Residual Investigation: Predictive and Precise Bug Detection

KAITUO LI, University of Massachusetts, Amherst, USA
CHRISTOPH REICHENBACH, Goethe University Frankfurt, Germany
CHRISTOPH CSALLNER, University of Texas at Arlington, USA
YANNIS SMARAGDAKIS, University of Athens, Greece

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 analyzer 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.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging – debugging aids; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms: Design, Reliability, Verification

Additional Key Words and Phrases: False warnings, existing test cases, RFBI

ACM Reference Format:
Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis, 2013. Residual Investigation: Predictive and Precise Bug Detection. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 29 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

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 [Musuvathi and Engler 2003; Zitser et al. 2004; Wagner et al. 2005; Rutar et al. 2004; Ayewah and Pugh 2010]. 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

Authors' addresses: K. Li, Computer Science Department, University of Massachusetts, Amherst, MA 01003, USA; C. Reichenbach, Department of Informatics and Mathematics, Goethe-University Frankfurt, 60054 Frankfurt am Main, Germany; C. Csallner, Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA; Y. Smaragdakis, Department of Informatics, University of Athens, Athens 15784, Greece.
This is a revised and extended version of [Li et al. 2012] which was presented at ISSTA 2012 and received an ACM SIGSOFT Distinguished Paper Award.

DOI:http://dx.doi.org/10.1145/0000000.0000000

cost of investigation) to be issued only when there is high confidence that the error is real, 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., [Csallner and Smaragdakis 2005; 2006; Godefroid et al. 2005; Cadar and Engler 2005; Tomb et al. 2007; Tillmann and de Halleux 2008; Islam and Csallner 2014]) in order to achieve higher certainty than purely static approaches.

In this article we identify and 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. Its purpose is to determine 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 reinforce the statically detected error. That is, a residual investigation is a *predictive* dynamic analysis, predicting errors in executions not actually observed. The predictive nature of residual investigation is a significant advantage in practice: high-covering test inputs are hard to produce for complex programs.

Consider as an example the "equal objects must have equal hashcodes" analysis (codenamed HE) in the FindBugs static error detector for Java [Hovemeyer and Pugh 2004a; 2007; Ayewah and Pugh 2010]. 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. 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 a predictive dynamic analysis: it is both more general than mere testing and 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 [Kiczales et al. 2001]. 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 are 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 [Savage et al. 1997]: its analysis predicts races based on inconsistent locking, even when no races have appeared in the observed execution.) In contrast, we defined several predictive analyses in a brief period of time, by merely examining the FindBugs list of bug patterns under the lens of residual investigation.

The present article revises, generalizes, and extends the original conference publication on residual investigation [Li et al. 2012]. Thus, we integrate the original definition and presentation of residual investigation analyses in RFBI, but our presentation also reflects a broader perspective on how residual investigation can apply to different contexts, the enabling factors and limitations of the approach, and more.

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.
— We validate our expectation that the resulting analyses are useful by applying them to 9 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.
— We discuss the applicability of residual investigation to several other static analyses in the literature. These include race detection, SQL injection analyses, and other pattern-based code analyses in FindBugs. We also discuss how the quality of the test suite affects the applicability of residual investigation.

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. Background and General Concept

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 Section 1:

— 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. This is a key requirement, since it is easier to have a test case to expose suspicious behaviors than to have a test case to actually cause faults.

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

— $B_p(b)$, for "p has bug b", i.e., the program text contains an error, b, 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.
— $S_p(b)$, for "p induces a static error report on bug b", i.e., the program text contains a possible error that the static analysis reports.

[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 article 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.

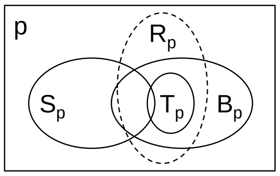

Fig. 1. The goal of residual investigation (R_p) of a program p is to provide a filter for the static bug warnings (S_p), such that R_p and S_p combined (i.e., the intersection of R_p and S_p) better approximates the program's set of true bugs (B_p) than static analysis alone.

— $T_p(b)$, for "p causes a test case fault due to bug b", when executed with p's test suite.
— $R_p(b)$, for "p triggers the residual investigation (dynamic) check" (associated with the static report for bug b), when executed with p's test suite.

Although we typically use the term "residual investigation" for the dynamic analysis, the error reporting process includes the static analysis. That is, a residual investigation issues a reinforced warning in case the static analysis predicted an error and the dynamic analysis confirms it, i.e., in case $S_p(b) \wedge R_p(b)$.

We assume that the dynamic testing is sound for the execution it examines.[2] Thus, we have:

$$\forall p, b : T_p(b) \Rightarrow B_p(b)$$

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

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

$$\exists p, b : S_p(b) \wedge \neg B_p(b)$$

(As discussed in the introduction, this assumption is true for most static analysis systems, as they produce false warnings.)

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

$$\exists p, b : B_p(b) \wedge \neg T_p(b)$$

(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):

$$\forall p, b : B_p(b) \text{ approximately iff } S_p(b) \wedge R_p(b)$$

This is the only informal notion in the above. It is practically impossible to have exact equivalence for realistic programs and error conditions, since $R_p(b)$ examines a finite number of program executions. Note that (approximate) equivalence means both that $S_p(b) \wedge R_p(b)$ (likely) implies a bug b exists and that, if there is a bug b, $S_p(b) \wedge R_p(b)$ will (likely) be true. 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* (produces false positives) or *irrelevant* (produces true but low-value positives).

[2]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.

— 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 overloads 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) {
  for (Object obj : this) {
    if (obj.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, which always 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`, is declared `final` or 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 overrides 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 the `instanceof C` check will be false for an `S` object.

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.

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:

[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.

— `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 trigger 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 the non-short-circuiting binary operators `&` or `|` where they intend to use the short-circuiting boolean operators `&&` or `||`. 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 do not 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 a plain FindBugs report.

This confidence filtering applies both positively and negatively. RFBI may confirm some FindBugs error reports, fail to confirm many because of a 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 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). Our experimental evaluation describes this distinction via a metric of the number of dynamic opportunities to confirm an error report that the residual investigation had, as we discuss in the next section.

