COMBINING STATIC AND DYNAMIC ANALYSIS FOR BUG DETECTION AND PROGRAM UNDERSTANDING

A Dissertation Outline Presented

by

KAITUO LI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

?

School of Computer Science

COMBINING STATIC AND DYNAMIC ANALYSIS FOR BUG DETECTION AND PROGRAM UNDERSTANDING

A Dissertation Outline Presented

by

KAITUO LI

Approved as to style and content by:

_____

Yannis Smaragdakis, Chair

_____

George Avrunin, Member

_____

?, Member

_____

?, Member

_____

Lori Clarke, Department Chair
School of Computer Science

CHAPTER 1

INTRODUCTION

Software practitioners have traditionally faced many problems with testing, documenting and maintaining software modules and/or systems. These problems are often the result of poor understanding of program properties, over-reliance on specific bug detecting tools, and so on. Various static and dynamic program analysis techniques can be used to remedy these problems. Each flavor of analysis has its own advantage. Dynamic analysis excels in control-flow accuracy, while static analysis is good at data-flow richness/generality [23]. One cannot make a statement that static analysis is always better than dynamic analysis or vice versa. This has led researchers to devise combinations of static and dynamic analyses (e.g., [2–4, 8, 12, 25, 26]) in order to achieve the best of both worlds.

Dynanmic symbolic execution [20] is such an example. Symbolic execution is a static analysis technique. For a vector of input values, a method performs an execution following a particular path, then returns output results. Instead of using concrete values, symbolic execution represents input and output values of a method by symbolic names. In an object-oriented programming language, symbolic execution can create symbolic representations of class variables. Symbolic execution can compute the path condition of each path in an analyzed program. A path condition is a conjunct of branch conditions along the path. A branch condition is a symbolic expression over input symbols. Symbolic execution cannot reason about native method calls because the source code of native method calls are not available. Dynamic symbolic execution resolves this by symbolically executing a program in parallel with the actual program execution. Calls to native methods are replaced with concrete values of the method call from the execution.

I propose new combinations of static and dynamnic analysis for bug detection and program understanding. The remainder of this proposal is organized as follows. Chapter 2 summarizes work done so far. Work I propose to do is presented in Chapter 3. Related work is discussed in Chapter 4. Chapter 5 concludes with a schedule for compelting the work proposed.

CHAPTER 2

WORK DONE SO FAR

2.1 Residual Investigation: Predictive and Precise Bug Detection

My work [14] with Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis identified and presented a new kind of combination of static and dynamic analyses, which we term residual investigation. A residual investigation is a dynamic analysis whose purpose is to be the run-time agent of a static analysis and to identify with higher certainty whether the error identified by the static analysis is likely true. In other words, one can see the dynamic analysis as the residual of the static analysis at a subsequent stage: that of program execution. The distinguishing feature of a residual investigation, compared to past static-dynamic combinations, is that the residual investigation does not intend to report the error only if it actually occurs, but to identify general conditions that confirm the statically detected error. That is, a residual investigation is a predictive dynamic analysis, predicting errors in executions not actually observed.

Consider as an example the "equal objects must have equal hashcodes" analysis (code-named HE) in the FindBugs static error detector for Java [1, 9, 10]. 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 warn-

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

2.2 Enhanced Dynamic Invariant Inference Using Second-Order Constraints

Dynamic invariant inference is fundamentally a search in the space of propositions over program variables for the tiny subset of propositions that are both true (on all executions) and relevant (to the programmer, or to some client analysis such as a theorem prover or test data generator). Any such search requires a way of generating candidate propositions and a

way of evaluating them. Even limiting consideration to propositions of limited length, over a particular vocabulary of operations and relations among variables, the space is enormous, and for a long time it seemed folly to even attempt such a search based on dynamic program execution. Daikon [7] was the first demonstration that a generate-and-test tactic could be feasible, generating a moderately large set of candidate propositions from syntactic templates and quickly pruning the vast majority as they were refuted by program execution. The success of Daikon generated a great deal of excitement and a flurry of research activity. Attempts to improve candidate generation include using program text (especially predicates governing control flow) to choose candidate propositions that are more likely to be related to program logic and, in DySy [5], generating propositions from symbolic execution along actual program execution paths. Attempts to more effectively prune false propositions have so far been limited to generating more thorough test suites [8, 22].

