Improving Dynamic Invariant Saliency with Static Dataflow Analysis

Daniel Ellsworth
dellswor@cs.uoregon.edu

July 29, 2013

Abstract

Saliency of invariants reported by dynamic detection techniques tend to be poor. We present a prototype intra-procedural static analysis and invariant filtering system that improves reported invariant saliency by applying a data flow based admission criteria. While successful at reducing the number of nonsensical invariants reported, the current prototype is overly aggressive due to the limitations of intra-procedural data flow. Extension to inter-procedural analysis is non-trivial. Some of the challenges are discussed.

1 Introduction

Dynamic invariant detection, introduced by Ernst et. al.[1], produces expressions that are true on all observed runs of a program at particular points in execution. One of the expected uses of dynamic invariant detection is improving developer understanding of a code base in real-life settings. We believe that a key barrier to this use of dynamic invariant detection is the glut of extraneous, true but unrelated, invariants reported. For even simple functions in object oriented code, dynamic invariant detection often returns a lengthy list of potential invariants. A Java codebase we analyzed contained 675 lines of code (LOC) and produced over 1200 invariants, most of which were misleading or irrelevant. The unfavorable balance of invariants to lines of program text, paired with the low quality of the majority of invariants, are likely contributors to the poor adoption of dynamic invariant detection for program understanding.

A poor understanding of the existing codebase increases the chance that errors will be introduced when making modifications. When patching or enhancing an existing system, there is a risk that the new code will violate a property that other parts of the code depend upon. Change requests and bug reports are based on behaviors at the system interface, which is the usability facade wrapped around the complexities of the underlying system. Under the best circumstances the system will be well designed with abstractions and a clear separation of duties that are consistent with the needs of the change request and the documentation will clearly and accurately represent the codebase. In practice, documentation is regularly deprioritized by the customer in favor of earlier delivery or some other project.

Change requests and bug reports tend to flow from differences between the users' cognitive model for a task and the model used for software implementation. Through a parade of patches and incremental changes by different developers the architecture degrades to a big ball of mud[2]. Even in cases where an architecture appears to be intact and documentation is present, there is the nagging concern that some important detail or constraint required for correct operation was not captured. Salient, programmatically detected, invariants may help to address this gap.

Invariants are one mechanism to summarize the behavior of a region of code. A well formed invariant is a statement of the relationship of values within the program that is true each time a particular location is reached during execution. To safely make a change, the change must not invalidate the invariants required by the callers or callees. If an invariant is violated, then further analysis will be required to assess the changes needed in the other impacted regions. In systems for which invariants are poorly or not documented, the lack of information leaves later programmers little choice aside from guessing the relevant properties and manually assessing impact after making a code modification.

This work contributes a prototype implementation of a new invariant filtration technique, preliminary results using the technique, and a framework for integration of analysis tools. Our filter combines a static data flow analysis with dynamic invariant detection to suppress invariants that are unlikely to be intentionally maintained by a function. In the section 2 we present some background on analysis techniques. Section 3 presents an overview of our approach to filtering out low value invariants and section 4 discusses our implementation. In section 5 we discuss our results applying the filter. We conclude with some thoughts regarding next steps in section 6.

2 Background

Software analysis is one mechanism to aide in understanding a program's runtime behavior as well as the effects of changes. Analysis techniques have classically been binned into two categories, static or dynamic. Static analysis uses program text to prove or extract features of a program's runtime behavior. Dynamic analysis monitors a running instance of a program to record observed runtime behavior and infer likely properties based on the recorded data. Diduce [3] uses this approach to support detection of runtime anomalies. In the training mode Diduce maintains and silently modifies a list of invariants based on values observed. In checking mode Diduce generates warnings when invariants are violated. In hybrid tools, static and dynamic analysis are both used to to produce the final analysis result. Static analysis may be used to focus where and how the dynamic analysis is applied, is in Saner [4], or dynamic analysis may be used to direct static analysis, as in TamiFlex [5]. Additional hybrid tools, like CLARA[6], use static analysis to reduce the amount of dynamic instrumentation without compromising soundness of the runtime instrumentation. Work has also been done on a dynamic technique referred to as experimental analysis [7][8]. In experimental analysis, the analysis tool modifies the software or data between automated executions to aide in identifying and understanding interesting regions.

The most ambitious goal for static analysis is automating formal verification of arbitrary properties for arbitrary software. This lofty goal is unattainable [9]. Producing a result in finite time requires the introduction of abstractions to address points where multiple execution paths occur, i.e. loops, branches, and recursion [10]. Typical abstractions produce sound results that a given property *must* or *may* occur at runtime. Soundness in the analysis guarantees that no possible execution can contradict the analysis. Completeness in the analysis guarantees that a property actually can occur at runtime. Completeness is constrained by how well the abstraction models the underlying system, the precision of the abstraction.The most precise abstraction would be tight loop or branch invariants impacting the analysis property. However, static invariant detection has been elusive and developers are unlikely to reliably generate such documentation manually.

Dynamic analysis avoids the precision loss of static analysis by using the actual execution path of the running program. Monitoring is often done via instrumentation of the software and increases runtime. The results of dynamic analysis are complete, the property was observed on an actual run, but soundness is lost since only observed executions are considered. For many practical development activities loss of soundness in the analysis is acceptable due to the similarity of most anticipated executions.

Hamlet discusses how Daikon invariants differ from invariants according to Floyd/Hoare proof theory[11]. Floyd/Hoare invariants are first order formula that if true before execution are true after execution. Using this definition an invariant can't be invalidated, it wasn't ever actually an invariant. Ernst is careful in his early dynamic invariant detection work to refer to dynamically detected invariants as "likely invariants". Other work in the dynamic invariant detection community, including this paper, have been less careful to maintain the linguistic distinction. Daikon's invariants capture the observed relationship between variables at function entrance and exit. Likely invariants are insufficient for formal proofs of correctness but may be helpful the construction of proofs of correctness.

Whole program analysis using static and dynamic techniques may not be practical due to time and memory constraints. Techniques such as summarization [12] allow a pre-computed summary to be used where recomputation would otherwise be required. Component-Level analysis produces summaries at the granularity of module boundaries to express whole program analysis as a composition of component results [13]. For tools implementing whole program analysis, modularity in the analysis is desirable due to the reduction in resources required to complete analysis between code iterations.