2.4. Implementation Complexity

Residual investigation requires effort in capturing dynamic evidence related to static bug reports, by using instrumentation infrastructure such as ASM and AspectJ. The degree of effort varies with the complexity of runtime evidence that we are trying to collect. In simple cases, residual investigations are easy to develop. For example, for the bug pattern "Equal Objects Must Have Equal Hashcodes", we only have to insert a method for each class in a blacklist as the class is being loaded. But in complex cases that require interaction of different instrumentation tools or a variety of dynamic information, residual investigation needs non-trivial development effort. A typical example is the implementation of "Non-Short-Circuit Boolean Operator", which needs to make AspectJ communicate with ASM on the code blocks for which field writes, exception handlers, network or I/O operations, etc. need to be instrumented.

In total, our tool contains 3292 lines of Java code, including 298 lines for utility classes, such as logging facilities and Java agents, and 2994 lines for 7 bug patterns altogether. Note that this does not include AspectJ code generated dynamically. Figure 2 quantifies the implementation complexity via source lines of code[4] for each bug pattern. The figure also lists the set of tools used to develop the residual investigation.

Bug Pattern	Implementation Tool	Lines of Code
Bad Covariant Definition of Equals	ASM	40
Cloneable Not Implemented Correctly	AspectJ	512
Dropped Exception	ASM, AspectJ	769
Equals Method May Not Be Symmetric	AspectJ	465
Equal Objects Must Have Eq. Hashcode	ASM	40
Non-Short-Circuit Boolean Operator	ASM, AspectJ	811
Read Return Should Be Checked	ASM, AspectJ	357

Fig. 2. Size of implementation and implementation tools in residual investigations.

3. EVALUATION

We evaluate RFBI in two complementary ways. First, we do a targeted evaluation using two medium-size systems with which we were already familiar, and, hence, can directly assess the quality of bug reports. Second, we apply RFBI on several well-known large open-source systems. For such systems, judging the quality of a bug report is harder and can only be done via sampling and best-effort estimates.

In order to make the important distinction between "relevant code not exercised" (undetermined bugs) and "relevant code exercised, yet no confirmation of the bug found" (rejected bugs), we use a *dynamic potential* metric. Generally, for each static error report, we automatically pick a method whose execution is part of our analysis (chosen according to the kind of bug reported) and measure how many times this method gets executed by the test suite. The resultant count gives us an upper bound on the number of reports we can expect from residual investigation. We found this metric to be invaluable. Generally, when multiple conditions need to be satisfied for a residual investigation, 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, as we described earlier, 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.

[4]The counting is done using SLOCCount—http://www.dwheeler.com/sloccount/

3.1. Targeted Evaluation

The targeted evaluation consists of applying RFBI to programs whose behavior we know well, so that we can determine whether RFBI's warnings are correct or incorrect by examining the systems' source code. We evaluated RFBI on two medium-sized systems, Daikon [Ernst et al. 2001] (Revision a1364b3888e0) and Apache Jackrabbit (Revision 1435197), using the systems' standard test suites. Since the size of Apache Jackrabbit is still large for our purpose of targeted evaluation, we used only the core component of Apache Jackrabbit (which has 21 components overall).

Analysis of reports. FindBugs reports 14 bugs for Daikon and Jackrabbit. The standard system test suites exercise 8 of these bugs, i.e., there are 8 reports for which the DP metric is non-zero. This ratio between RFBI and FindBugs reports is high, compared to our later experiments with other large third-party systems.

Of the 8 RFBI reports, 3 are reinforced and 5 are rejected. By manual inspection, we found that RFBI correctly confirms 2 of the 3 reports, incorrectly reinforces a false positive, and correctly rejects 5 reports. (This yields a precision of 66.6% and a recall of 100% over the 8 exercised FindBugs reports, but clearly the absolute numbers of reports are low for statistical accuracy.)

— Findbugs reports two "Cloneable Not Implemented Correctly" warnings for Jackrabbit. RFBI observes one dynamically exercised instance and correctly confirms the instance: rather than calling `super.clone()` in its superclass, `clone` in the instance directly constructs a new object.
— Findbugs reports four "Dropped Exception" warnings for Daikon and two for Jackrabbit. RFBI does not observe any dynamically exercised instance for Daikon, but does observe two for Jackrabbit. Of the two dynamically exercised instances, RFBI incorrectly confirms one instance, and correctly rejects another one. In the incorrectly confirmed instance, exception throwing is used as a means for handling control flow within a large method body. The method in question tries several alternatives for an action, in a fixed order. It encodes each alternative as a single block of code wrapped into its own (empty) exception handler. If an alternative succeeds, the block returns from the method to signal overall success. If the alternative fails, the block raises an exception and thereby continues to the next block. Thus, the dropped exception here needs no exception handler. The correctly rejected bug report ignores a statically reported exception that supposedly can be raised while shutting down a virtual file system. In practice, this exception can only arise if initialization failed. However, if the virtual file system failed to initialize correctly, some other exceptions would already have been thrown at creation time, and the related catch blocks would have handled or reported these exceptions. Thus, it is not necessary to handle or report the same exception again when the virtual file system is closed.
— FindBugs reports two "Equal Objects Must Have Equal Hashcodes" warnings for Daikon and three for Jackrabbit. All reported classes override their superclasses' `equals` methods without overriding `hashCode`. Instances of all reports are exercised in test suites. RFBI correctly confirms one report on Daikon and rejects another. To understand the confirmed Daikon case, note that Daikon takes an arbitrary client program, instruments it, and performs dynamic analysis. In the correctly confirmed error report, Daikon takes arbitrary objects from the client program and uses them as keys to a hash map. Since Daikon knows nothing about the client program a priori, this behavior is inherently unsafe. The rejected Daikon case involves a class whose instances are only inserted in an `ArrayList` structure, which does not employ `hashCode`.
RFBI correctly rejects the three Jackrabbit reports. In two of these cases, instances of the suspicious class are used as values of a hash map, but never as keys (i.e., they are never hashed). Instead, a uniquely-identifying field of the suspicious class is used as the key of the `HashMap` structure. The field type is not involved in the FindBugs error report, since it correctly defines both `hashCode` and `equals`. The third Jackrabbit report, similar to the rejected Daikon case, involves a class whose instances are only used in an `ArrayList` structure.
— FindBugs reports one "Read Return Should Be Checked" warning for Jackrabbit, but the relevant code is not exercised dynamically.

