Sieve: A Tool for Automatically Detecting Variations Across Program Versions

Murali Krishna Ramanathan Ananth Grama Suresh Jagannathan
{rmk, ayg, suresh}@cs.purdue.edu
Department of Computer Science
Purdue University, West Lafayette, IN 47907

Abstract

Software systems often undergo many revisions during their lifetime as new features are added, bugs repaired, abstractions simplified and refactored, and performance improved. When a revision, even a minor one, does occur, the changes it induces must be tested to ensure that invariants assumed in the original version are not violated unintentionally. In order to avoid testing components that are unchanged across revisions, impact analysis is often used to identify code blocks or functions that are affected by a change. In this paper, we present a novel solution to this general problem that uses dynamic programming on instrumented traces of different program binaries to identify longest common subsequences in strings generated by these traces. Our formulation allows us to perform impact analysis and also to detect the smallest set of locations within the functions where the effect of the changes actually manifests itself. Sieve is a tool that incorporates these ideas. Sieve is unobtrusive, requiring no programmer or compiler intervention to guide its behavior. Our experiments on multiple versions of open-source C programs shows that Sieve is an effective and scalable tool to identify impact sets and can locate regions in the affected functions where the changes manifest. These results lead us to conclude that Sieve can play a beneficial role in program testing and software maintenance.

1 Introduction

Revisions to an existing piece of software can occur for a variety of reasons. These include the addition of new features and functionality, code restructuring to improve performance, or refactoring for improved maintainability. Regardless of the reasons that cause a revision, testing the effects of its changes is important. Revisions are rarely intended to violate backward compatibility; existing functionality and invariants should thus not be affected as a result of changes that occur between two versions of a program. Quite often, however, this dictum does not hold. Changing a set of components in a program can sometimes result in unwanted changes in other components, leading to software defects and bugs. As a result, expensive test regimes are required [10]. Recent work on isolating and correcting software bugs [12, 17, 27, 18] provide efficient strategies for testing a single instance of a program with respect to desired invariants, but they do not easily generalize to comparing changes across multiple program versions.

We focus our attention on identifying dissimilarities across program versions. We do so by using test results on older versions to automatically identify regions in newer versions that are affected (or impacted) by the changes that characterize their differences; it is precisely these regions that merit comprehensive review and more elaborate testing using sophisticated techniques [12]. We state this problem more formally as follows:

"*Given two versions of a program, is there an efficient mechanism to dynamically detect the functions affected in the newer version by modifications made to the older? Moreover, can we precisely identify the regions in the affected functions where the effect of these modifications manifest?*"

In seminal work, Law and Rothermel [16] define the problem of dynamic impact analysis and presented a solution, PathImpact, based on whole program path profiling. In subsequent work [15], they present algorithms that allow the data needed by PathImpact to be collected incrementally. In [6], Breech *et al.* present an online approach for calculating impact sets. Execute-after sequences [1] and coverage impact analysis [22] also attempt to identify functions that are potentially affected by a program change using program traces and test data. In [1], Apiwattanapong *et al.* describe an efficient and precise dynamic impact analysis technique based on the following hypothesis: "if a function follows a modified function in at least one execution sequence, it is affected." The algorithm used to detect the affected functions has similar precision as path impact analysis but is more efficient. At the other extreme, the execute-after sequence approach is as efficient as coverage impact

analysis, but is more precise. Ren *et al.* present a tool for change impact analysis of Java programs in [25]. Their approach analyzes two versions of a program, and decomposes their difference into a set of atomic changes. The impact of changes between the versions is reported in terms of affected tests whose execution behavior is influenced by these changes.

While existing designs for impact analysis are significant first steps, they provide only a partial solution to the problems we consider. Since the sole purpose of these conservative designs is to detect impact sets efficiently, current solutions are unable to identify precisely the regions in a newer version of a program that are affected by changes to an older version. To achieve this degree of precision requires accurate tracking of program execution. For example, functions in an execution sequence that are invoked after a call to a modified function may nonetheless be totally unaffected by the modifications made. Even with precise knowledge about a program execution's control and dataflow behavior, new techniques are still required to precisely identify the regions within impacted functions that are affected by changes.