2.1 Static Dataflow Analysis

Static data flow analysis is well studied and has long had application to compile time optimizations. In order to safely reorder instructions there must be a guarantee that control flow and data flow side effects match with the original execution order. Data flow refers to how the variable values are related during execution. Control flow refers to the order in which instructions are executed. Control flow and data flow are closely related since the control flow determines which assignments are made and data flow determines the effects of branches.

A common transformation to simplify reasoning about data flow is known as single static assignment (SSA). In SSA form each variable takes on only one value. In many programs, a variable V will take on many values. SSA introduces a unique name, such as V_x, for each location where the value of V may be changed. At points in the code where control flow merges and two unique names for the same variable in the source text are present, such as V_1 and V_2, SSA will add another unique name to represent the value V_1 or V_2. Cytron describes how to, given program text, efficiently compute an SSA form[14].

The full control flow graph represents all paths through a program. Many times a programmer is interested in only the subset of paths having to do with a feature being implemented or debugged. Program slicing is a technique for selecting only the part of the control flow graph related to a specific instruction and discarding the rest. Efficient computation of slices is well studied[15].

Most static data flow analyses make use of abstract and symbolic execution. The program state is modeled as a set of symbolic formulas that capture the range of values that might occur. During the analysis each instruction is abstractly executed, the effects of the concrete instruction are applied to the symbolic representation based on a model of how the concrete instruction effects the state of the property being analyzed. Loops are typically handled by abstract execution of the loop until a fixpoint is reached.

As object oriented programming has become more prevalent, analysis tools that are unable to handle indirection become less useful. In procedural languages, there is little indirection. Cytron alludes to the difficulty of handling arrays, structs, and heap memory when discussing the conversion from program text to SSA[14]. Virtual function dispatch complicates static construction of a sound and complete call graph since the concrete function called depends on the current heap instance referenced by a variable at execution time. Bodden sidesteps this complexity by monitoring runtime types and feeding the results back into SOOT's static analysis[5].

2.2 Dynamic Invariant Detection

Dynamic invariant detection uses invariant templates and runtime observations to attempt to determine likely invariants at instrumented program points[1]. A target application is instrumented and run to generate a trace, capturing the values of all visible variables at particular program points. The variables at a program point are then arranged into expressions that can be tested, and the dynamic invariant detector tries the candidate invariant with all values observed. Candidate invariants that are not falsified by data in the trace are likely invariants.

Daikon provides a few optimizations to improve upon the basic strategy[16]. The basic strategy results in many invariants to test at each program point, the majority of which will be untrue. Daikon improves runtime performance by removing invariants from consideration after the first observed contradiction. Some invariants may not be well supported due to few observations. Daikon employs some analysis to produce a confidence metric based on the number of times the invariant was evaluated and suppresses invariants beneath a confidence threshold. For program points where one or more variables are observed to always have the same value, Daikon generates invariants using only one of the variables. Additionally, Daikon suppresses invariants for a program point that can be logically inferred from other invariants reported for the point.

Kuzmina extends Daikon to consider polymorphism. Kuzmina's extension, Turnip[17], can provide more precise post-condition invariants on methods where different subtypes, with different method implementations, are used. Balint[18] improves upon Turnip by mapping invariant names to those in a JML specification for the software under analysis. In DySy, Csallner uses symbolic execution during concrete execution to determine a program's likely invariants[19]. DySy invariants are built by tracking branches taken to reach instrumented program points and abstracting the values controlling the branches. These systems improve the quality of returned invariants by more carefully constructing invariants than Daikon's default approach and require significant integration with the internal behavior of the invariant detector. In contrast, our tool operates as a post processing step and requires no changes to the detection process.

DynComp is the most closely related tool we are aware of [20] and can be used with our filter. DynComp uses dynamic analysis to infer abstract types; these can be thought of as extending base types with some dimension (eg. dollars, miles). The dimensions can be used to constrain the invariant templates Daikon instantiates to templates where all variables have matching dimensions. DynComp applies a dynamic data flow analysis, by tagging and monitoring values in the runtime, to generate the comparability classes to feed into the invariant generation process.

Our work differs from DynComp in a few significant ways.

- DynComp reduces the amount of work done by Daikon by removing invariant templates from consideration before their first application. Our tool does not become involved until after Daikon has completed invariant computation.

- DynComp uses dynamic analysis, requiring the test suite be run once to generate the comparability classes before Daikon can be run. Our data flow analysis can be run in parallel with Daikon.

- DynComp is a path sensitive whole program analysis that produces a flow insensitive result. Variables determined to have the same abstract type at any time during the observed execution will be allowed together in Daikon templates at all program points. Our analysis is intra-procedural and sensitive to data flow within functions.

2.3 Hybrid Filtering

Our proposed filter uses static analysis to filter the output of dynamic analysis; we are aware of work that uses dynamic analysis to filter static analysis results. FindBugs analysis is different from many of the other static analysis in the literature; FindBugs does not attempt to be sound or precise [21]. The FindBugs tool analyzes source code and reports on statically detected patterns in the program text that may indicate unsafe usage or non-conformance with language conventions. FindBugs does not make any attempt to prove that the detected pattern may actually leads to incorrect execution, so it may over report the defects present. FindBugs has become an integrated part of Google's business processes to improve overall software quality[22]. More recent work has looked at adding residual testing to help filter extraneous FindBugs output based on observed executions[23], using dynamic analysis to filter static analysis.

3 Approach

Recall that Daikon produces likely invariants at method entrance and exit points (pre- and post- conditions for execution of the function) based on the set of all visible variables at the instrumented points. In object oriented programs the visible set includes all fields of argument variables. Our work is motivated by the observation that a large number of extraneous post-condition invariants included variables that were never used in the function's control or data flow. Such invariants are very unlikely to be salient. Further, invariants with this property are easy for a programmer to identify as unlikely and as such have a larger negative impact on user trust of the invariants reported than more difficult to invalidate invariants [24].

Variables that are never referenced or set within a region of code are unlikely to be affected by that region. Clearly, if a particular variable and its aliases are never used within a function, the variable neither impacts or is impacted by the execution of the function. The condition in our prototype is slightly stronger: Invariants involving variables in fully independent slices, slices for which there is no intersection, represent isolated and separable computations. We state our filter criteria as: admissible invariants are only those where all variables participating in the invariant share at least one common ancestor in the variable dependency graph. By limiting the reported invariants to only those matching our filter criteria, we increase the saliency of the invariants reported to the user.

