynamically exercised FindBugs reports (43). Empty cells mean that there were no static error reports for this test subject and analysis—this is in contrast to 0/0/0 cells, which correspond to static warnings that were never exercised dynamically.

Bug Pattern	#confirmed bugs/dynamic potential(#executed methods)/avg. times executed						
	JBoss	BCEL	NetBeans	Tomcat	JRuby	ApacheCC	Groovy
Bad Covariant Definition of Equals			0/0/0	0/0/0			
Cloneable Not Implemented Correctly			0/0/0	0/1/9		3/3/657.7	1/1/11
Dropped Exception	0/4/378		0/1/79	0/0/0	0/1/5		0/1/25
Equals Method May Not Be Symmetric	0/0/0		0/0/0		1/1/2.6M		
Equal Objects Must Have Eq. Hashcodes	1/2/1	20/20/77k	0/0/0	1/1/5k	2/2/3.5		1/3/14
Non-Short-Circuit Boolean Operator			0/0/0		0/1/194		
Read Return Should Be Checked			0/0/0	0/0/0	0/0/0		1/1/571

rather than ignoring it. Of the remaining six cases, four affected JBoss. In three of these cases the code correctly handles the erroneous case by exploiting the exceptional control flow in other ways (e.g., when an assignment throws an exception, the left-hand side retains its previous value, which the code can test for) or by falling back on alternative functionality (for example, JBoss attempts to use I/O access to `/dev/urandom` to generate random numbers, but falls back on the Java random number generator if that approach fails). The fourth JBoss case ignores exceptions that may arise while shutting down network connections. We assume that the programmers' rationale is that they can do no more but trust that the library code tries as hard as possible to release any resources that it has acquired, and that afterwards the program should run in as robust a fashion as possible.

In one of the two remaining cases (JRuby), exceptions are dropped in debug code and can only arise if the JRuby VM has been corrupted or has run out of memory. In the final case (Groovy), the dropped exception is a ClassLoaderException that could only arise if the Java standard library were missing or corrupt.

- We observed one out of eight "Equals Method May Not Be Symmetric" instances dynamically, in JRuby's `RubyString` class. RFBI here indicated that the report was correct, pointing to an implementation of `equals` that differs subtly from the equivalent Ruby equality check for the same class. In practice, the 'Java' version of equality is only used in rare circumstances and unlikely to cause problems, unless the integration between Java and Ruby were to be changed significantly. We thus found it unclear whether this bug report was a true positive (as suggested by RFBI) or not.

- RFBI confirms 20 "Equal Objects Must Have Equal Hashcodes" reports for BCEL. The 20 RFBI reports concern classes that represent branch instructions in the Java bytecode language. All reported classes define an application-specific value equality `equals` method, without ever defining `hashCode`. Objects for branch instructions, however, get entered in a `HashSet`, as part of a seemingly oft-used call: `InstructionHandle.addTargeter`. Therefore, we believe that the bug warning is accurate for these instructions and can result in obscure runtime faults.

  RFBI incorrectly confirms two "Equal Objects Must Have Equal Hashcodes" bug reports in JRuby: in both cases, `equals` is overridden but `hashCode` is not. In one of the pertinent classes, the superclass `hashCode` implementation uses a custom virtual method table for Ruby to look up a correct subclass-specific `hashCode` implementation; such complex indirection is impossible to detect in general. In the other class, `hashCode` is only used to generate a mostly-unique identifier for debugging purposes, instead of hashing. This breaks the heuristic assumption that `hashCode` and `equals` collaborate. In Groovy, RFBI incorrectly confirms a bug for exactly the same reason. Meanwhile, the two bugs we rejected in Groovy again did not see invocations of the missing methods, and we consider our results to be accurate in those cases. RBFI also incorrectly confirms a bug for JBoss (and rejects one, correctly): although the `equals` method is overridden, it does nothing more than delegate to the superclass method, which also defines an appropriate `hashCode`.

- RFBI correctly rejects a "Non-Short Circuit Boolean Operator" bug report in JRuby involving a method call, as the method in question is only a getter method (and thus has no side effects that might unexpectedly alter program behavior).