Bug Pattern	FindBugs	RFBI	Dynamic Potential	Test Cases
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

Fig. 3. Summary of results: reports by FindBugs vs. RFBI, the Dynamic Potential metric of how many of the static error reports had related methods that were exercised dynamically, and the number of original test cases that reported an error.

In summary, although small, our experiments with Daikon and the Apache Jackrabbit core package show the potential for residual investigation.

Runtime overhead. Because Daikon's own instrumenter and Jackrabbit's testing framework are in conflict with RFBI's dynamic instrumenter, we pre-instrumented the code bases at build time. When we run pre-instrumented code, the overhead is mostly due to logging dynamic clues that reinforce static warnings. We do not measure the overhead to compute the DP values because we do not report them by default. The analysis time was measured for one run each on a 4-core 3.1 GHz Intel i5 with 8 GB of RAM. Running the uninstrumented code of Daikon takes 41m:19s, whereas running the pre-instrumented code of Daikon takes 43m:38s. Running the uninstrumented code of Jackrabbit takes 7m:32s, whereas running the pre-instrumented code of Jackrabbit takes 8m:30s. Thus, the runtime slowdown of running pre-instrumented code for these residual investigations is small. The overhead will, however, vary based on the kinds of dynamic analyses installed, as we will also see in the next section.

3.2. Evaluation on Large Systems

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).

Volume of reports. Figure 3 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 say nothing about the *quality* of the reports—we examine this topic later.

By examining the dynamic potential metric in Figure 3, 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 Section 1. 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.

Quality of residual investigation 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 Figure 3, RFBI classifies 31 of the 43 dynamically exercised reports as bugs (i.e., reinforces them), thus rejecting 12 reports. Figure 4 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.

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

Fig. 4. Quality of RFBI warnings on the 43 dynamically exercised FindBugs reports.

From Figure 4, 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.

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. Figure 5 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 have been rejected correctly and one to be unclear. The unclear case involved the NetBeans test harness ignoring an exception caused by backing store problems during synchronization; 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 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

Bug Pattern	**#confirmed bugs/dynamic potential(#executed methods)/avg. times executed**						
	JBoss	**BCEL**	**Net Beans**	**Tomcat**	**JRuby**	**Apache CC**	**Groovy**
Bad Covariant Def. of Equals			0/0/0	0/0/0			
Cloneable Not Impl. Correctly			0/0/0	0/1/9		3/3/658	1/1/11
Dropped Exception	0/4/378		0/1/79	0/0/0	0/1/5		0/1/25
Equals Method Not Symmetric	0/0/0		0/0/0		1/1/2.6M		
Equal Objs ⇒ Eq. Hashcodes	1/2/1	20/20/77k	0/0/0	1/1/5k	2/2/3.5		1/3/14
Non-Short-Circuit Bool. Oper.			0/0/0		0/1/194		
Read Ret. Should Be Checked			0/0/0	0/0/0	0/0/0		1/1/571

Fig. 5. 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.

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`.

Bug Pattern	Execution time with and without instrumentation [min:s]						
	JBoss	BCEL	Net Beans	Tomcat	JRuby	ApacheCC	Groovy
Bad Covariant Def. of Equals			13:07	3:04			
Cloneable Not Impl. Correctly			7:17	5:20		5:25	39:15
Dropped Exception	655:01		16:00	16:05	17:08		41:44
Equals Method Not Symmetric	531:42		8:23		11:03		
Equal Objs ⇒ Eq. Hashcodes	363:48	1:20	13:07	3:03	3:44		7:25
Non-Short-Circuit Bool. Oper.			9:36		11:13		
Read Ret. 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

Fig. 6. 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.

— 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.

Runtime overhead. Figure 6 shows the runtime overhead of our residual investigation. We measured the analysis time of one run on a 4-core 2.4 GHz Intel i5 with 6 GB of RAM. 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, possibly 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.

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: we use a data structure library, two language runtime systems, a bytecode engineering library, a dynamic axiom detector, a content repository, 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. Six of our patterns have been described in an article [Hovemeyer and Pugh 2004b] by the FindBugs authors. That article describes a total of 18 patterns. For our eval-

uation 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. We selectively mention a few more FindBugs patterns in our discussion of the greater applicability of residual investigation in the next section.

— 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. The next section discusses some such directions, but only anecdotally.

4. RESIDUAL INVESTIGATION FOR RACE DETECTION

Residual investigation is a concept of much greater applicability than just our RFBI tool. The essence is to combine static analyses with predictive dynamic analyses that ascertain the validity or importance of the warned-about error. We show one example of the applicability of residual investigation by designing and implementing a residual analysis for race detection. Applying the residual analysis, we successfully analyze four real-world applications and evaluate the practical benefit experimentally.

4.1. Problem and Design

Static race detection is an analysis domain of very high recent interest [Lahiri et al. 2009; Sterling 1993; Engler and Ashcraft 2003; Voung et al. 2007; Naik et al. 2006; Abadi et al. 2006; Radoi and Dig 2013]. At the same time, static race detection is notoriously fraught with a high false positive rate. In the most common static report pattern that causes programmer objections, the "race" is on a variable that is never publicized, i.e., it is only ever accessed by one thread.[5] It is hard to statically establish whether a memory location (object field, class field or array element) is ever accessed by multiple threads, especially since the number of memory locations is not statically bounded. A *thread-escape* analysis is often used but it is by nature both a whole-program analysis and quite conservative and overapproximate. In this way, memory locations are often conservatively considered thread-shared when they are not, thus resulting in false static race reports. Indeed, non-shared locations are more likely to result in static race reports since the code will access them with no synchronization.

This kind of false positive is a perfect fit for applying residual investigation. Our heuristic here is to check whether a memory location is shared: A memory location is shared in a given execution iff two different threads access the location and the thread performing the first access remains active (i.e., not yet joined) at the time of the second access to the location.