Since Daikon invariants are produced at the granularity of functions, intra-procedural analysis seems appropriate. Each pre- or post-condition invariant reported by Daikon expresses the behavior of a function independent of all other functions based on observed values. Intra-procedural static analysis confines the abstract and symbolic execution to the facts available within a single function, which appears to match the level of isolation provided by Daikon invariants. Our filtering criteria also seems to naturally fit at the level of intra-procedural analysis; only variables used in the function should be allowed to participate in invariants. As the project progressed we realized intra-procedural analysis is insufficient and will discuss why our intuition fails in subsection 4.4.

To align with the Daikon output, the filter must be able to map from the variable names used by Daikon to their data and control dependence from function exits towards the function entrance. We independently invented a variable dependence graph (VDG) representation very close to that discussed by Jackson[25] to capture the relations between variables during abstract execution. Each variable is represented as a node with edges to the variables that contributed to the current value. The debugging symbols within the java bytecode allow mapping between Daikon variable names and the variables in the VDG at function entrance and exit.

4 Implementation

In the remainder of this paper we present our prototype invariant filter. The filter is a hybrid analysis built atop Daikon and a static data flow analysis implemented in our Relational Abstraction For Targeted Exploration of Code (RAFTEC). Daikon is the leading academic dynamic invariant detector with more than a decade of active development and research. Daikon has also inspired Agitator, a commercial product supporting testing activities [26]. RAFTEC was built as a framework for assisting in the development of modular integrated software analysis[27]. The design is informed by extract, load, transform (ELT) systems used in industry to support business intelligence.

In RAFTEC intermediate and final representations are kept within a relational database. The relational database provides a natural integration point to incrementally add additional analysis and some capabilities to easily record relations between processing phases as the representation is processed. Modularity in analysis can also be provided for low cost as tables and schemas provide a transparent boundaries between analysis tools and their underlying intermediate representations. The relational database also provides a concise interface for the developer to query for answers, audit results, and use results from other analysis.

RDBMS have been rejected in the 1980s and 1990s for program analysis due to poor performance[28]. However there is a growing body of work relying on datalog as a core technology for static analysis. In 2005, Lam published work on context sensitive analysis using datalog[29]. In 2011, Smaragdakis published work on a high performance points to analysis for Java atop datalog[30]. Ali presents an application-only call graph construction, using datalog, that uses less time and space than Smaragdakis work[31].

We elected to use an RDBMS over datalog primarily due to familiarity and the diversity of tools for working with SQL data stores. One of the concerns we've seen reported in other work is SOOT memory consumption when performing analysis since all of the data must be keep in memory for the duration of the run[32]. Using an RDBMS avoids this limitation by allowing the RDBMS to manage efficient caching to disk during execution. The relational database provided some additional advantages during development for auditing the implementation. Queries to match records between transformation source and sinks are relatively easy to write and provide a programmatic way to verify that transformation coverage matches with expectations.

RAFTEC, for analyzing modestly sized programs, was reasonably performant on low-end consumer grade desktop without performance tuning. At one point, we loaded the complete Java 1.6 standard library (10s of billions of control flow nodes and edges) in roughly a week on low-end hardware; the majority of time was spent waiting on IO due to the slow storage. Whole program transformations, aside from a non-terminating recursive query used for line numbering, were reasonably performant on the hardware configuration. Environmental improvements such as access to high speed storage, migration to a threaded RDBMS, access to kernel memory configuration, and bulk rather than row insertion are all changes we would expect to significantly improve RAFTEC performance.

Three RAFTEC components were built to support the work in this paper: Java Bytecode Recorder (JBR), Intraprocedural Variable Dependency Grapher (IVDG), and Daikon Invariant Recorder (DIR). JBR and DIR handle intake from the base data sources, java class files and daikon invariant files respectively. IVDG performs abstract execution against the data from the JBR to generate the VDG needed to filter the Daikon invariants.

4.1 Java Bytecode Recorder

The Java Bytecode Reader is responsible for decompilation and loading the target class file into the RAFTEC database. Consistent with the ELT philosophy, the data is loaded with minimal transformation. Once the data is loaded, stored procedures are used within the database to complete construction of the control flow graph and various simple analyses and produce derived facts. An entity relationship diagram for the JBR tables is shown in figure 4.1.

Bytecode decompilation is performed using ASM. The information provided is loaded with minimal modification into the backing datastore. ASM was selected since it appeared to be the only high quality free Java disassembler under active development. BCEL and Jasmin appeared to have been effectively abandoned based on age of the latest releases at the time we made the decompiler selection. Two APIs are provided by ASM for accessing the byte code: a tree based representation that supports queries against the byte code and a streaming visitor based API. The ASM documentation recommends using the visitor based API, so our solution does so.

As each byte code entity is visited, it is written to the backing database. Initially only an ASM InstructionAdapter was implemented, this is the visitor responsible for byte code instructions. Our ASM InstructionAdapter

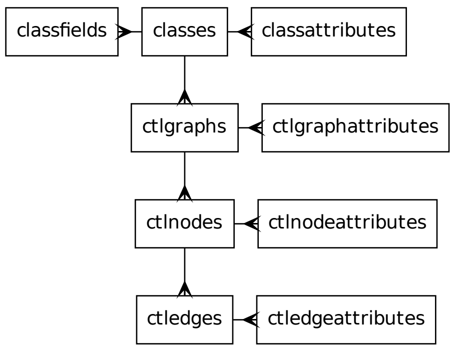

Figure 4.1: Entity Relationship Diagram for the JBR tables

(MethodControlFlowVisitor) stores the details of each byte code instruction to the RAFTEC datastore on visit. Later an ASM ClassVisitor (ClassControlFlowVisitor) was implemented to support direct collection of class member fields and computation of class hierarchies within the RAFTEC. The tables packed by the respective visitors are listed in table 4.1.

After the raw data is loaded, a series of transformations and basic analyses are executed. First the data is cleaned by transforming the ASM metadata into node attributes. ASM Metadata includes jump target labels, variable declarations, and source code line numbers. Following cleaning, the edges of the control flow graph are constructed by linking jumps and removing unused serial edges. At this point the data is considered scrubbed and the basic analysis begins. Basic analysis includes adding an over-approximation of the call graph based on variable types and the class hierarchy, performing classification of some basic method properties, construction of the intra-procedural pre- and post-dominator sets, as well as computing the intra-procedural control dependency.

