ABSTRACT
For higher quality software, static analysis and dynamic analysis should be used in a complementary manner. In this work, we explore the concept of partitioning a program such that the partitions can be analyzed separately. With such partitioning, potentially different analysis techniques can be applied to different program partitions, with each analysis having to deal with reduced code which can increase its effectiveness. Our experiments show that partitioning can indeed increase the effectiveness of static analysis. Our initial experience in using testing data for partitioning and applying static analysis only on the complementary partition, indicates that it results in detection of more errors which might have gone unnoticed otherwise.

Categories and Subject Descriptors
D.2.4 [Software/Program Verification]: Miscellaneous

General Terms
Verification, Reliability

Keywords
Static analysis, Dynamic analysis, Program partitioning

1. INTRODUCTION
Reliability of the software system is the basic concern during development. To achieve high reliability in software system the bugs present in the program must be detected and removed from the program. In general, the behavior of a program can be known only at runtime. However, some of the inconsistencies and errors prevailing in the program can be directly caught by analyzing the code. These errors or bugs present in the program may cause undesired effects during execution.

Static analysis is one of the techniques used for detecting defects in programs. In static analysis, the program is not executed and the analysis is performed on the source code or some representation of the program. The results observed from such an analysis will remain valid for all input data. Hence we can have guarantee on the applicability of the inferences made by static analysis. Model checkers [2, 5], annotation checkers [4, 7], pattern checkers [12, 15] etc are different approaches for static analysis. There are many static analysis tools available publicly and commercially to detect bugs in the program [10, 3]. However, a complete static analysis of the program through all the paths is not always possible, as for efficiency reasons many static analysis tools limit the analysis to few paths or maintain less information during analysis [1].

Dynamic analysis [6] is the process of analyzing the properties of a program based on its behavior during execution. Testing and profiling are two major techniques used in dynamic analysis. Dynamic analysis examines the program during runtime and derives properties that hold true for one or more executions. The results of dynamic analysis do not generalize to future executions of the program and the observed execution may not be representative of all possible program executions. The results obtained from this analysis are precise, but depend on the given input.

Since a particular static analysis may be limited in terms of context-sensitivity, path-sensitivity, state space or other features, it may be advantageous to split the program into parts and have the analysis process each part separately. We propose an approach which partitions the program using its control flow graph. As each partition has fewer paths, a static analyzer is encouraged to do a more thorough analysis which it could not do earlier due to the large number of paths, potentially leading to detection of more defects. If some dynamic technique like testing has already been applied, then using its history we can use partitioning to create a partition that includes only the parts that have not been tested/analyzed and then analyze this partition through static analysis.
Program partitioning can provide a framework for combining dynamic and static analysis techniques. However, for analysis, it should be clear that we cannot arbitrarily partition a program, and partitioning has to preserve properties like program behavior. In this paper, we develop the concept of proper partitioning, which partitions a program in a manner that each partition has the same behavior as the original program, though for a subset of input data. Note that this concept of program partitioning is different from that of partitioning the input domain for the purpose of selecting test cases, as is done in equivalence class partitioning or in [8].

To study the impact of partitioning on static analysis, we implemented a simple partitioning algorithm for Java programs and then did some experiments using FindBugs and Jlint as the static analyzer. Our experiments indicate that partitioning and then separately (statically) analyzing the partitions, detects new defects which were not identified from the whole program analysis.

For using partitioning as a framework for combining static and dynamic analysis, we use the data of dynamic analysis to create a partition. Then we determine the complementary partition and subject it to static analysis. We have used this approach to combine testing and static analysis, and have developed the necessary tools for it, including for capturing data in testing and then feeding it to a partitioning algorithm to find the complimentary partition. Our experiments show that indeed combining them using partitioning is feasible, and as static analysis of a partition has a better performance, new warnings do get identified.

This work is an extension of our earlier work [14], where we did some experiments in combining testing and static analysis. In this work, we have developed the concept of proper partitioning, which forms the foundation of the framework. We have also done experiments to study the impact of partitioning on static analysis, as it is through improvement in performance of static analysis that we do get the benefits. We have also conducted further experiments in combining testing and static analysis using a different static analyzer. A related work is the Java Path Finder[9], a model checking tool for Java which uses runtime information to guide its model checking engine for detecting deadlocks. PREfix[1] also performs static analysis over single paths by actually simulating them. Our concept of using dynamic analysis to indicate the parts of program where static analysis needs to be done, differentiates it from these works.