- Only one report of "Read Return Should Be Checked" is exercised in unit tests. This report involves Groovy's Json lexer, which in one instance does not check the number of bytes returned by a read operation. However, the bytes read are written into an empty character array that is immediately converted to a string, which is then checked against an expected result: if the number of bytes read was less than requested, this later check must fail, because the generated string will be too short. Such complex logic is beyond the scope of RFBI, which erroneously confirms the static report.

In summary, the few misjudged bug reports arose because the code violated the assumptions behind our concrete residual investigation heuristics (e.g., application-specific use of `hashCode`). Incorrect RFBI bug reports typically were due to complex mechanisms that achieve the desired result in a way that requires higher-level understanding yet proves to be semantically correct (e.g., leaving out a test for bytes read because it is subsumed by a string length check). It is unlikely that any automatic technique can eliminate bug reports that are erroneous because of such factors.

3.6 Runtime overhead

Table 4 shows the runtime overhead of our residual investigation. As can be seen, compared to the baseline (of uninstrumented code)

residual investigation slows down the execution of the test suite typically by a factor of 2-to-3, but even going up to 6. The "dropped exception" analysis is the worst offender due to executing the test suite twice and watching a large number of the executed method calls.

3.7 Threats to validity

Our experimental evaluation of the efficacy of residual investigation shows that it yields higher-precision bug reporting and a reliable classification of bugs. The main threats to validity include the following threats to external validity.

- Choice of subject applications: We did not select our seven subject applications truly randomly from the space of all possible Java applications or even from all current Java applications. I.e., our empirical results may not generalize well to other applications. However, our applications cover a variety of application areas. I.e., we use a data structure library, two language runtime systems, a bytecode engineering library, two web servers, and an IDE. Given the large size of these subject applications, we suspect that our findings will generalize to a large extent, but this remains to be confirmed as part of a larger empirical study.

- Choice of FindBugs patterns: We did not select the patterns randomly from the list of all FindBugs patterns. I.e., residual investigation likely does not generalize to all FindBugs patterns. Our choice of patterns was influenced by subjective considerations such as how well-known we deemed a pattern to be. I.e., six of our patterns have been described in an article [15] by the FindBugs authors. That article describes a total of 18 patterns. For our evaluation we picked patterns for which we suspected that FindBugs would produce false warnings on our subject applications. We argue that this is not a strong threat to validity, since we easily obtained strong results for a third of the sample presented by the earlier FindBugs article.

  If we step back and review all current FindBugs bug patterns, we can easily identify several of them that are simple enough to allow for a fully precise static detector, and such a fully precise static detector will not benefit from residual investigation. However, many other bug patterns are too complex to allow for a precise static detector. For example, Hovemeyer and Pugh tested twelve out of the 18 patterns they described (including four of ours) for false positives in two applications. They found that ten of the twelve patterns (including ours) produced false positives with the pattern implementations they had available in 2004. We suspect that the 18 patterns that they described are at least somewhat representative of all FindBugs patterns, which would mean that many other current FindBugs detectors may similarly benefit from a residual investigation.

- Choice of static analysis system: We did not select FindBugs, our static analysis system, truly randomly. We picked FindBugs because it is arguably the most widely known and used such tool for Java. We suspect that our findings will generalize to other static analysis tools and approaches.

4. RELATED WORK

Static and dynamic analyses are routinely chained together for checking program correctness conditions in programming languages, i.e., in compilers and runtime systems. Compilers check certain properties statically and insert runtime checks for remaining properties. A classic example is checking that an array read does not read a memory location outside the bounds of the array. To enforce this property, Java compilers traditionally insert a dynamic check into the code before each array read. To reduce the runtime overhead, static analyses such as ABCD [5] have been developed that can guarantee some reads as being within bounds, just the remaining ones have then to be checked dynamically. Beyond array bounds checking, a similar static dynamic analysis pipeline has been applied to more complex properties. The Spec# extended compiler framework [2] is the prime example, it can prove some pre- and post-conditions statically and generates runtime checks for the remaining ones. Gopinathan and Rajamani [13] use a combination of static and dynamic analysis for enforcing object protocols. Their approach separates the static checking of protocol correctness from a dynamic check of program conformance to the protocol.