JBR forgoes converting the raw control flow graph into one based on basic blocks. This is consistent with Fosdick and Osterweil's observation that "...it is not clear that a significant advantage can be obtained by this initial preprocessing of basic blocks and reduction of the graph." [33]. For abstract execution every raw node will need to be visited, effectively bypassing any value the basic blocks might provide for automated analysis. Construction of a basic block representation within RAFTEC should be trivial and could

Vistor	Tables
MethodControlFlowVistior	classfields
	classes
	classattributes
ClassControlFlowVisitor	ctlgraphs
	ctlgraphattributes
	ctlnodes
	ctlnodeattributes
	ctledges
	ctledgeattributes

Table 4.1: Table showing the visitor responsible for packing the tables.

likely be handled efficiently by introducing a new table relating the *ctlnode* rows for block entrance and exit nodes to a unique block number.

Intra-procedural control dependence is computed using dominator sets. Goergiadis presents several efficient methods for computation of dominators[34]. We selected a simple iterative algorithm, based on the formal definition, for ease of expression in SQL. Control dependence is used by IVDG to add in variable dependencies that are the result of control flow.

Basic method property classification is done because it was cheap to implement and allows some additional insight into the qualities of our target code bases. The classification adds attributes to the each method in RAFTEC: Does the method accesses the heap, make any method calls, contain loops, call itself directly, or call itself indirectly. Indirect recursion uses the call graph produced using variable type at call sites and the possible method implementations given the captured class hierarchy, potentially producing an over approximation. Loop and indirect recursion is done using Tarjan's algorithm for detecting strongly connected components in a directed graph since we were unable to find a suitable algorithm that could take advantage of potential parallelism available in our representation[35].

4.2 Intraprocedural Variable Dependency Grapher

The Intra-procedural Variable Dependency Grapher is responsible for computation of the statically determined variable dependency graph. Computation of the VDG is done by abstract execution of each method contained

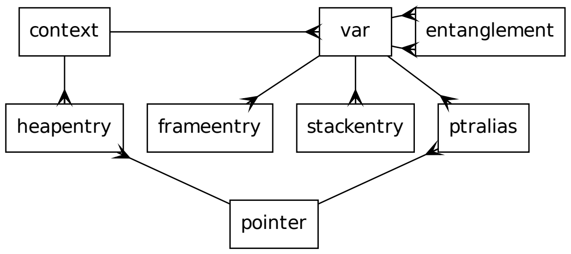

Figure 4.2: Entity Relationship Diagram for the IVDG tables

in RAFTEC. At each step of the abstract execution information is added to the symbolic representation regarding data dependence. Abstract execution is handled by Java code while the symbolic representation is stored and manipulated in the database. The IVDG entity relationship diagram is shown in figure 4.2. Once the graph has been generated, determining that one variable is dependent on another can done by searching for a path connecting the variables.

Our construction algorithm executes each instruction only once. Each instruction in the control flow graph has two associated contexts: A context immediately prior to instruction execution, in-context, and a context immediately following execution, out-context. Each context contains all of the live variables in the context as individual nodes. Data dependency edges are added between variable nodes in the in-context and variable nodes in the out-context based on the effect of the instruction. Data dependences between instructions are handled by connecting variables in the same locations from the preceding instruction's out-context to the current instruction's in-context via a stored procedure. The strategy avoids needing to handle branch and join points in the control flow as special cases. Conditional branch instructions have an attribute added to the control node identifying the variables from the in context used to make the branch decision; the attribute is used in post-processing to capture control dependencies in the VDG. Our VDG is very similar to one described by Jackson[25].

Abstract execution models the operand stack, stack frame, and heap as they evolve during method execution. The size and composition of these regions of memory on instruction exit are propagated to the entrance of the immediately following functions. The operand stack and stack frame are modeled as lists. When merging an out-context to an in-context, variables at corresponding list indices are marked as dependent. Abstract execution of multiple methods could be done in parallel, but that feature has not been implemented. We can also envision the ability to perform abstract execution in parallel at the instruction level since the operand stack and stack frame composition are statically known and provided in the byte code.

Storing complete memory context for each instruction has the potential to consume a large amount of memory. Each context is used few times during abstract execution, keeping all contexts in main memory throughout abstract execution is unnecessary. Designing a performant strategy for writing and retrieving contexts from disk within our code would be difficult and might introduce errors. Instead, we use the RDBMS a bit like virtual memory. All of the contexts are stored in the database and, as instructions are executed, the database is queried for the correct context. We take advantage of optimizations within the database to handle when and how data is written to disk as well as how and when to efficiently retrieve data from disk. Storing all of the contexts in the database provided additional benefits during implementation since they provided a complete and easily queried log for the effects of the abstract execution code.

4.3 Daikon Invariant Recorder

The Daikon Invariant Recorder (DIR) is primarily responsible for reading Daikon invariant files and loading the contents into the appropriate tables. Like the JBR, invariants are written to the datastore as they are read from the serialized invariant file produced by Daikon. DIR uses Daikon's file reader class to read the invariants file from disk and then parses and stores the invariant data in the RAFTEC database. An entity relationship diagram for the DIR tables is shown in figure 4.3. Mapping between variables in the IVDG and Daikon will also be discussed in this section.

Invariants provided by Daikon are stored in the database with very little processing. The API provided by Daikon's invariant file reader provides invariants by program point, where each program point has some number of associated invariants. The program points are recorded in RAFTEC with their respective class and method strings. The API for invariant objects in Daikon does not provide a list of variable-like objects to work with, but it does expose a string representation of all of the Daikon variable names. DIR

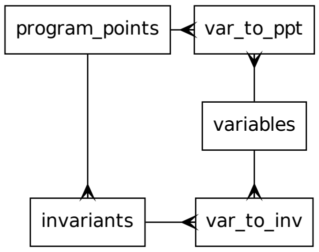

Figure 4.3: Entity Relationship Diagram for the DIR tables

parses that string when recording the invariant and the variables used by the invariant.

Matching between Daikon and IVDG is non-trivial. The strings Daikon uses when expressing method signatures and variable names are just enough different to make directly matching with the ASM strings unreliable. Additional complications come from the meta variables Daikon uses, such as variables for array size and original value. Stored procedures are used to normalize the method names and signatures when building the view to map Daikon program points to instructions. Daikon variables must be interpreted based on the program point in which they occur and the composition of the stack frame at the point. The interpretation is done via stored procedure during construction of the final filter reporting table. The ability to use the RDBMS to quickly audit that all program points and variables were matched as well as the ability to query for additional details regarding those that were unmatched was extremely helpful when working through mapping between the representations.