The design of our approach is motivated by solutions to similar problems in computational biology. Mutations are a common phenomena in biological systems. Intuitively, we envision multiple versions of a program as being analogous to collections of mutations from an original source. One popular way to perform sequence matching for the purpose of identifying mutations is to abstract it to the problem of finding an optimal alignment between two sequences using dynamic programming. The optimal alignment problem is a dual of the popular longest common subsequence problem [8]. Dynamic programming vis-a-vis the longest common subsequence problem is a powerful tool, and more effective than simple string matching because it helps to identify the minimum set of locations that cause a mismatch between two strings. In contrast, string matching always provides a boolean response. Dynamic programming also provides flexibility to define the cost function for alphabet (mis)matches.

Based on these insights, we develop a tool called Sieve for identifying regions of change across program versions. Over a range of benchmarks, the results of our experiments show a reduction of 30-60% in the number of functions that are marked as impacted compared to impact analysis based on execute-after sequences. Furthermore, we also observe that the majority of affected functions across all benchmarks have small regions where changes manifest; typically the size of these regions is three lines or less. The significance of the latter result is that Sieve simplifies the task of determining if changed behavior in a revision is intended or accidental, and facilitates devising test suites to validate desired properties on revisions.

Sieve, like other dynamic impact analyzers, can generate false negatives, i.e., functions that are actually affected may go undetected due to the quality of the test inputs. Because impacted functions are identified based on the effects of their observed actions, Sieve is more sensitive to test input quality than related systems, e.g., [1], that use coarser-grained techniques (e.g., function call graphs) to build impact sets. Thus, we expect Sieve's utility to be best exploited when comprehensive test suites are available.

The impact sets produced by Sieve are also intimately tied to the set of operations that are tracked. In this paper, we evaluate a Sieve implementation that tracks all operations to the program stack and heap, as well as an implementation that tracks only heap operations. If two versions of a function differ in the use of a register allocated variable only, neither of these implementations identify the function as being impacted by the change. For realistic programs, we believe these simplifications do not lead to significant loss of safety.

On the other hand, this sensitivity can also lead to greater precision, and thus fewer false positives. In other words, a test input that exercises different behavior across two versions of a function in terms of the operations tracked by Sieve will result in these functions being flagged as impacted. Conversely, unlike other dynamic impact analyzers, a test input that does not reveal different behavior across two versions of a function will not result in that function being flagged even if changes were introduced upstream in the function's callers.

In Execute-After Sequences [1] (EAS), to safely estimate the impact set of functions [21], the newer version is executed on some test inputs and the functions traversed in the process of execution are time stamped. Our approach also shares similarities with this technique in the methodology of trace collection, except as opposed to time-stamping the functions traversed, we track fine granular data (memory read or write) in the newer version. In both cases the dynamic execution of the newer version must be monitored, albeit at different granularity. As in EAS [1], we also demonstrate that dynamic tracking can be efficient and scalable.

This paper makes the following technical contributions:

1. **New Mechanism:** We propose a novel mechanism to abstract program behavior. Our technique considers program execution in terms of memory reads and writes and use dynamic programming to detect variations across two different (binary) program versions. No *a priori* information is needed to help identify changes across program versions, other than a mapping to indicate the functions that should be compared.

2. **Impact Analysis:** Our technique automatically detects functions in a newer version that are (un)affected by

the modifications made to an older version. The precision and safety of the approach is based on the quality of test inputs, as is the case with many comparable designs and testing methodologies.

3. **Efficiently Identifying Changed Regions:** We identify regions of code in affected functions at which the changes to the source manifest themselves in the program. The proposed approach is scalable and is a function of the program size.

4. **Sieve:** We have implemented a tool using our approach that has been tested on a number of open-source C programs. Sieve uses binary program instrumentation and dynamic programming on memory traces derived from instrumented programs. No annotation of program sources or compiler enhancements are required.

1.1 Overview

As a first step towards detecting and isolating variations, we abstract a program as a sequence of memory reads and writes. Test input is fed into two versions and the trace is collected using binary instrumentation. A trace is composed of sequences of <*Operation, Value*> tuples, where *Operation* is either a read or write to memory and *Value* is the value read from, or written to memory. The trace is analogous to a string and the tuple analogous to an alphabet. Comparing two functions that exist in two program versions is equivalent to comparing the subsequence of the trace corresponding to the two functions under comparison. Based on a user-defined cost function, the Levenstein [13] distance is calculated and the gaps [4] in the comparison recorded. Recall that the Levenstein distance between two strings is defined as the shortest sequence of edit operations that lead from one string to the other. By repeating the process for multiple test inputs, cumulative information on the gaps present in the older version relative to the newer version is obtained. By mapping the tuples back to the corresponding regions in the source, information on the affected locations within an impacted function is obtained. If the Levenstein distance between the two functions is zero, we regard the function in the newer version as being unaffected by changes in the older version.