In this light, how we confirm a static race is intuitively clear: given a static warning for a pair of racing events, we should find a pair of dynamic events on a matching shared memory location. Here, "matching" merely means that the program locations match up; the exact choice of pair(s) of dynamic events that we use is left as an implementation choice.

At the same time, we can reject a static race if the test suite confirms that one of the events in the race report can occur but offers no evidence that a pair of events matching the static race report occur dynamically.

[5]This was brought to our attention by Stephen Freund, during the conference presentation of our work.

Specifically:

— We *confirm* a statically detected race if we can find a dynamic event pair with a shared memory location that can act as a dynamic witness to the race.
— We *reject* any statically detected races that we cannot confirm and for which we have at least one matching dynamic memory access.

The correctness of a static race that is neither confirmed nor rejected is undetermined. From our past experience, undetermined races are generally due to a sparse test suite.

Observing dynamically whether a suspicious memory location is indeed thread-shared is easy. It is also significantly predictive: having the variable be shared is a much more general condition than actually observing a race. Nevertheless, observing a suspicious variable being shared *in conjunction* with the static race report significantly enhances the suspicion of a program error and warrants a reinforced report to the programmer.

To further explore the applicability of residual investigation in this high-value but diverse domain (compared to our main work), we implemented a prototype based on the above insight, applying it to a state-of-the-art static data race detection tool for Java, JChord [Naik et al. 2006].

4.2. Implementation

We tailored our implementation to the needs of confirming or rejecting JChord reports. This task was far from trivial, mainly because of the impedance mismatch in detecting and reporting concurrent events statically vs. dynamically, as discussed next.

— Our implementation applies only to races in non-JDK classes. A lot of JChord's reports are on fields or arrays inside JDK classes instead of fields or arrays inside the application classes. For example, if an application class contains a field of type `java.io.ByteArrayInputStream` and there are some races that can occur on the `ByteArrayInputStream` object's fields, JChord will report those races on the JDK class fields (e.g., a race will be reported on a field such as `buf` or `pos` inside the `ByteArrayInputStream` object). However, it is hard to map this to information collected from dynamic analyses. Without modifications of the underlying Java Virtual Machine, current dynamic race analysis can only report races on fields or arrays inside non-JDK classes (e.g., the field of type `ByteArrayInputStream` itself). The reason is that dynamic race analysis typically has to instrument each field or array of each class. Doing so for JDK classes would crash the Java Virtual Machine. To process JChord's reports through dynamic thread-local analysis, we limit our focus to races on shared variables inside non-JDK classes.
— We use dynamic stack traces to match paths of call sites in JChord's reports. Each JChord race report contains a field or an array, a pair of accesses represented by a pair of (call-)paths in the call graphs leading to a race on that field, and a pair of access types (read or write). Each path begins with the call site of the `main` method or the `run` method of class `java.lang.Thread` and ends with the call site of the access (at least one of them is a write) to the field. The internal states that we want to collect dynamically are quite similar. When a dynamic memory access event happens, if the memory sharing condition holds, we collect the field or array name representing the memory location, the most recent and current stack traces of accesses to the same memory location, and their access types. We use stack traces because static call sites are most accurately identified by using the full dynamic stack trace of the access instruction. However, collecting stack traces incurs a heavy runtime overhead. We minimize this overhead heuristically by collecting stack traces only for the first N unique accesses to a shared memory location. An access is uniquely determined given the stack trace and name of the variable on which the access is deemed to occur, i.e., if two memory accesses are by the same instruction with different stack traces or different variable names, they are considered different accesses. Furthermore, we only collect traces for the first M unique accesses to the same field/array in the program text. (The same field of different objects can point to different memory locations, so $M > N$. Cur-

rently $M = 300$, $N = 100$.) Once these limits are reached, any access to the memory location, or to all memory locations pointed by the same field is subsequently ignored.

The dynamic thread-local analysis is implemented using Roadrunner [Flanagan and Freund 2010]. Roadrunner manages the low-level details of adding probe code to the target program's bytecode at load time and creates a stream of events for field and array accesses, thread joins, etc. Our probe code performs thread locality checks, while recording the states of both thread-local heap memory locations and shared memory locations.

Our residual investigation of JChord is accomplished via a three-phase algorithm: *pre-analysis* phase, *instrumented execution* phase, and *cross-checking* phase.

In the *pre-analysis* phase, we identify all fields and arrays that appear in JChord race reports. Accesses to these fields and arrays will be instrumented at class load time in the *instrumented execution* phase.

In the *instrumented execution* phase, we run the instrumented program once and store the application's observations for use in the *cross-checking* phase. There are two kinds of observations that we utilize (with duplicates eliminated) during a dynamic memory access:

— *shared access observations*, which are quintuples $\langle f, r', a', r, a\rangle$ if stack traces r' and r for accesses a' and a are accessing the same memory location of field/array f (with or without synchronization) and at least one of a' and a is a write. The components of this quintuple correspond directly to a JChord race report. We check whether the most recent access to the same memory location as a or a' was by a different thread (establishing an access to a shared memory location, per our initial definition) and at least one of the two accesses is a write. If so, the algorithm records the shared access observation $\langle f, r', a', r, a\rangle$.
— *single access observations*, which are triples $\langle f, r, a\rangle$. All access history for all memory locations is recorded. The components of a single access observation correspond directly to either endpoint of a JChord race report. We use these triples to reject JChord races that cannot be confirmed by any shared access observations.

We pre-process the above shared and single access observations, since dynamic stack traces contain richer information than static call-paths. For instance:

a) There are extra method calls in stack traces by Roadrunner: For example, methods $\langle init\rangle$ or $\langle clinit\rangle$ appear for instance or class initializers; every time an instrumented method is called, an extra call to a dynamically generated method (specific to the method called) is recorded.
b) In dynamic traces, the instrumented method name is getting prefixed with `$rr__Original___$rr` by Roadrunner.
c) There are missing method calls in JChord's call sites due to its imprecise call-graph construction. Missing methods are ignored when matching is performed in the *cross-checking* phase.

In addition, recall that establishing the shared memory condition requires checking that the accessing threads are not joined. To do this, in the event of a thread joining, the id of the joining thread is recorded. Every time a memory location is accessed, the most recent access is considered to be able to share the memory with the current access only if the thread id of the most recent access is not one of a joined thread.