To enhance the dynamic invariant detection approach my work [15] with Christoph Reichenbach, Yannis Smaragdakis, and Michal Young identified and described the idea of second-order constraints that can help improve the consistency and relevance of dynamically inferred invariants. We called these constraints second-order because they are constraints over constraints: they relate classes of invariants (first-order constraints). Such second-order constraints can be known even though the invariants are unknown. We defined a vocabulary of common and useful second-order constraints and provided an implementation of the constraints in our vocabulary in the context of the Daikon system. Experimental results demonstrate that a minimal effort on specifying second-order constraints is sufficient to produce several valid and helpful invariants even in systems of significant size, such as the Apache Commons Collections.

2.3 Dynamic Example Data Generation for Dataflow Programs

Dataflow programming has emerged as an important data processing paradigm in the area of big data analytics. Dataflow programming consists of specifying a data processing

program as a directed acyclic graph. When a user writes a dataflow program, he/she will typically employ example data or test cases to validate it. Validating with large real data is impractical, both for reasons of efficiency (running on large data sets takes a long time) and for reasons of ease-of-validation (it is hard to tell whether the result is what was expected). One alternative is to sample the real data available. The sample data need to thoroughly exercise the program, covering all key behavior of each dataflow operator. This is very hard to achieve via random sampling, however. For instance, equi-joining two sample data tables of small size is likely to produce an empty result, if the values being joined are distributed arbitrarily. Another alternative is to synthesize representative data. Such data synthesis is complicated by the complexity of dataflow language operators as well as by the presence of user-defined functions. Current state-of-the-art in example data generation for dataflow programs [17] is of limited help. Such techniques can generate high-coverage data for dataflow programs with simple constraints. However, for dataflow programs with complex constraints, e.g., with numerous filters, arithmetic operations, and user-defined functions, the generated data are incomplete due to shortcomings in constraint searching and solving strategies.

My work [13] with Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner addressed the problem of efficient example data generation for complex dataflow programs by bringing *symbolic reasoning* to bear on the process of sample data generation. We presented the first technique and system for systematically generating representative example data using dynamic symbolic execution (DSE) [2, 8, 24] of dataflow programs. Our concrete setting is the popular Pig Latin language [18]. We adapted the technique of dynamic-symbolic execution to the domain of dataflow languages. By doing so, we exploited the unique features of this domain, thus enabling high coverage. Specifically, we exploited the absence of side-effects in order to perform a multiple-path analysis: observations on the values of a user-defined function on different execution paths can help solve constraints involving the user-defined function.

CHAPTER 3

PROPOSED WORK

In our evaluation of residual investigation, we can only confirm or down-play 43 bugs out of 436 FindBugs reports. This is largely because a lot of program paths have not been exercised by the native test suite. We propose to improve the recall of residual investigation by making existing test suites better. To achieve this, we generate tests automatically using dynamic symbolic execution. In particular, we focus on generating system tests. Generated unit tests might not be realistic, while generated system tests attract less debate about the realism of the test cases, because users can see how the tests are used on the client side. The challenge of using dynamic symbolic execution to generate system tests is that not all system tests can be generated, because existing dynamic symbolic execution engines are not powerful enough to deal with complicated real-world code (e.g., existing tools cannot generate sequences of method calls to construct desired object states) [27]. To address this challenge, we propose to tap into existing system tests to approximate new system tests. The approximate new system tests consist of states (i.e. assignments to variables) induced by existing system tests and assignments to some variables that are key to triggering a dynamic condition of residual investigation. The approximate new system tests are not system tests. They are unit tests with some system state attached. They are justified by existing system tests so that the unit tests can connect with some execution path tracing back all the way up to program entry (i.e., the main function). We cannot guarantee the correctness of such an approximation. It is likely that the argument assignments in the unit tests contradict with the states with which we pair the unit tests.