Given memory traces of length m and n for two versions, the time complexity of dynamic programming is $O(mn)$. Thus, even traces of modest length (approximately 15K) can considerably slow down the comparison process. Indeed, for some applications, there are a several million reads or write operations to memory. We found in earlier implementations of Sieve that comparison across dynamic traces using the above described representation incurs a significant cost and may not scale to larger systems. The following optimization reduces this cost significantly.

Instead of representing a trace as a sequence of tuples of type $< Operation, Value >$, we represent a trace as a sequence of tuples of type $< \ell, h >$, where ℓ is the line number in the source and h is a hash of ordered sequences of $< Operation, Value >$ tuples corresponding to operations performed by ℓ. Using this representation has two implications with respect to efficiency:

1. The amount of space required to store the trace is proportional to the number of lines in the program and not the number of instructions executed.

2. The time taken for performing alignment using dynamic programming becomes negligible.

Observe that a hash need not necessarily be calculated at the end of program execution. Efficient techniques based on Rabin fingerprinting [23] are available to compute hashes intermittently. This ensures that the amount of memory required to hold the trace is small. For the benchmarks used in this paper, it is sufficient for us to calculate the hash at the end of program execution, and using Rabin Fingerprinting was not necessary. In the rest of the paper, any reference to a trace refers to the trace obtained using the optimization technique discussed above, unless explicitly stated otherwise.

Since this optimization critically relies on hashing to reduce overheads, it is certainly possible to construct examples that exhibit behavior different from the idealized technique described above. Indeed, undesirable false negatives that impact the safety of the analysis could be introduced simply because a collection of dissimilar $< Operation, Value >$ sequences in two versions of a function have the same hash. Our experimental results indicate though that improvements in efficiency due to the optimization does not come at the expense of safety or accuracy for realistic programs, and that the potential loss of safety due to unintended hash collisions does not occur in practice.

2 Motivation and Background

Common modifications to a function include adding new variables, renaming or deleting existing variables, changing the interface of the function by adding or deleting parameters, changing return values, inlining function calls, making external state changes, or modifying function logic. Some of these changes, for example, variable renaming or inlining, have no effect on other functions in most cases; on the other hand, modifying program logic or making external state changes can affect other function behavior. Since testing is an expensive process, focusing test cases on function components changed as a consequence of this latter category is beneficial. Even here, changing a function's logic

may not necessarily lead to observable change in the function's callers.

Our technique for detecting locations at which changes to an older version lead to different behavior in the newer one is similar to solutions to related problems in the area of computational biology. More specifically, alignment of new sequences with previously characterized sequences can help in characterizing molecules corresponding to the new sequence [4]. The problem of aligning two sequences is abstracted into the longest common subsequence problem. The solution to this problem [8] is a popular application of dynamic programming. Finding the minimum edit distance between any two strings is a dual to the longest common subsequence problem.

```
1  void main(){          void main(){
2    ...                    ...
3    old(s);                new(s);
4    f(s);                  f(s);
5    ...                    ...
6 }                      }

7
8  void old(LIST *s){   void new(LIST *s){
9    LIST *t;               LIST *r, *p;
10
11                          r = malloc(LIST);
12   t = s->next;           p = s->next;
13                          r->next = s;
14
15   while(s != NULL){     for(u = s;
                             u != NULL;u=u->next){
16     print(s->val);        print(u->val);
17     s = s->next;
18   }                     }

19                         s = delete_r(r);
20
21   if(t->val > NUM)      if(p->val > NUM)
22     print("error");       print("error");
23 }                       }
```

Figure 1. Two different versions of a list manipulating procedure

For example, given two strings aabcabcd and abacbd, the longest common subsequence is aacbd. One possible alignment of these two sequences is: a-abcabcd and aba-c-b-d. The edit distance in this case is four, assuming unit cost for insertions and deletions. The optimality of an alignment is dependent on the cost function used, which can be defined in many ways. In this paper, we consider a simple notion of optimality. The space introduced into an alignment to compensate for insertions and deletions in one sequence relative to another is defined as a *gap* [4]. Gaps in our alignments have unit cost, while all other alphabets have zero cost. Thus an optimal alignment is one that has the smallest number of gaps; observe that for any pair of strings, there maybe many such optimal alignments. The flexibility in defining cost based on the application context is an important characteristic that makes it useful for applications in sequence alignment. As we describe below, we also make use of this flexibility in our approach.