Another factor to consider is that operations inside probe code can be involved in concurrency errors since they may be executed concurrently by the same threads they are monitoring. To avoid concurrency errors in probe code, synchronization is added to the code that will affect the reporting of a race, such as checking thread-locality and updating the most-recent-access information for the memory location.

The *instrumented execution* phase computes, in the form of the above access observations, all the dynamic information needed to classify static races. The subsequent *cross-checking* phase performs a two-pass matching: a first pass is responsible for confirming JChord races and a second pass is responsible for rejecting the JChord races that are not confirmed. When confirming a JChord race,

	JChord	**Residual Investigation**	
project	Races	Confirmed	Rejected
EM3D	11	0	4
JUnit	6	2	3
Lucene	3132	0	49
Weka	721	0	71

Fig. 7. Summary of our experiments on race detection.

we check pairs of events in the JChord race against matching shared access observations (which we interpret as dynamic event pairs). If we do not find a shared access observation as witness to the static report, the JChord race is not confirmed and we continue to check its events against every single access observation to see if we can reject it, again following our earlier definitions. If we cannot reject the JChord race either, its classification is undetermined.

A critical step in either confirming or rejecting a JChord race is to define the logic of matching static events against dynamic events. We consider a dynamic event and a static event to match if (a) both refer to the same field or array name, (b) both have the same access type (read or write), and (c) the call stacks of the statically predicted invocation are *similar* to the ones in the dynamic invocation. For our purposes, two call stacks (one static, one dynamic) are similar iff all stack frames from the static invocation occur in the dynamic invocation, in order (but possibly separated by other stack frames). For instance, the call sites of the sequence $\langle foo, bar, baz \rangle$ appear in order in the stack frame sequence $\langle foo, bar, foobar, baz, fun \rangle$.

4.3. Experiments

We evaluate the efficiency of our prototype implementation by applying it to 4 open-source Java projects coupled with test harnesses: EM3D, JUnit, Lucene, and Weka. These test subjects were used in prior literature [Radoi and Dig 2013] and are heterogeneous: their domains are varied and their code sizes range from hundreds to hundreds of thousands lines. Also, as in prior work [Radoi and Dig 2013], we configure JChord so that:

— it reports races between instructions belonging to the same abstract thread.
— it filters out races between the non-main threads and main thread because the benchmarks only have parallelism in the form of parallel loops [Okur and Dig 2012]. Reports of races between the main thread and non-main threads are clearly false positives for these benchmarks.
— it ignores races in constructors.
— it does not use conditional must-not-alias analysis [Naik and Aiken 2007], as it is not currently available.
— it excludes a large number of known thread-safe JDK classes from analyses. A class is thread-safe if any invocation of it cannot be involved in races.

Our experimental results show that residual investigation precisely identifies real races and rejects false JChord warnings, in reasonable running time. All analyses were run on a machine with a 4-core Intel Xeon E5530, 2.40GHz processor and 24GB of RAM, using Oracle JDK version 1.6.45, with the maximum heap size set to 24GB.

Figure 7 shows the number of JChord races that our tool can confirm or reject in each benchmark. To assess how many of these confirmations and rejections are correct, we manually inspected JChord races for EM3D and JUnit. We did not inspect race warnings for Lucene and Weka due to a high number of warnings by JChord and the large size and non-trivial complexity of both Lucene and Weka.

Out of 11 EM3D races detected by JChord, 4 were rejected by our tool. Further inspection revealed that the 4 rejections were justified: EM3D is race-free. JUnit gave rise to 6 JChord races and 2 of them were confirmed by our tool. Examining the source code indicated that the 2 confirmed races were real races, both of which were related to unsynchronized accesses to a field used for

	JChord	**Residual Investigation**			**Plain**
project		Pre	Instrumented	Checking	
EM3D	51.00	0.01	0.81	0.01	0.11
JUnit	125.67	5.61	6.50	15.50	0.36
Lucene	138.67	18.25	15.08	23.62	6.56
Weka	150.67	37.79	946.29	35.13	3.90

Fig. 8. Average running time in seconds for residual investigation on race detection.

printing JUnit test result. Again for JUnit, our tool correctly rejected 3 JChord races: All of these 3 pairs of accesses were well-ordered. For the remaining 2 benchmarks, Lucene and Weka, our tool classified fewer than 10% of the JChord race warnings as confirmed or rejected.

Figure 8 collects the runtime overhead of our residual investigation tool, in seconds. "JChord" gives the running time of JChord for the benchmark. "Pre", "Instrumented", and "Checking" report the running time in the *pre-analysis* phase, *instrumented execution* phase, and *cross-checking* phase of residual investigation, respectively. "Plain" reports the running time of the benchmark programs without any instrumentation.

The overall runtime overhead required for the three phases of residual investigation is occasionally significant but always practically feasible (especially considering the human time benefits that a better race classification affords). In particular, as can be seen in the "Instrumented" column, applications typically slow down by a factor of 2 to 20 during the *instrumented execution* phase, although the memory-intensive Weka results in roughly 300x slowdown. This is hardly surprising. Dynamic race detection tools, such as Eraser [Savage et al. 1997] and FastTrack [Flanagan and Freund 2009], also incur a large slow-down on *Weka*. Eraser also suffers a roughly 300x slowdown; FastTrack cannot finish running in 24 hours since it requires $O(n)$ time, with n being the total number of threads created, and for Weka n grows to 15021.

5. GREATER APPLICABILITY OF RESIDUAL INVESTIGATION IN OTHER DOMAINS

In the following, we discuss several kinds of analyses to which residual investigation is applicable. We also present our experimental experience from attempting to apply residual investigation to randomly chosen open-source projects. This experience illustrates vividly the standard caveat not just of residual investigation but of every dynamic analysis: the quality of results depends on the test suite.

5.1. Applicability to Other Analyses

Static analysis typically attempts to be exhaustive and, thus, suffers the possibility of false positives. The challenge for applying residual investigation is to identify common failure scenarios that affect the false positive rates of the analysis yet can be easily ascertained dynamically, even when no fault is observed.