In next section we describe the basic concepts of program partitioning. Section 3 explains the approach for static analysis using partitions. In Section 4 we present a framework for using program partitioning to combine static and dynamic analysis. Section 5 contains conclusions and future possibilities.

2. PROGRAM PARTITIONING

In program partitioning a program is divided into two (or more) parts where each partition represents a subset of the original program. As our aim is to analyze the program partitions to find real defects that were not identified from whole program analysis, the partitioning has to be done in a manner such that the bugs in the partition are very likely to represent bugs in the original program. The notion of partitioning depends on the concept of the control flow graph (CFG) of a program. A CFG is a static representation of the program, and represents all alternatives of control flow. Each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets. There are two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

Let G represent the CFG of a program P. We define a program P₁ with CFG G₁ to be a proper partition of P if
the following properties are satisfied:

- \( G_1 \) is a sub-graph of \( G \)
- \( G_1 \) includes both the entry and the exit block of \( G \).
- For every node in \( G_1 \), there should be at least one path from the entry block to the exit block which includes this node.

This will ensure that (1) the partition \( P_1 \) is a valid program (2) the CFG of the partition will always have a path from the entry block to the exit block of \( G \) and (3) set of paths in \( G_1 \) is a subset of paths in \( G \) which implies that for a subset of input domain of \( P \), the behavior of \( P_1 \), in terms of final output as well as intermediate states, is same as that of \( P \) (Note that the behavior of such a partition can be different from that of the original program from the inputs outside this subdomain).

We partition the program \( P \) into two parts \( P_1 \) and \( P_2 \) (We can partition into more parts but we restrict to partitioning to two only). Let \( G_1 \) and \( G_2 \) be their respective control flow graphs. We call a partitioning as proper partitioning if the following holds:

- \( P_1 \) and \( P_2 \) are proper partitions.
- Union of set of nodes in \( G_1 \) and \( G_2 \) is same as set of nodes in \( G \).
- The set of paths (from entry to exit block) in \( G_1 \) and \( G_2 \) are disjoint.

By these conditions we have all the nodes (of \( P \)) covered in \( P_1 \) and \( P_2 \) to ensure that no code segment is left un-analyzed. An example of proper partitioning is shown in Figure 1. The result of partitioning on a program is explained in Figure 2. So, with proper partitioning we have two proper partitions \( P_1 \) and \( P_2 \), both providing same behavior as \( P \) but for disjoint input subdomains. When proper partitioning is done in this manner, we call \( P_1 \) (or \( P_2 \)) the complementary partition of \( P_2 \) (or \( P_1 \)), as \( G_1 \) includes all the nodes in \( G \) that are not present in \( G_2 \).

3. STATIC ANALYSIS WITH PARTITIONS

Our approach for combining static and dynamic analysis is to use data from dynamic analysis to construct a partition and then subject the complementary partition to static analysis. This approach will be beneficial if static analysis of a partition reveals more defects. So, we first study the impact of partitioning on the effectiveness of static analysis. Static analysis of a partition can be helpful in identifying new bugs because of the following reasons. Firstly, the reduction in size and number of paths in the program helps in reducing the state space of the static analyzer. This allows the analyzer to perform a more thorough and focused analysis of the partitions separately. Secondly, reduction in number of conditionals also helps in identification of new bugs in case of several analyzers like FindBugs [11], which carry the context of variables only till one conditional statement.

3.1 Partitioning

There are many ways for doing proper partitioning of a program. Here we suggest a simple approach to partition a program \( P \) into two parts. We build the first partition \( (P_1) \) by generating a random path from entry to exit block of \( P \). If the size of this path exceeds some threshold \( t \) (fraction of total number of blocks in \( P \)), then this random path represents the first partition, else we generate more random paths until the size of union of these paths exceed \( t \). The union of these paths is our partition \( P_1 \). Then the complementary partition is determined as the smallest control flow graph containing all the blocks not in \( P_1 \). Note that in this way no new paths are added and all the properties for proper partitioning stated above are satisfied.

The basic approach for creating two partitions can be extended for multiple partitions. For this we may use a recursive approach of dividing the bigger partitions further. However in our experiments we focus on two partitions only.

We have implemented this approach to create the partitions of Java programs. For creating the partitions \( P_1 \) and \( P_2 \), we changed the Java class file so that it reflects the new program which has only those paths which result from our program partitioning (by suitably replacing and/or adding jump instructions so that only necessary paths are present in the current partition). Finally, the modified class files which reflect our partitioning are given as input to the static analyzer. The prototype is completely written in Java.