4.4 Analysis Limitations

The call graph currently stored in RAFTEC is a large over estimation. Handling virtual dispatch in a static analysis tool is difficult, since the runtime types that might inhabit a variable must be known. For most non-trivial programs, heap objects are passed as parameters. To determine the type of an object at a call site, so that a precise call graph can be computed, requires a precise call graph to determine the types of parameters. Emami et. al.

mention this recursive problem when discussing an static inter-procedural points-to technique[36] that does not handle function pointers in the heap. Lattner et. al. present a solution involving indirection via the heap based on producing and efficiently updating partial solutions[37].

Generics in Java provide some extra resolution regarding the return type of methods. Type erasure during compilation removes this information from the byte code, so it is unavailable to our tool to aid in analysis. Generics would help in call graph construction and improving variable typing; changing JBR to augment the ASM output using source code analysis (when available) would likely improve the possible results.

Reflection is impossible to support in general during static analysis since the runtime behavior is not defined until runtime. There are some uses of reflection, such as calls with static strings, that may be possible to support during static analysis. We have made no attempt to support reflection with our tool.

Exceptions are not currently represented in the control flow graph and therefore not considered during abstract execution. Additional control flow edges could be introduced from instructions raising exceptions to the first instruction handling the exception. Introduction of these edges becomes difficult without a precise call graph since an exception thrown in one method may be caught by a caller several stack frames earlier. If exception edges were added, the abstract execution code would need to be modified to handle jumping between operand stack and stack frame compositions.

Variables referencing objects in the heap carry a list of pointers, representing the objects the variable may point to based on the control flow abstractly executed. While the pointer lists are merged at join points, newly added pointers are not propagated along already executed paths. The effect of instructions following the join point may not be applied to the newly added instance. Instance variables for objects created or assigned within a method may not have accurate dependencies. Only the heap pointers carried by the instance variable at the time of abstract instruction execution will be present in the generated dependencies. One strategy to address the missing dependencies would be to continue execution with the updated pointer list until a fixpoint is reached regarding variable dependencies.

Abstract execution of cast instructions is not very robust. Casts are treated as a function call that returns an object of the cast type. We would prefer a strategy that would better protect the data dependencies of instance fields across during down casts and map back to the original instance during upcasts.

Debugging symbols are currently required in the Java byte code. Enabling the debugging symbols during compilation is simple, only a single flag is required, and avoids needing to compute gen and kill sets during abstract execution. Debugging symbols also help improve the readability of Daikon invariants.

Object construction is handled as two separate byte code instructions in the Java virtual machine (JVM). The first instruction allocates the memory space for the instance and the second calls the constructor. IVDG processes the JVM byte code looking at only a single byte code instruction at a time. Since the existing analysis is intra-procedural only, the parameters sent to the initializer do not become dependencies for the variables within the instance. A quick way to address this deficiency would be to handle initializers differently than other method calls. However, other mutator functions would retain their existing behavior. We have decided to leave this unaddressed until we have a strategy for handling the heap during inter-procedural analysis. solving the issue for both initializers and mutators.

Object instances may represent recursive data structures. When object instances are spawned during abstract execution, only the object instance and its member variables are initialized. Daikon invariants do not go deeper than one level, so this is satisfactory for our invariant filter. For inter-procedural analysis, deeper modeling of object instances may be required.

Class instances and object instances are kept separately in IVDG; the class instances end up as anonymous heap objects in the database representation. The Daikon invariant representation does not distinguish between class and instance variables. Adherence to the Java naming conventions is the extent of the differentiation in the naming between class and instance variables. Due to the potential ambiguity, the filter does not attempt to handle invariants involving class variables.

5 Evaluation

The filter performance was evaluated by a manual review of the invariants returned by Daikon, Daikon with DynComp, and Daikon with our filter. Daikon and Daikon with DynComp executions used the default settings. One of Daikon's optimizations maintains sets of equivalent variables and reports invariants using only one member of the equivalency set. Since our

```
public int getBlasterDamage() {
    return rand.nextInt(maxBlaster);
}
```

Figure 5.1: Code listing for getBlasterDamage

Invariant
this.maxhp > return
this.turnsToZombie < return
Alien.rand == orig(Alien.rand)
this.name == orig(this.name)
this.name.toString == orig(this.name.toString)
this.maxhp == orig(this.maxhp)
this.maxhp == orig(this.curhp)
this.maxhp == orig(this.maxBlaster)
this.turnsToZombie == orig(this.turnsToZombie)

Table 5.1: Invariants reported by Daikon with leader election for getBlaster-Damage().

filter needs the specific variable name to map correctly, we disabled this optimization when generating invariants for use with our filter. Initially we had planned to use invariant counts as our primary evaluation criteria. While writing this paper we discovered a behavior, that we believe to be an error in Daikon, making raw invariant counts incomparable.

In some cases, many variables may have the same value each time a program point is visited. Such instances can result in numerous similar invariants based on permutations of the variable names. By default, Daikon selects only one of the variables (the leader) to represent all of the variables having equal value when generating invariants. Since our filter uses the variable names within the invariant to map to nodes in the variable dependency graph, a poorly selected leader can result in missing a salient invariant.

Figure 5.1 and table 5.1 show a sample of code and the invariants reported with leader election. One of the invariants that we would expect to see relates the return value of *getBlasterDamage* to the value of *maxBlaster* since *maxBlaster* sets the upper bound for the call to *nextInt*. The invariant provided by Daikon relates the return value to *maxhp*, which is never refer-

Class	Methods	with loops	with branches	accesses heap	makes calls	recursive
tcas	11	0	3	10	5	0
AlienAI	3	1	1	2	3	0
Alien	8	0	6	8	4	0
Point	10	1	7	9	6	0
Wall	1	0	1	0	1	0
DirectedMoveAction	4	0	4	2	2	0
Simulation	5	3	2	2	4	0
SimulationBoard	16	8	2	14	16	0
SimulationAction	1	0	1	0	1	0
EmptySpace	1	0	1	0	1	0
RandomMoveAction	4	0	4	2	2	0
BiteAction	4	0	4	2	2	0
Zombie	7	0	6	6	5	0
ShootAction	4	0	4	2	2	0
ZombieAI	7	5	1	2	7	1
SimulationObject	6	0	6	2	1	0

Table 5.2: Properties of the methods for each class detected by the JBR analysis. JTCAS contains only one class: tcas. All other classes listed are part of ZAS.

enced in the method. Using the shown Daikon output, our filter will not be able to identify any salient invariants for *getBlasterDamage*.