5.1.1. Flow Analyses: SQL Injection. Another kind of static analysis that will often result in false positives is analysis that deals with string operations and attempts to discern the structure of programmatically generated strings. An example of this is SQL injection analysis for SQL queries created in code written in an imperative language (typically Java, C++, C#). Although powerful analyses have been proposed (e.g., [Gould et al. 2004]), it is difficult to make an accurate determination of the possible contents of generated strings, and this difficulty is compounded with the obligation to also track how the strings can be used. Residual investigation approaches can be of help in this domain. For instance, in a course-based evaluation of the industrial static code analyzer Fortify[6], D'Souza et al. [D'Souza et al. 2007] observe false positives in "the report of a SQL injection at places where [they] did not find any hint of a dynamic SQL query generation." This observation directly suggests an easy dynamic test ("does the suspicious string ever get included in

[6]http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812#.UtAcLvjft0w

a SQL query?") that will eliminate the false positives, yet will allow reporting the error even when the current execution does not exhibit any fault. Thus, our argument is that if we combine a static SQL injection report with dynamic taint analysis [Newsome 2005] (i.e., tracking of which user-influenced strings flow where) we get a much higher-confidence SQL injection report. Note that in this case the taint analysis by itself is not an indication of a fault, unlike other security analyses where tainting of a function's input by user data is direct evidence of a problem.

Note that combinations of static and dynamic analyses for SQL injection detection have already been explored [Huang et al. 2004; Halfond and Orso 2005]. Huang et al. [2004] insert dynamic guards to secure a variable that is found to be involed in an insecure statement by static verification. Halfond and Orso [2005] use static analysis to build a conservative model of the legitimate queries generated by the program, yet follow it with a dynamic analysis that checks for conformance with that model. These approaches, however, do not quite follow the residual investigation pattern: the static analysis by itself does not detect errors but prepares the ground for a dynamic analysis that will report faults. Still, the residual investigation approach is largely applicable to this domain, if not already applied, to an extent.

Saner [Balzarotti et al. 2008] composes static analysis and testing to find bugs of the input validation process for cross-site scripting and SQL injection flaws. Saner statically identifies incorrect or incomplete code sections of input validation routines and uses testing to confirm that these code sections are actually ineffective. Instead of using testing, residual investigation employs predictive dynamic analyses to confirm statically reported bugs. Thus, given the same test suite, residual investigation is less likely to miss a true vulnerability (but still nearly precise).

5.1.2. Other FindBugs Patterns. FindBugs currently supports several hundred bug patterns, each with its own analysis, ranging from plain textual matching to more complicated type- and flow-based analyses. We implement a set of bug patterns in RFBI (mostly from the 18 bug patterns in the original FindBugs publication [Hovemeyer and Pugh 2004b]) and examine them in our main evaluation, but, as already mentioned, residual investigation is applicable to a lot more. Examples include:

— The "*Covariant compareTo method defined*" bug pattern reports classes that define method `compareTo` covariantly (i.e., with an argument type that is not `java.lang.Object` but a subtype of it). This, and other bug patterns on erroneous overriding of standard methods, can be handled in much the same way as the "Bad Covariant Definition of Equals" pattern, discussed earlier.
— The "*Private method is never called*" pattern is a source of false positives because some private methods are intended to be called only via reflection. In this case the method is private for static code, but public for reflective code. For this to happen, the `setAccessible` method needs to be called on the `Method` object corresponding to the suspicious method. Thus, the presence of a `setAccessible` call on this method is a hint to suppress the static error report.
— Reflection and dynamic loading are common in Java enterprise applications and invalidate several patterns. Yet the uses of reflection are typically simple and only limited aspects of reflective behavior change dynamically. The "*Call to equals comparing unrelated class and interface*" bug pattern, as well as the "*Call to equals comparing different interface types*" pattern both assume a closed view of the inheritance hierarchy. Dynamic code can invalidate the patterns by loading a new class that is a subtype of both compared types. The error warning is thus reinforced if we track loaded classes and observe that, indeed, while running the test suite no such common subtype arises.

5.2. Applicability Relative to Test Suites

It is worth considering the preconditions for applicability of residual investigation in terms of the analyzed program. Clearly, the program being examined should have a means for being exercised under representative usage. That is, one needs either an off-line system-level test suite or a human tester that will run the program through actual usage scenarios. In the absence of the above, residual investigation is not likely to bear fruit.

The state of the practice regarding test suites is rather grim, however. The vast majority of open source projects maintain no test suite alongside the code, or only integrate a rudimentary unit test suite. For an example, we considered the SF100 benchmark suite [Fraser and Arcuri 2012], which has recently been the focus of significant attention. SF100 selects at random 100 open-source projects from the SourceForge repository and puts them forward as a statistically sound sample for empirical SE results. We studied at random 40 of the 100 benchmark programs. A handful of them were further excluded as they were outdated—e.g., providing an API for Amazon's XML Web Service 3.0, which has been long replaced by Amazon Cloud Computing. (These benchmarks could not run in our environment, although the intent of the SF100 suite is to include all dependencies in the benchmark package.) We found that the vast majority of the SF100 projects do not contain a test suite for unattended system-level testing. This is perhaps expected, since most of the SF100 benchmarks are small-to-medium size: more than four-fifths of the programs have fewer than 100 classes, more than one-fifth have fewer than 10.

As a result, RFBI did not produce any useful results for our sampling of SF100 programs: the dynamic potential (DP) metric for all bugs was zero, reducing RFBI to merely FindBugs, which often also did not manage to produce good reports. This reinforces the dependence of residual investigation on a reasonable test suite. As already seen, this is no more stringent a requirement than already satisfied by test suites supplied by the creators of large, mature open-source projects (such as JBoss, BCEL, NetBeans, Tomcat, etc.).

Of course, this does not mean that residual investigation is not useful for the SF100 programs. However, it has to be used in conjunction with the usual testing process that developers employ for these programs. This is likely to be manual testing under inputs representative of actual usage. In principle, RFBI is fully compatible with the notion of manual testing: instead of collecting dynamic program behavior under automatic test execution, we could also collect this information during manual program execution. Generally, testing for residual investigation has a low bar to clear—we do not need to amass many observations, since just a few runs of a program are enough to reinforce or discount static warnings.