3.2 Experiments

We have evaluated the effectiveness of partitioning on static analysis for a few Java packages with FindBugs as the static analysis engine. The packages which we have used for program partitioning and subsequent static analysis are Java Coverage Analyzer, Evolve, PIMS, JavaRisk and another package for graph plotting. In the experiments, each package is divided into two partitions according to the method given in the previous section and subjected to static analysis with FindBugs. All the data required for evaluation was collected including time of execution, number of reported bugs and newly identified bugs with our framework.

We have applied the approach of partitioning and tried to identify the real defects among the reported warnings. This experiment reveals that partitioning actually finds new defects in the program. The results of our experiments are shown in Figure 3. We also studied the effect of size of partition on the effectiveness of static analysis on a different set of programs, the results of which are shown in Figure 4.

During the experiments it was realized that the approach was limited in certain aspects as selecting the first partition randomly may select a path which cannot be possibly taken during the actual execution of the program. This may lead to false positives (when this program partition is given as input to the static analyzer), which were not present in the static analysis of the original program. From our experiments we found that the proportion of these false positives was significantly lower than that of true warnings generated after partitioning. We can overcome this limitation by selecting feasible paths only in the first partition using dynamic analysis as described in the next section.

4. USING PARTITIONING FOR COMBINING STATIC AND DYNAMIC ANALYSIS

We have seen that program partitioning can be used to
### Program/Package

<table>
<thead>
<tr>
<th></th>
<th>Jrisk</th>
<th>PIMS</th>
<th>Coverage Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WHOLE PROGRAM ANALYSIS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total warnings in $P$</td>
<td>34</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>Real defects found in $P$</td>
<td>21</td>
<td>29</td>
<td>24</td>
</tr>
</tbody>
</table>

|                      |       |      |                   |
| **PARTITION BASED ANALYSIS** |       |      |                   |
| Total new warnings in $P_1 + P_2$ | 20 | 8 | 3 |
| New real defects in $P_1 + P_2$ | 14 | 7 | 2 |

**MISSED WARNINGS / DEFECTS FROM PARTITIONING**

(Warnings present in whole program analysis which were missed by partitioning)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall missed warnings</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Missed defects</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 3: Impact of partitioning on Static Analysis

Figure 4: Size of $P_1$ Vs Number of new warnings

Figure 4: Effect of partition size on newly found bugs

make static analysis more effective. We would like to leverage this property for combining static and dynamic analysis in a complementary manner. In this section we explore this aspect. It would seem that one natural approach to using two different techniques on a program would be to partition the program into two partitions and apply the two techniques on them separately. We already know that for static analysis this is better. However, dynamic analysis involves execution of the programs with some inputs, collecting information from these executions, and then making some assertions about the program behavior. It is not clear how dynamic analysis can be restricted to operate on one partition only. Hence we take a different approach. Instead of applying dynamic analysis on a partition we apply dynamic analysis on the whole program and use the data from it for partitioning. We describe the approach and some preliminary experiments we have done with it.

### 4.1 Partitioning using data from dynamic analysis

Dynamic analysis by its nature evaluates only that path of the program which is actually executed. One way to have dynamic and static analysis in complementary manner is to first perform dynamic analysis, use the dynamic analysis information to prune the program and then subject the pruned program to static analysis. For this approach, we have to find suitable methods of pruning a program. This is where the concept of program partitioning can be used. During the dynamic analysis we instrument the program and collect data on which paths have been followed during the analysis. As a union of paths from the entry to exit block form a proper partition, we construct a partition using these paths. Then we use the concept of proper partitioning to determine the complementary partition which will have only those paths that have not been examined in dynamic analysis. The complementary partition can then be subjected to static analysis. We have seen in our experiments discussed above, that static analysis of partitions can yield better results than analyzing the whole program.

Let $G_1$ be the subgraph that has been analyzed during dynamic analysis. $G_1$ is actually the set of nodes and edges visited during the dynamic analysis. As dynamic analysis is based on execution, $G_1$ will be formed from the nodes and edges in actually executable complete paths (paths from entry block to exit block). Hence $G_1$ will form a proper partition $P_1$. Now we have to form the complementary partition $P_2$ which contains the code not analyzed in dynamic analysis. We construct $P_2$ based on the smallest possible control flow graph containing all the blocks not in $P_1$. This approach is given in Figure 5. Partition $P_2$ is then subjected to static analysis. This approach will be more beneficial if the dynamic analysis is performed till the size of $G_1$ is sufficiently large to cover a good fraction of total blocks of $P$.