In residual dynamic typestate analysis, explored by Dwyer and Purandare [11] and Bodden et al. [3, 4], a dynamic typestate analysis that monitors all program transitions for bugs is reduced to a residual analysis that just monitors those program transitions that are left undecided by a previous static analysis. This approach exploits the fact that a static typestate analysis is typically complete, i.e., it over-approximates the states a program can be in. If for a small sub-region of the program the over-approximated state sets do not contain an error state, all transitions within such a region can therefore safely be summarized and ignored by the subsequent residual dynamic analysis. At a high level, our approach adopts this idea, by only monitoring those aspects of the program that the static analysis has flagged as suspicious. However, our approach is more general in two dimensions, (1) typestate analysis is restricted to verifying finite state machine properties ("do not pop before push"), while our approach can be applied to more complex properties ("do not pop more than pushed") and (2) our dynamic analysis is predictive: It leverages dynamic results to identify bugs in code both executed and not executed during the analysis.

Beyond typestates, the idea of speeding up a dynamic program analysis by pre-computing some parts statically has been applied to other analyses, such as information flow analysis. For example, recent work by Chugh et al. [8] provides a fast dynamic information flow analysis of JavaScript programs. JavaScript programs are highly dynamic and can load additional code elements during execution. These dynamically loaded program elements can only be checked dynamically. Their staged analysis statically propagates its results throughout the statically known code areas, up to the borders at which code can change at runtime. These intermediate results are then packaged into residual checkers that can be evaluated efficiently at runtime, minimizing the runtime checking overhead.

Our analysis can be seen in a similar light as residual dynamic typestate analysis and residual information flow analysis. If we take as a hypothetical baseline somebody running only our residual checkers, then adding the static bug finding analysis as a pre-step would indeed make the residual dynamic analysis more efficient, as the static analysis focuses the residual analysis on code that may have bugs. However, our goals are very different. Our real baseline is an established static analysis technique whose main problem is over-approximation, which leads to users ignoring true warnings.

Our earlier work on Check 'n' Crash [9] and DSD-Crasher [10, 24] can be seen as strict versions of residual analysis. These earlier techniques share our goal of convincing users of the validity of static bug warnings. However, Check 'n' Crash and DSD-Crasher guarantee that a given warning is true, by generating and executing concrete test cases that satisfy the static warning, until a static warning can be replicated in a concrete execution or a user-defined

Table 4: Running times for all residual investigations. The baseline (bottom line of the table) is the non-instrumented test suite running time. Empty cells mean that there were no static error reports for this test subject and analysis.

Bug Pattern	Execution time with and without instrumentation [min:s]						
	JBoss	BCEL	NetBeans	Tomcat	JRuby	ApacheCC	Groovy
Bad Covariant Definition of Equals			13:07	3:04			
Cloneable Not Implemented Correctly			7:17	5:20		5:25	39:15
Dropped Exception	655:01		16:00	16:05	17:08		41:44
Equals Method May Not Be Symmetric	531:42		8:23		11:03		
Equal Objects Must Have Eq. Hashcodes	363:48	1:20	13:07	3:03	3:44		7:25
Non-Short-Circuit Boolean Operator			9:36		11:13		
Read Return Should Be Checked			16:49	10:19	12:30		42:25
No Instrumentation	178:07	:23	6:42	3:03	3:28	2:05	7:13

limit is reached. While such proof is very convincing, it also narrows the techniques' scope. I.e., our earlier tools could only confirm very few warnings. In our current residual analysis, we relax this strict interpretation and also consider clues that are just very likely to confirm a static warning.

Dynamic symbolic execution is a recent combination of static and dynamic program analysis [6, 12, 17, 25]. Dynamic symbolic execution is also a strict approach that warns a user only after it has generated and executed a test case that proves the existence of a bug. Compared to our analysis, dynamic symbolic execution is heavier-weight, by building and maintaining during program execution a fully symbolic representation of the program state. While such detailed symbolic information can be useful for many kinds of program analyses, our current residual investigations do not need such symbolic information, making our approach more scalable.

Monitoring-oriented programming (MOP), by Chen and Roşu, shows how runtime monitoring of correctness conditions can be implemented more efficiently, even without a prefixed static analysis [7]. JavaMOP, for example, compiles correctness conditions to Java aspects that add little runtime overhead. This technique is orthogonal to ours. I.e., as some of our dynamic analyses are currently implemented manually using AspectJ, expressing them in terms of JavaMOP would be a straightforward way to reduce our runtime overhead.