We applied our static analysis and filtering technique to two sample applications; JTCAS and Zombies vs Aliens Simulator (ZAS). JTCAS is a commonly cited program from the Sieman's Suite[38] for exploring software analysis[39][40]. ZAS was written to support labs for first year computer science majors; the program exercises inheritance in addition to loops, objects, and recursive features of Java. After producing our preliminary results with JTCAS, we realized that the JTCAS code is not representative of most programs. Table 5.2 presents the classes and count of methods within each class exhibiting features RAFTEC classifies. The discussion in this section will focus primarily on analysis of ZAS due to the limited range of language features exercised by JTCAS.

Before beginning work on the prototype presented in this paper, we conducted preliminary investigation on our intended technique. JTCAS was the

	LOC	Daikon	DynComp	With out Leader
ZAS	675	1042/32	1171/34	1336/22
JTCAS	155	853/0	442/0	6414/0

Table 5.3: Count of invariants reported by Daikon, Daikon+DynComp, Daikon without leader election. The raw invariant count is followed by the number of invariants admitted by our filter. Daikon without the leader election optimization generated 1336 invariants, only 25 of which were admitted by our filter.

target of investigation during our preliminary work due to its use in other related work we were reviewing at the time. Our preliminary filter was built by extending a simple inter-procedural static taint checker, from a course project, to interact with a Daikon invariant file. The base code did not support any information stored in the heap and we extended the code to handle object fields by adding them just in time to the symbolic representation. Early results looked promising with respect to the number of invariants filtered though, on manual review, there was some question as to wether any of the invariants produced by Daikon or admitted by our filter for JTCAS were useful to program understanding. The code was unable to be applied reliably to other code bases we tried. Upon investigation we discovered that the preliminary implementation could not handle recursion or loops due to the heap model used and was therefore unsuitable for analyzing the majority of interesting Java applications.

To generate the invariants for JTCAS, we collected traces for all 1608 test cases provided for Daikon, Daikon+DynComp and Daikon-equiv+RAFTEC; generation of the trace data and invariants took roughly 3 hours for each target. To generate the invariants for ZAS, we collected trace data from a single execution of the simulator per target. Trace data and invariant generation for ZAS took roughly 6 hours. Once the datasets were produced by Daikon, we loaded the related byte code, performed the static analysis, loaded the invariants, then generated the report; these steps were done in serial and took roughly 20 minutes. The time required to identify salient invariants with our technique is clearly dominated by the time spent on the dynamic analysis.

Of the 1336 invariants detected with leader election disabled, 41 were class or object invariants. Our filter is unable to process class and object invariants because they are inter-procedural in nature and do not map to method entrances or exits. Of the remaining 1295 invariants, 336 were entrance invariants which are filtered out because there is no intra-procedural data flow available at method entrance to use as a basis for admission and 47 were not evaluated due to a gap mapping constructor method names between ASM and Daikon. Upon manual inspection of the 912 invariants that are within our filter's capability, the filter results match with expectations given the filter limitations and program text.

In manually reviewing the invariants, some invariants were identified that might have been useful but are admissible. An example of such invariants can be seen with the *directionTowardsInv* method shown in figure 5.2. The method returns an integer value based on the relation between *o.x* and *this.x*. There is no control or data dependence between *o.x* and *this.x* so these invariants are dropped. If the invariants relating *o.x* and *this.x* were present with the constant return values, it would be possible to determine the directional meaning of each of the return values. Admitting all invariants involving variables in involved in the control dependence for the return instruction may be able to capture these invariants. Sadly, the invariant reporting the constant return value was also dropped by the filter. Dropping constant returns is a side effect of the return variable not having data dependencies on any other variable within the method, the return variable is indistinguishable from an unused variable when traversing the variable dependency graph. Keeping constant return invariants could be handled in the future as a special case.

5.1 Daikon Incomparability

The results presented in table 5.3 are surprising. DynComp is a preprocessing filter that constrains the variables that may appear together in detected invariants. One would expect that reducing the pool of instantiated invariant templates would reduce the number of reported invariants; when the opposite appears to have happened. Additionally, turning off leader should generate a strict super set of the invariants generated with leader election turned on. Every invariant reported with leader election should produce one or more invariants without leader election; each invariant containing the leader can be written as an invariant containing another variable in the set. When Daikon without leader election is used our filter admits less invariants than when Daikon is used with the default configuration.

The RAFTEC database was used to investigate the results of Daikon with and without leader election; we ran out of time to investigate the DynComp

```
public int directionTowards(Point o) {
                if(x>o.getX()) {
                        // 5 6 7
                        if(y<o.getY()) {
                                return 7;
                        }
                        if(y>o.getY()) {
                                return 5;
                        }
                        return 6;
                }
                if(x<o.getX()) {
                        // 1 2 3
                        if(y<o.getY()) {
                                return 1;
                        }
                        if(y>o.getY()) {
                                return 3;
                        }
                        return 2;
                }
                // 0 4
                if(y<o.getY()) {
                        return 0;
                }
                return 4;
        }
```

Figure 5.2: Code listing for directionTowards

line	invariant
49	orig(this.y) < orig(o.y)
49	orig(this.x) < orig(o.x)
49	return == 1
54	orig(this.x) < orig(o.x)
54	orig(this.y) == orig(o.y)
54	return == 2
39	orig(this.y) < orig(o.y)
39	orig(this.x) > orig(o.x)
39	return == 7
52	orig(this.y) > orig(o.y)
52	orig(this.x) < orig(o.x)
52	return == 3
42	orig(this.x) > orig(o.x)
42	orig(this.y) > orig(o.y)
42	return == 5
58	orig(this.x) == orig(o.x)
58	orig(this.y) < orig(o.y)
58	return == 0
44	orig(this.x) > orig(o.x)
44	orig(this.y) == orig(o.y)
44	return == 6
60	orig(this.x) == orig(o.x)
60	orig(this.y) > orig(o.y)
60	return == 4

Table 5.4: Desirable invariants for *directionTowards* that Daikon produced that were filtered out.

invariant discrepancy. Having the data in the relational database made comparison much easier than working directly with Daikon's textual output. Our first step was an inner join to verify that the 1042 invariants generated with leader election were a strict a subset of the 1336 invariants generated without leader election. We discovered only 678 invariants were present in both data sets; 364 invariants are generated with leader election that are not generated without leader election. Using a left join we were able to quickly compute the missing and matched invariants for inspection. Missing invariants are all for exit and sub exit program points. We suspect that there is a defect in Daikon having to do with the interaction between exit invariant generation and the equivalence optimization that causes invariants to be missed when the equivalence optimization is turned off.