Let us walk through the concepts involved with an example. Consider the following code

```
main() {
   v =
   foo(v);
}

foo(v) {
  bar(v2);
  baz(v3);
}

bar (x) {
  foobar(x2);
  foobaz(x3);
}

baz(y) {
  foobaz(..);
}

foobar(x) {

}

foobaz(...) {

}
```

The corresponding static call graph (approximated) is

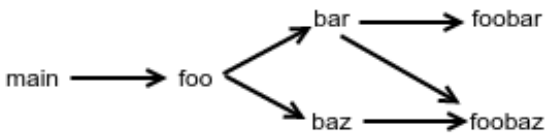

Figure 3.1: The call graph of the example program

We define a potential error revealing method (PERM) as the method that might trigger the dynamic condition of residual investigation for a FindBugs bug pattern. For example, FindBugs reports an error when a class overrides the equals(Object) method but not the hashCode() method. Residual investigation would install dynamic checks for the condition

that Object.hashCode() is called on an object of a class that redefines equals(Object) and inherits the implementation of hashCode(). The method that contains a call to hashCode() on the suspect class (i.e. the class that redefines equals and inherits hashCode) or its subclass is the PERM for this bug pattern. Another example is that FindBugs statically detects dropped exceptions and reports them. Residual investigation examines which methods end up being dynamically invoked in the suspect code block and watches whether the same methods ever throw the dropped exception when called from anywhere in the program. The methods that contain calls to a method that can be invoked in the suspect code block and can throw exceptions are the PERMs for this bug pattern. We analyze the PERMs using text search like grep and static analysis such as call graph analysis and Java reflection API. For example, for the dropped exception pattern, we analyze the types of the exceptions declared to be thrown by the the methods in the suspect code block (suspect methods) using Java reflection and then search for methods whose names are the same as that of any suspect method and throw the same exceptions using grep. Suppose the method foobar in the above code example is our PERM.

Once finding a PERM, we interpose k levels back in the call graph from a PERM and call this a switching method. Suppose k=1. In our example, the method bar is a switching method. At the beginning we run existing system tests. When we hit a switching method, we start exploring (i.e. switching to dynamic symbolic execution mode, instead of just dynamic execution). To allow exploration, we relax (i.e., forget) some concrete values. The first cut is to forget the arguments of switching methods. In short, the invocation of a switching method turns on dynamic symbolic execution mode during the execution of an existing system test. We declare victory when we find an approximate system test that calls the PERM method: the unit test case for bar that invokes foobar during its execution given the relaxed state. In our example, this corresponds to a path like

Figure 3.2: The path that calls the PERM method in the example program

The dotted line arrow marks the path before we start dynamic symbolic exploration (i.e. the path exercised by the existing system test), while the solid line arrow marks the path that the generated unit test case for bar establishes.

PERM definitions for other bug patterns not mentioned in the above text:

- Bad Covariant Definition of Equals: FindBugs detects that programmers write equals method that accept a parameter of type other than Object. Residual investigation checks whether the ancestral equals method, Object.equals(Object), is called on an instance of a class that has a covariant definition of equals. The PERM for this bug pattern is the method that contains a call to equals() on the suspect class (i.e. the class redefines equals and inherits hashCode) or its subclass.

- Cloneable Not Implemented Correctly: This bug pattern does not require native test suite. Thus, we do not need to improve existing test suite.

- Equals Method Overrides Equals in Super-class and May not Be Symmetric: FindBugs detects whether 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. Residual investigation dynamically calls both equals method whenever it observes a comparison involving a contentious object and test if the results match. The PERM for this bug pattern is the method that contains a call to equals() on the suspect superclass or subclass.

- Non-Short-Circuit Boolean Operator: FindBugs issues warnings for use of `&` and `|` inside the condition of an if statement. Residual investigation checks for actual side-

effects on the right-hand side of a non-short-circuiting boolean operator. The PERM for this bug pattern is the method that contains a `&` or `|` in an if statement.

- Read Return Should be Checked: FindBugs checks if the return value from read in java.io.InputStream is ignored. Residual investigation waits until it sees a read method on an object of any subclass of InputStream returns fewer bytes than requested (even for a call that does check the return value). The PERM for this bug pattern is the method that contains a call to read on the suspect class.

CHAPTER 4

RELATED WORK

Similar to us, hybrid concolic testing [16] interleaves random testing and concolic testing to increase coverage. The switch from random testing to concolic testing happens when random testing does not find new coverage points after a predetermined steps; the switch from concolic testing to random testing happens when concolic testing finds an input to an uncovered branch goal (though concolic testing may need to explore several feasible execution paths to generate the new input). But hybrid concolic testing is not designed to trigger one specific bug condition. We are targeting at generating tests to trigger one specific bug condition, using call graph search and dynamic symbolic execution.