2.1 Example

A motivating example is given in Figure 1. We show two program fragments, one labeled old, and the other new. Both procedures perform similar actions involving traversing and printing elements of an input list. However, new adds a new temporary cell, and subsequently deletes it in delete_r before returning. Assuming delete_r is implemented correctly, the behavior of the two procedures is exactly the same with respect to their callers.

```
Trace Element: <Operation,Value>
Op: Read(R),Write(W)
Value: 32 bit value
q: new cell allocated by malloc in  new

old: <R, y>, <W, y>, <R, 10>, <R, y>,
     <W, y>, <R, 15>, <R,φ>, <W,φ>, <R, 15>

new: <W, q>, <R, y>, <W, y>, <R, x>, <W, x>,
     <R, x>, <W, x>, <R, 10>, <R, y>, <W, y>,
     <R, 15>, <R,φ>, <W,φ>, <R, q>, <W, x>,
     <R, 15>
```

Figure 2. Memory Trace (without hashing) associated with the functions in Figure 1

Using our approach, memory traces associated with the invocation of these procedures on the same test input are first obtained. Suppose the list referenced by s contains pointers to cells $\{x,y\}$, where x holds 10, y holds 15. The memory trace generated for the approach without any hashing optimization is given in Figure 2. The associated alignment is shown in Figure 3.

Based on the approximation introduced in Section 1.1, the hash of the sequence of operations performed in each line is given in Table 1. Correspondingly, the memory trace generated and the optimal alignment obtained using dynamic programming is shown in Figure 2. In the figure, gaps are represented by a hyphen. Consequently, the regions in the actual source can also be aligned. For example, the statement r->next = s in new does not have a corresponding statement in old. This corresponds to a gap in sequence alignment. Similarly, other gaps are present for

```
old: -, <R, y>, <W, y>, -, -, -, -, <R, 10>,
     <R, y>, <W, y>, <R, 15>, <R,φ>, <W,φ>,
     -, -,  <R, 15>

new: <W, q>, <R, y>, <W, y>, <R, x>, <W, x>,
     <R, x>, <W, x>, <R, 10>, <R, y>, <W, y>,
     <R, 15>, <R,φ>, <W,φ>, <R, q>, <W, x>,
     <R, 15>
```

Figure 3. Alignment for the trace (without hashing) shown in Figure 2. The gap cost is 7.

old	new
-	$<11, h_5>$
$<12, h_1>$	$<12, h_1>$
-	$<13, h_6>$
-	$<15, h_7>$
$<16, h_2>$	$<16, h_2>$
-	$<19, h_8>$
$<17, h_3>$	-
$<21, h_4>$	$<21, h_4>$

Table 2. Alignment for the trace shown in Figure 1. The gap cost is 5.

the newly allocated cell, and the call to `delete_r`. Renaming variables (e.g., `t` is renamed `p`), adding new variables, etc. does not trigger an alignment mismatch because their effects remain unchanged. Observe that there are multiple operations in line 15 of procedure `new`. While in our current implementation, we restrict ourselves to line numbers, instrumentation can be done to obtain the column number of the operation and the above approach can be easily extended. Observe that the approach with the approximation can align the operations `s = s->next` and `u = u->next` in the source. However, the hashing approximation may discard the possible alignment. Indeed, such constructs may potentially lead to a larger number of regions in the program being identified as affected than those detected by the optimal approach. Given the cost-benefit tradeoffs, such false positives on the conservative side do not pose a major problem.

Operation sequence	Hash
<R,y>, <W,y>	h_1
<R,10>, <R,15>	h_2
<R,y>, <W,y>, <R,ϕ >, <W,ϕ >	h_3
<R,15>	h_4
<W,q>	h_5
<R,x>, <W,x>	h_6
<R, x>, <W, x>, <R,y>, <W,y>, <R,ϕ >, <W,ϕ >	h_7
<R, q>, <W, x>	h_8

Table 1. Operation sequences and the corresponding hash values

If this were the only change in the program, our approach would identify functions `new` and `delete_r` as potentially affected. In contrast, path impact analysis [16], for example, uses the program's call graph and the syntactically changed functions as markers; it would identify *all* functions that are executed after `new` in any test case as impacted. For example, functions `f`, `main` and functions that succeed `f` would be recorded as affected by these changes.