Secondarily, we posit that statistically sound benchmark samples, such as the SF100, will be of greater value if they also include a *biased* sampling of applications based on specific characteristics, such as size or the integration of a complete test suite. Some empirical SE studies (e.g., on automatic test generation, which was the first application of the SF100 [Fraser and Arcuri 2012]) have no such need, but others (such as our study) do.

5.3. Applicability to Refactoring

Refactoring is the process of changing a program's structure without altering its behavior. Developers have used test suites to manually gauge if behavior remains unchanged [Fowler et al. 1999], but automatic refactoring tools provide a complementary approach through static analysis [Griswold and Notkin 1990; Mens and Tourwe 2004]. These static analyses, in turn, necessarily come with trade-offs in the same way that the analyses of static bug-checkers do. Consider a seemingly innocuous refactoring such as "rename." This refactoring will systematically replace all occurrences of one particular identifier (e.g., a method name) throughout the program by another. In a language like Java, this transformation is not necessarily behavior-preserving, as the relevant method might be called through reflection, or via an external client program that is not visible to the refactoring engine.

Existing refactoring tools might thus plausibly reject any attempt at renaming a large subset of Java identifiers, i.e., those that could be read or modified via Java reflection. For example, program metamorphosis [Reichenbach et al. 2009], a generalization of traditional refactoring, materializes such concerns as persistent warnings that users can review during the refactoring process. This is analogous to a static bug report issued by FindBugs in that developers may now raise objections, such as "this method is private and not supposed to ever be called via reflection."

Production refactoring tools anticipate this objection to the point where they concede defeat to it: we are not aware of any production refactoring engine that will reject a rename refactoring because it cannot exclude the possibility of behavioral change.

Residual investigation can serve to improve recall of such lenient refactoring systems, as well as precision of more conservative systems (such as the aforementioned program metamorphosis system). For example, for any newly renamed method, a refactoring system can statically determine name captures (e.g., accidental method overriding or overloading). The system can determine *some* instances of behavioral change due to reflection, by analyzing string usage [Christensen et al. 2003]. To detect the remaining cases of reflective method invocation, we can introduce two dynamic checks: (1) a search for reflective invocations of methods with the old (pre-renaming) name, regardless of the dynamic type of their target, and (2) any use of reflection on instances of the refactored class (perhaps including its superclasses). If either or even both checks trigger, the risk of a behavioral change increases and may now warrant a more severe warning to the user.

These dynamic checks can then complement the existing static checks in the same vein as the dynamic analyses we have described for static bug checkers: whenever a static check cannot conclusively determine that the refactoring is safe or unsafe, it can emit the above checks as additional safeguards of behavior preservation.

6. 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 access 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 [Bodik et al. 2000] have been developed that can guarantee some reads as being within bounds, so that only the remaining ones have 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 [Barnett et al. 2004] is a prime example: it can prove some pre- and post-conditions statically and generates runtime checks for the remaining ones. Gopinathan and Rajamani [2008] 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 [2007], Bodden [2010] and Bodden et al. [2007], 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 be safely 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. [2009] 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 [Csallner and Smaragdakis 2005] and DSD-Crasher [Csallner and Smaragdakis 2006; Smaragdakis and Csallner 2007] 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 limit is reached. While such proof is very convincing, it also narrows the technique's scope. I.e., our earlier tools could only confirm very few warnings. In our current residual investigation, we relax this strict interpretation and also consider predictive clues that are likely to confirm a static warning.

Dynamic symbolic execution is a recent combination of static and dynamic program analysis [Godefroid et al. 2005; Cadar and Engler 2005; Tillmann and de Halleux 2008; Islam and Csallner 2014]. 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) [Chen and Roşu 2007a] shows how runtime monitoring of correctness conditions can be implemented more efficiently, even without a prefixed static analysis. 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 using data mining. Kremenek et al. [2004] 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 [2007] 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 of good quality. For static tools such as FindBugs, which analyze Java bytecode to generate warnings, they also require compilation of each revision. Bodden et al. [2008] use a machine-learning approach to filter out false positives of their static analyses for validating a finite state property. They identify cases where imprecise static analysis information can appear due to factors such as dynamic class loading and interprocedural data flow. They then use each case as a feature for decision tree learning. The key differene between our approach and data-mining-based ones is that our approach considers the possible objections to certain static error reports. As a result, we can validate more complicated error situations that require understanding what a programmer has in mind in practice when writing their code.

Besides Eraser [Savage et al. 1997], some other related work on race detection comes under the label of predictive race analysis. Chen and Roşu [2007b] predict races by logging only relevant information in a program trace and then model-checking all feasible trace permutations. jPredictor [Chen et al. 2008] presents a polynomial algorithm that can search a thread scheduling for a potential race that did not occur in the observed execution. jPredictor is not sound. Smaragdakis et al. [2012] define the causally-precedes relation (CP) that weakens the traditional Happens-Before relation. The CP approach can generalize beyond an observed execution and guarantee soundness and polynomial complexity. These sophisticated predictive approaches can offer greater precision than residual investigation in terms of race detection, whereas our residual investigation based race detector can have fewer false negatives, due to the better coverage of static analysis tools and the conservativeness of our dynamic analysis.

The precision of bug detection software can be greatly enhanced with human supervision at residual steps. After an automated static program verification step, Dillig et al. [2012] rely on humans to help reduce false positives by asking humans simple and relevant questions. Xie [2012] proposes to use humans to provide guidance to otherwise intractable problems faced by test case generators. It would be interesting to combine such techniques in our approach. For instance, residual investigation's precision is influenced by the native test suite coverage. With the help of human users, additional test cases can be constructed if required.

7. 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. 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.

ACKNOWLEDGMENTS

We thank Bill Pugh for helpful comments and interesting discussions. Stephen Freund suggested race detection as a possible domain of application for residual investigation and helped us set up RoadRunner. Cosmin Radoi provided the parallel implementation of the evaluation suites for race detection. This material is based upon work supported by the National Science Foundation under Grants No. 0917774, 0934631, 1017305, 1115448, and 1117369. We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant (PADECL) and a European Research Council Starting/Consolidator grant (SPADE); and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award (MORPH-PL).