6 Future Work

One of our early realizations as we looked through the filter results in the context of the programs was the failure to account for delegated computations. This is a common pattern for testing nontrivial conditions. A function is made to encapsulate the condition and then called to cause the correct path to execute. When a call is made, such as a boolean function used as a branch condition, we lose visibility into the fields used to compute the return value and may incorrectly suppress salient invariants. Getter and setter methods similarly hide useful invariants from our filter. Improving heap modeling and adding inter-procedural analysis are important steps for continued work on this invariant filtering technique.

We are not aware of a precise way to extend the technique we are using for VDG construction to correctly handle aliased values. Jackson [25] did not address this issue and we are not aware of other work using a similar representation that address pointers into the heap. Jackson mentions modular incorporation of aliasing as a current challenge but does not suggest in any detail how the challenge might be solved. A simple approach sets the variable node associated with a read to depend on the values of all heap object fields that the read may have targeted by traversing the pointer set. Incorrect dependencies may be collected since the variable resulting from the read becomes a join point for otherwise unrelated variables. D'Silva remarks that dynamically allocated data structures are a challenge when modeling program properties and that none of the tools he surveyed could assert even trivial properties[41].

Inter-procedural analysis for object oriented languages is non-trivial. As mentioned earlier in 4.4, virtual dispatch complicates generation of a precise call graph since the concrete method used often depends on the heap. Uses of design patterns, such as the observer pattern, further obscure the call graph by placing a second layer of indirection in the heap. Horwitz describes inter-procedural dependency graph construction for slicing using conversion to an alias free representation, by cloning functions, to address the challenge of references[15]. We have some reservation regarding applying Horwitz technique to modern object orient code since aliasing is so prevalent and may result in exponential growth in the size of the graph. TAJ use one level of context sensitivity when performing points-to analysis and on-the-fly call graph construction[42]. The strategy used in TAJ may be a reasonable starting point for call graph construction but we ran out of time to implement it.

Indirection is a known issue for static analysis and is at the core of the issues with virtual dispatch. Calling an instance method, in Java, requires the runtime to lookup the implementing method based on the runtime type. Potentially, the matching method of any class that extends or implements the variable type could be the one used at runtime. For instance, a variable of type Iterator has over 500 possible method bodies available from the standard library; clearly the variable type is insufficient. Points-to analysis is a static technique to determine which instances a particular variable may point to at runtime. Sound points-to analysis requires sound dataflow analysis; the recursive challenge here is self evident. Bodden side steps this problem in TamiFlex by using runtime monitoring rather than static techniques to determine the method invoked at runtime[5].

7 Conclusion

Our experience indicates using data dependance as a criterion for invariant saliency is a promising direction for further investigation. For many interesting cases, intra-procedural analysis appears insufficient since called functions hide relevant dependencies. Further work is needed to understand how to expand the scope to include inter-procedural analysis; which may involve solving several other hard problems in static analysis to produce precise results.

Additionally, we have had a positive experience using a relational database to support our analysis. The relational database provided several unexpected gains with respect to development, debugging, and auditing the behavior of our system.

References

[1] Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 21st international conference on Software engineering. ICSE '99. New York, NY, USA: ACM; 1999. p. 213–224. Available from: `http://doi.acm.org/10.1145/302405.302467`.

[2] Foote B, Yoder J. Big ball of mud. Pattern languages of program design. 1997;4:654–692.

[3] Hangal S, Lam MS. Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02. New York, NY, USA: ACM; 2002. p. 291–301. Available from: `http://doi.acm.org/10.1145/581339.581377`.

[4] Balzarotti D, Cova M, Felmetsger V, Jovanovic N, Kirda E, Kruegel C, et al. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: Security and Privacy, 2008. SP 2008. IEEE Symposium on; 2008. p. 387 –401.

[5] Bodden E, Sewe A, Sinschek J, Oueslati H, Mezini M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11. New York, NY, USA: ACM; 2011. p. 241–250. Available from: `http://doi.acm.org/10.1145/1985793.1985827`.

[6] Bodden E, Lam P, Hendren L. Partially Evaluating Finite-State Runtime Monitors Ahead of Time. ACM Trans Program Lang Syst. 2012 Jun;34(2):7:1–7:52. Available from: `http://doi.acm.org/10.1145/2220365.2220366`.

[7] Zeller A. Yesterday, my Program Worked. Today, it Does Not. Why? In: Software Engineering — ESEC/FSE '99. vol. 1687 of Lecture Notes in Computer Science; 1999. p. 253–267.

[8] Ruthruff JR, Elbaum S, Rothermel G. Experimental program analysis: a new program analysis paradigm. In: Proceedings of the 2006 international symposium on Software testing and analysis. ISSTA '06. New York, NY, USA: ACM; 2006. p. 49–60. Available from: `http://doi.acm.org/10.1145/1146238.1146245`.

[9] Turing AM. On Computable Numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society. 1936;42:230–265.

[10] Hantler SL, King JC. An introduction to proving the correctness of programs. ACM Computing Surveys (CSUR). 1976;8(3):331–353.

[11] Hamlet D. Invariants and state in testing and formal methods. SIGSOFT Softw Eng Notes. 2005 Sep;31(1):48–51. Available from: `http://doi.acm.org/10.1145/1108768.1108806`.

[12] Yan D, Xu G, Rountev A. Rethinking Soot for summary-based whole-program analysis. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. SOAP '12. New York, NY, USA: ACM; 2012. p. 9–14. Available from: `http://doi.acm.org/10.1145/2259051.2259053`.

[13] Rountev A. Component-Level Dataflow Analysis. In: Heineman G, Crnkovic I, Schmidt H, Stafford J, Szyperski C, Wallnau K, editors. Component-Based Software Engineering. vol. 3489 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg; 2005. p. 21–23. Available from: `http://dx.doi.org/10.1007/11424529_6`.