[6] saves system test states and restores those states as regression tests. We also save system test states. But our purpose is helping setting up environment (i.e. assignments to heap variables) for starting dynamic symbolic execution of the switching method.

The closest work to us is [19], which uses system tests to set up the environment. Different from [19] that uses symbolic execution to generate unit tests, we use dynamic symbolic execution to generate unit tests. This enables us to handle native method calls and some complicated constraints by replacing them with concrete values.

CHAPTER 5

MANAGEMENT AND TIMELINE

My overall goal is to defend the dissertation by September 2015. I plan to submit a paper on the proposed work to ICSE 2016; the deadline for this conference is September 4, 2015. Right now the underlying dynamic symbolic execution engine Dsc [11] is very similar to jCute [21] released by University of Illinois at Urbana-Champaign. I plan to improve Dsc to make it become cutting-edge and open-sourced by the end of next March. I plan to work out the requried static analysis (e.g. call graph analysis) by May 2015. I plan to finish the implementation of the proposed work by July 2015. I will be writing the dissertations, performing rounds of implementations, and plan to complete the dissertation by September 2015.

BIBLIOGRAPHY

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

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

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

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

[5] Csallner, Christoph, Tillmann, Nikolai, and Smaragdakis, Yannis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)* (May 2008), ACM, pp. 281–290.

[6] Elbaum, Sebastian, Chin, Hui Nee, Dwyer, Matthew B., and Dokulil, Jonathan. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, ACM, pp. 253–264.

[7] Ernst, Michael D., Cockrell, Jake, Griswold, William G., and Notkin, David. Dynamically discovering likely program invariants to support program evolution. In *Proc. 21st International Conference on Software Engineering (ICSE)* (May 1999), IEEE, pp. 213–224.

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

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

[10] Hovemeyer, David, and Pugh, William. 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)* (June 2007), ACM, pp. 9–14.

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

[12] Islam, Mainul, and Csallner, Christoph. 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.

[13] Li, Kaituo, Reichenbach, C., Smaragdakis, Y., Diao, Y., and Csallner, C. Sedge: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* (Nov 2013), pp. 235–245.

[14] Li, Kaituo, Reichenbach, Christoph, Csallner, Christoph, and Smaragdakis, Yannis. Residual investigation: Predictive and precise bug detection. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)* (July 2012), pp. 298–308.

[15] Li, Kaituo, Reichenbach, Christoph, Smaragdakis, Yannis, and Young, Michal. Second-order constraints in dynamic invariant inference. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, ACM, pp. 103–113.

[16] Majumdar, Rupak, and Sen, Koushik. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 416–426.

[17] Olston, Christopher, Chopra, Shubham, and Srivastava, Utkarsh. Generating example data for dataflow programs. In *Proc. 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2009), ACM, pp. 245–256.

[18] Olston, Christopher, Reed, Benjamin, Srivastava, Utkarsh, Kumar, Ravi, and Tomkins, Andrew. Pig latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2008), ACM, pp. 1099–1110.

[19] Păsăreanu, Corina S., Mehlitz, Peter C., Bushnell, David H., Gundy-Burlet, Karen, Lowry, Michael, Person, Suzette, and Pape, Mark. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ISSTA '08, ACM, pp. 15–26.

[20] Schwartz, Edward J., Avgerinos, Thanassis, and Brumley, David. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and*

Privacy (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 317–331.

[21] Sen, Koushik, and Agha, Gul. Cute and jCute: Concolic unit testing and explicit path model-checking tools. In *Proc. International Conference on Computer Aided Verification (CAV)* (Aug. 2006), Springer, pp. 419–423.

[22] Sen, Koushik, Marinov, Darko, and Agha, Gul. CUTE: a concolic unit testing engine for C. In *Proc. 13th International Symposium on Foundations of Software Engineering (FSE)* (Sept. 2005), ACM, pp. 263–272.

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

[24] Tillmann, Nikolai, and de Halleux, Jonathan. Pex – white box test generation for .NET. In *Proc. Second International Conference on Tests and Proofs (TAP)* (Apr. 2008), Springer. To appear.

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

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

[27] Xiao, Xusheng, Xie, Tao, Tillmann, Nikolai, and de Halleux, Jonathan. Precise identification of problems for structural test generation. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 611–620.