### 4.2 Implementation and Experiments

Combining static and dynamic analysis based on program partitioning can work with different analysis techniques. We have used this approach to combine static analysis with testing for Java programs. For collecting path data from testing we have used a coverage analyzer that stores the path executed for each test case [13]. This also allows us to combine different test cases for the analysis. Our test coverage analyzer instruments the byte code for collecting the coverage
Data Structures:

- $B$ : Set of all blocks
- $B_i$ : Blocks in partition $P_i$

Method:

1. Generate the control flow graph (CFG) for the program.
2. Generate partition $P_1$ by including all the blocks of $P$ which have been covered during dynamic analysis.
3. Include all the blocks of $P_1$ in $B_1$.
4. Identify all the blocks $B'_1 = \{ B - B_1 \}$ which are not present in the partition $P_1$.
5. Build partition $P_2$ using the blocks $B'_1$ which are not present in $P_1$. Generate all the complete paths from start block to exit block passing through any block in $B'_1$. Include all the blocks in the path in $B_2$ (i.e add them in $P_2$).
6. Identify the blocks $B'_2 = \{ B - B_2 \}$ which are not needed for partition $P_2$.
7. Create the partition $P_2$ by removing the links to $B'_2$ (Retaining only the links within $B_2$).

Figure 5: Program Partitioning for combining dynamic and static analysis

Data. We have built a tool that given a bytecode and a set of blocks to be removed, converts it into bytecode of the complementary partition using the approach given in Figure 5. Our earlier experiments[14] with some Java programs and FindBugs show that more warnings and defects can be found using this approach.

We have performed the experiments with three packages: EIRC \(^1\), EVolve \(^2\) and STEP \(^3\), using Jlint as the static analyzer. In our experiments we first analyzed the complete program using Jlint and recorded the number of warnings generated. Then we randomly tested the program with some test data and then using the coverage data determined the complementary partition. Finally we analyzed the complementary partition using Jlint. The summary of experiment data is given in Figure 6.

We see that nearly 10% new warnings were issued by analysis of the complementary partition. These new warning are for the code in the complementary partition. As these are third party programs, we did not analyze what fraction of warnings are indeed real defects. As partitioning can introduce some very obvious false positives in some cases, we examined the new warnings to look for these. Of the warnings, generally about 15%-20% were such that they came about because of the fact that the partition is the representative of the original program only for a restricted input subdomain, while the static analyzer performs the analysis for complete input domain (and hence are obviously false warnings).

These experiments indicate that this approach for combining dynamic and static analysis is indeed feasible and provides better defect detection capability. However, our current experiments were limited to using testing as the dynamic analysis approach. We believe that the approach will be effective for other methods as well, but experiments need to be done to validate it.

5. CONCLUSION AND FUTURE WORK

It is generally believed that static and dynamic analyses are complementary and should be performed in a manner to support each other. However, few formal frameworks are available for achieving this. In this work we propose the framework of program partitioning for combining dynamic techniques with static analysis for a better analysis of a given program.

In program partitioning we partition a program into two proper partitions, one being the complementary of other. Each proper partition includes a set of nodes from the CFG of the program and provides the behavior of the original program for a limited input subdomain. Our experiments have shown that static analysis on a partition can reveal more defects than analyzing the whole program.

For combining dynamic and static analysis, we propose that using information from dynamic analysis we form a proper partition of the program. Then we determine the complementary partition, for which we have also proposed an algorithm. The complementary partition can be subject to static analysis. We applied this approach for combining testing with static analysis. We have built tools to capture the needed path data during testing for Java programs and then we use the data to determine the complementary partition. We did some experiments using Jlint for static analysis of the complementary partition. We obtained encouraging results from our experiments and observed nearly 10% increase in the number of warnings found from static analysis of the complementary partition. Similar results have also been observed using FindBugs.

\(^1\)http://eirc.sourceforge.net/
\(^2\)http://www.sable.mcgill.ca/evolve/
\(^3\)http://www.sable.mcgill.ca/step/main.html
A limitation to the current approach is that the program partition is valid only for a subdomain of the input, but the static analyzer analyzes it for the whole input space resulting in a few false positives in the warnings obtained from the complementary partition. Future work in this area can aim to overcome this shortcoming and use other dynamic analysis approaches to partition a program in a manner that the static analyzer finds more defects in the resulting partition.

The concept of partitioning need not be used only for combining two types of techniques. It can also be used for combining two different methods of static analysis - different partitions being analyzed separately potentially using different analyzers. We are currently developing methods of making static analysis more effective using partitioning.

6. REFERENCES