[14] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst. 1991 Oct;13(4):451–490. Available from: `http://doi.acm.org/10.1145/115372.115320`.

[15] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS). 1990;12(1):26–60.

[16] Ernst MD, Czeisler A, Griswold WG, Notkin D. Quickly detecting relevant program invariants. In: Proceedings of the 22nd international conference on Software engineering. ICSE '00. New York, NY, USA: ACM; 2000. p. 449–458. Available from: `http://doi.acm.org/10.1145/337180.337240`.

[17] Kuzmina N, Gamboa R. Extending dynamic constraint detection with polymorphic analysis. In: Proceedings of the 5th International Workshop on Dynamic Analysis. IEEE Computer Society; 2007. p. 1.

[18] Balint M, Minea M. Automatic inference of model fields and their representation. In: Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs. FTfJP '11. New York, NY, USA: ACM; 2011. p. 9:1–9:6. Available from: `http://doi.acm.org/10.1145/2076674.2076683`.

[19] Csallner C, Tillmann N, Smaragdakis Y. DySy. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. IEEE; 2008. p. 281–290.

[20] Guo PJ, Perkins JH, McCamant S, Ernst MD. Dynamic inference of abstract types. In: Proceedings of the 2006 international symposium on Software testing and analysis. ISSTA '06. New York, NY, USA: ACM; 2006. p. 255–265. Available from: `http://doi.acm.org/10.1145/1146238.1146268`.

[21] Hovemeyer D, Spacco J, Pugh W. Evaluating and tuning a static analysis to find null pointer bugs. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. PASTE '05. New York, NY, USA: ACM; 2005. p. 13–19. Available from: `http://doi.acm.org/10.1145/1108792.1108798`.

[22] Ayewah N, Pugh W. The Google FindBugs fixit. In: Proceedings of the 19th international symposium on Software testing and analysis. ISSTA '10. New York, NY, USA: ACM; 2010. p. 241–252. Available from: `http://doi.acm.org/10.1145/1831708.1831738`.

[23] Li K, Reichenbach C, Csallner C, Smaragdakis Y. Residual investigation: predictive and precise bug detection. In: Proceedings of the 2012

International Symposium on Software Testing and Analysis. ACM; 2012. p. 298–308.

[24] Madhavan P, Wiegmann DA, Lacson FC. Automation failures on tasks easily performed by operators undermine trust in automated aids. Human Factors: The Journal of the Human Factors and Ergonomics Society. 2006;48(2):241–256.

[25] Jackson D, Rollins EJ. A new model of program dependences for reverse engineering. SIGSOFT Softw Eng Notes. 1994 Dec;19(5):2–10. Available from: `http://doi.acm.org/10.1145/195274.195281`.

[26] Boshernitsan M, Doong R, Savoia A. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In: Proceedings of the 2006 international symposium on Software testing and analysis. ACM; 2006. p. 169–180.

[27] Zeller A. The Future of Programming Environments: Integration, Synergy, and Assistance. In: Future of Software Engineering, 2007. FOSE '07; 2007. p. 316 –325.

[28] Linton MA. Implementing relational views of programs. ACM SIGPLAN Notices. 1984;19(5):132–140.

[29] Lam MS, Whaley J, Livshits VB, Martin MC, Avots D, Carbin M, et al. Context-sensitive program analysis as database queries. In: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. PODS '05. New York, NY, USA: ACM; 2005. p. 1–12. Available from: `http://doi.acm.org/10.1145/1065167.1065169`.

[30] Smaragdakis Y, Bravenboer M. Using Datalog for Fast and Easy Program Analysis. In: de Moor O, Gottlob G, Furche T, Sellers A, editors. Datalog Reloaded. vol. 6702 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg; 2011. p. 245–251. Available from: `http://dx.doi.org/10.1007/978-3-642-24206-9_14`.

[31] Ali K, Lhoták O. Application-Only Call Graph Construction. In: Noble J, editor. ECOOP 2012 – Object-Oriented Programming. vol.

7313 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg; 2012. p. 688–712. Available from: http://dx.doi.org/10.1007/978-3-642-31057-7_30.

[32] Ramachandra K, Guravannavar R, Sudarshan S. Program analysis and transformation for holistic optimization of database applications. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. SOAP '12. New York, NY, USA: ACM; 2012. p. 39–44. Available from: http://doi.acm.org/10.1145/2259051.2259057.

[33] Fosdick LD, Osterweil LJ. Data flow analysis in software reliability. ACM Computing Surveys. 1976;8:305–330.

[34] Georgiadis L, Tarjan RE, Werneck RFF. Finding Dominators in Practice. J Graph Algorithms Appl. 2006;10(1):69–94.

[35] Mclendon Iii W, Hendrickson B, Plimpton SJ, Rauchwerger L. Finding strongly connected components in distributed graphs. Journal of Parallel and Distributed Computing. 2005;65(8):901–910.

[36] Emami M, Ghiya R, Hendren LJ. Context-sensitive interprocedural points-to analysis in the presence of function pointers. SIGPLAN Not. 1994 Jun;29(6):242–256. Available from: http://doi.acm.org/10.1145/773473.178264.

[37] Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. PLDI '07. New York, NY, USA: ACM; 2007. p. 278–289. Available from: http://doi.acm.org/10.1145/1250734.1250766.

[38] Laboratory E. SIR Usage Information;. Available from: http://sir.unl.edu/portal/usage.php.

[39] Pytlik B, Renieris M, Krishnamurthi S, Reiss SP. Automated Fault Localization Using Potential Invariants. CoRR. 2003;cs.SE/0310040.

[40] Abreu R, González A, Zoeteweij P, van Gemund AJC. Automatic software fault localization using generic program invariants. In: Proceedings of the 2008 ACM symposium on Applied computing. SAC '08. New York, NY, USA: ACM; 2008. p. 712–717. Available from: `http://doi.acm.org/10.1145/1363686.1363855`.

[41] D'Silva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on. 2008;27(7):1165–1178.

[42] Tripp O, Pistoia M, Fink SJ, Sridharan M, Weisman O. TAJ: effective taint analysis of web applications. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. PLDI '09. New York, NY, USA: ACM; 2009. p. 87–97. Available from: `http://doi.acm.org/10.1145/1542476.1542486`.