REFERENCES

Martín Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 2 (March 2006), 207–255.

Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. In *Proc. 19th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 241–252.

D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and Giovanni Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. 387–401. DOI: http://dx.doi.org/10.1109/SP.2008.22

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2004. The Spec# programming system: An overview. In *Proc. International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*. Springer, 49–69.

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

Eric Bodden, Laurie Hendren, and Ondřej Lhoták. 2007. A staged static program analysis to improve the performance of runtime monitoring. In *Proc. 21st European Conference on Object-Oriented Programming (ECOOP)*. 525–549.

Eric Bodden, Patrick Lam, and Laurie Hendren. 2008. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 36–47.

Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating array bounds checks on demand. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 321–333.

Cristian Cadar and Dawson R. Engler. 2005. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th International SPIN Workshop on Model Checking Software*. Springer, 2–23.

Feng Chen and Grigore Roşu. 2007a. Mop: An efficient and generic runtime verification framework. In *Proc. 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 569–588.

Feng Chen and Grigore Roşu. 2007b. Parametric and Sliced Causality. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 240–253.

Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. 2008. jPredictor: A Predictive Runtime Analysis Tool for Java. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 221–230. DOI:http://dx.doi.org/10.1145/1368088.1368119

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Extending Java for High-Level Web Service Construction. *ACM Transactions on Programming Languages and Systems* 25, 6 (November 2003), 814–875.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 50–62.

Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering (ICSE)*. ACM, 422–431.

Christoph Csallner and Yannis Smaragdakis. 2006. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 245–254.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated Error Diagnosis Using Abductive Inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 181–192. DOI:http://dx.doi.org/10.1145/2254064.2254087

Derek D'Souza, Yoon Phil Kim, Tim Kral, Tejas Ranade, and Somesh Sasalatti. 2007. Tool Evaluation Report: Fortify. http://www.cs.cmu.edu/ aldrich/courses/654/tools/dsouza-fortify-07.pdf. (April 2007). Accessed Jan. 2014.

Matthew B. Dwyer and Rahul Purandare. 2007. 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)*. ACM, 124–133.

Dawson R. Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 237–252.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb. 2001), 99–123.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. DOI:http://dx.doi.org/10.1145/1542476.1542490

Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '10)*. ACM, New York, NY, USA, 1–8.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gordon Fraser and Andrea Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 178–188.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 213–223.

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

Carl Gould, Zhendong Su, and Premkumar Devanbu. 2004. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 697–698.

W. G. Griswold and D. Notkin. 1990. *Program Restructuring as an Aid to Software Maintenance*. Technical Report. University of Washington, Seattle, WA, USA. citeseer.ist.psu.edu/griswold91program.html

William G.J. Halfond and Alessandro Orso. 2005. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proc. 3rd International Workshop on Dynamic Analysis (WODA)*. 22–28.

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

David Hovemeyer and William Pugh. 2004b. Finding bugs is easy. *SIGPLAN Notices* 39, 12 (Dec. 2004), 92–106.

David Hovemeyer and William Pugh. 2007. 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)*. ACM, 9–14.

Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, New York, NY, USA, 40–52. DOI:http://dx.doi.org/10.1145/988672.988679

Mainul Islam and Christoph Csallner. 2014. Generating test cases for programs that are coded against interfaces and annotations. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23 (May 2014), 21:1–21:38. Issue 3.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *Proc. 15th European Conference on Object Oriented Programming (ECOOP)*. Springer, 327–353.

Sunghun Kim and Michael D. Ernst. 2007. 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)*. ACM, 45–54.

Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation exploitation in error ranking. In *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 83–93.

Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. 2009. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Proc. 21st International Conference on Computer Aided Verification (CAV)*. Springer, 509–524.

Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. 2012. Residual Investigation: Predictive and Precise Bug Detection. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. 298–308.

Tom Mens and Tom Tourwe. 2004. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (2004), 126–139. DOI:http://dx.doi.org/10.1109/TSE.2004.1265817

Madanlal Musuvathi and Dawson Engler. 2003. Some lessons from using static analysis and software model checking for Bug Finding. In *Proc. Workshop on Software Model Checking (SoftMC)*. Elsevier.

Mayur Naik and Alex Aiken. 2007. Conditional Must Not Aliasing for Static Race Detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 327–338.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 308–319.

James Newsome. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

Semih Okur and Danny Dig. 2012. How Do Developers Use Parallel Libraries?. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 54, 11 pages. DOI:http://dx.doi.org/10.1145/2393596.2393660

Cosmin Radoi and Danny Dig. 2013. Practical Static Race Detection for Java Parallel Loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 178–190.

Christoph Reichenbach, Devin Coughlin, and Amer Diwan. 2009. Program Metamorphosis. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 394–418.

Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A comparison of bug finding tools for Java. In *Proc. 15th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 245–256.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multi-threaded programs. In *Proc. 16th Symposium on Operating Systems Principles (SOSP)*. ACM, 27–37.

Yannis Smaragdakis and Christoph Csallner. 2007. Combining Static and Dynamic Reasoning for Bug Detection. In *Proc. International Conference on Tests And Proofs (TAP)*. Springer, 1–16.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 387–400.

Nicholas Sterling. 1993. Warlock: A Static Data Race Analysis Tool. In *USENIX Winter Technical Conference*. 97–106.

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex - White Box Test Generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*. Springer, 134–153.

Aaron Tomb, Guillaume P. Brat, and Willem Visser. 2007. Variably interprocedural program analysis for runtime error detection. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 97–107.

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static race detection on millions of lines of code. In *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 205–214.

Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. 2005. Comparing Bug Finding Tools with Reviews and Tests. In *Proc. 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom)*. Springer, 40–55.

Tao Xie. 2012. Cooperative Testing and Analysis: Human-Tool, Tool-Tool and Human-Human Cooperations to Get Work Done. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. 1–3. DOI:http://dx.doi.org/10.1109/SCAM.2012.31

Misha Zitser, Richard Lippmann, and Tim Leek. 2004. 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)*. ACM, 97–106.