Our analysis can be viewed as a ranking system on static analysis error reports. There has been significant work in this direction. Kremenek et al. [20] sort error reports by their probabilities. A model is used for computing probabilities for each error report by leveraging code locality, code versioning, and user feedback. The effectiveness of the model depends on (1) a fair number of reports and (2) strong clustering of false positives. Kim and Ernst [19] prioritize warning categories by mining change log history messages. Intuitively, they expect a warning category to be important if warning instances from the category are removed many times in the revision history of a system. Their method requires change logs with good quality. For static tools such as FindBugs, which analyze Java bytecode to generate warnings, they also require compilation of each revision.

5. CONCLUSIONS

We presented residual investigation: the idea of accompanying a static error analysis with an appropriately designed dynamic analysis that will report with high confidence whether the static error report is valid. We believe that residual investigation is, first and foremostly, an interesting *concept*. Identifying this concept helped us design dynamic analyses for a variety of static bug patterns and implement them in a tool, RFBI. We applied RFBI to a variety of test subjects to showcase the potential of the approach.

There are several avenues for future work along the lines of residual investigation. First and foremostly, our current work has been a proof of concept. That is, we have been interested in examining whether *some* application of residual investigation can be fruitful and not in determining how universally applicable the concept is to the realm of static analyses (i.e., how likely is it that a specific static analysis for bug detection can be accompanied by a profitable residual investigation).

Overall, we feel that for bug finding tools to get to the next level of practicality they need to incorporate some flavor of the dynamic validation that residual investigation offers.

6. ACKNOWLEDGMENTS

We thank Bill Pugh for helpful comments and interesting discussions. This material is based upon work supported by the National Science Foundation under Grants No. 0917774, 0934631, 1115448, 1117369, and 1017305.

7. REFERENCES

[1] N. Ayewah and W. Pugh. The Google FindBugs fixit. In *Proc. 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 241–252. ACM, 2010.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 49–69. Springer, Mar. 2004.

[3] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 5–14. ACM, May 2010.

[4] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proc. 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 525–549, July 2007.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 321–333. ACM, June 2000.

[6] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th International SPIN Workshop on Model Checking Software*, pages 2–23. Springer, Aug. 2005.

[7] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. In *Proc. 22nd ACM SIGPLAN*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 569–588. ACM, Oct. 2007.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–62. ACM, June 2009.

[9] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering (ICSE)*, pages 422–431. ACM, May 2005.

[10] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254. ACM, July 2006.

[11] M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 124–133. ACM, Nov. 2007.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[13] M. Gopinathan and S. K. Rajamani. Enforcing object protocols by combining static and runtime analysis. In *Proc. 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260. ACM, Oct. 2008.

[14] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 132–136. ACM, Oct. 2004.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, Dec. 2004.

[16] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 9–14. ACM, June 2007.

[17] M. Islam and C. Csallner. Dsc+Mock: A test case + mock class generator in support of coding against interfaces. In *Proc. 8th International Workshop on Dynamic Analysis (WODA)*, pages 26–31. ACM, July 2010.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. 15th European Conference on Object Oriented Programming (ECOOP)*, pages 327–353. Springer, June 2001.

[19] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proc. 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 45–54. ACM, Sept. 2007.

[20] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 83–93. ACM, Oct. 2004.

[21] M. Musuvathi and D. Engler. Some lessons from using static analysis and software model checking for bug finding. In *Proc. Workshop on Software Model Checking (SoftMC)*. Elsevier, July 2003.

[22] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proc. 15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256. IEEE, Nov. 2004.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proc. 16th Symposium on Operating Systems Principles (SOSP)*, pages 27–37. ACM, Oct. 1997.

[24] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *Proc. International Conference on Tests And Proofs (TAP)*, pages 1–16. Springer, Feb. 2007.

[25] N. Tillmann and J. de Halleux. Pex - white box test generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*, pages 134–153. Springer, Apr. 2008.

[26] A. Tomb, G. P. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107. ACM, July 2007.

[27] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom)*, pages 40–55. Springer, May 2005.

[28] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 97–106. ACM, Oct. 2004.