3 Sieve

3.1 Implementation

Sieve is a tool that consists of two components, an instrumentation module and a comparison module. Both components operate over program binaries. The binaries, representing a program and its revision, are instrumented using PIN [19], and execute on the same test input. The effect of the instrumentation yields memory traces. These traces are then compared using dynamic programming, and optimally aligned based on a user defined cost function. A block diagram of this process is given in Figure 4.

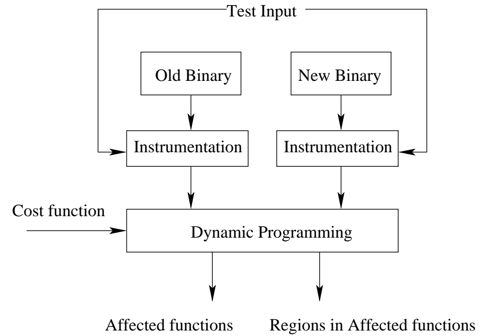


Figure 4. Block Diagram for Sieve.

Gaps in the alignment help detect operations performed by the newer version that are absent in the older version, and vice versa. Accumulating this information over all test inputs provides the set of affected regions in the newer version. If there are no gaps present in such a comparison over all test inputs, Sieve declares the functions to be unaffected. Otherwise, it identifies the affected regions (in the form of line numbers) in the newer version.

3.2 Instrumentation Module based on PIN

We use PIN [19], a dynamic binary instrumentation tool, for instrumentation purposes. PIN supports a rich set of abstract operations that can be used to analyze applications at the instruction level without detailed knowledge of the underlying instruction set. PIN uses dynamic compilation techniques to instrument executables while they are running, and instrumentation code can be inserted at desired locations in the binary.

The instrumentation module takes as input the binary and the list of functions in the binary that need to be instrumented. When the binary is executed on a given test input with dynamic instrumentation, a list of tuples is generated. The elements in the tuple include the type of operation, the value, the line number and the function in which the instruction was generated.

3.3 Comparison Module Using Dynamic Programming

The comparison module operates over traces generated by instrumenting the binaries to be compared as they execute on the same input. To provide an analogy, if the trace is viewed as a string, the equivalence of an alphabet in the string here is a tuple <*Line Number*, *Hash Value*>. A dynamic programming table is constructed. While more sophisticated cost functions can be defined, as a first step, the current implementation has a very simple cost function. The cost at any box, d_{ij} of the dynamic programming table is calculated as follows. If alphabets i and j are equal, i.e., the tuples are equivalent, then the cost $d_{i,j}$ is the minimum of $d_{i-1,j-1}$, $d_{i-1,j} + 1$ and $d_{i,j-1} + 1$. After filling up all the values in the table, a traversal from the end of the table (the last row and last column) through the boxes responsible for the values in the current box, gives the alignment of the two traces. A detailed description of the dynamic programming algorithm for Longest Common Subsequence is presented in [8].

3.4 Memory Aliasing

The values read from, or written to memory, are used in determining the equality of tuples. However, if these values are pointers to memory locations, the values for two different versions need not be the same, yet semantically may point to the same location. This necessitates use of techniques to check whether a value is a pointer. Similar problems arise in garbage collection techniques. In the current implementation of Sieve, we use techniques used in Boehm GC [5]. For example, one of the heuristics to identify whether an object is a value or a pointer examines the most significant byte. If it is set, then the object is considered a pointer and *vice versa*. We also address other issues related to memory aliasing across program versions in [24].

4 Evaluation

4.1 Experimental Setup

We have examined Sieve using two versions of the following software packages: `bzip2` [7], `bunzip2` [7], `gawk` [11], `wget` [26], `gzip`, `grep` and `flex` [9]. All these programs are written in C. The details on the versions used for the benchmarks, the lines of code, the number of functions and other parameters are given in Table 3. Terms used in the table are defined below:

Instrumentation Time (IT): The time taken to insert instrumentation code into the binary and subsequently execute all the test cases.

Analysis Time (AT): The time taken to analyze the results of instrumentation, analyze using Execute-After Sequence (*EAS*) [1], or perform dynamic programming in Sieve.

We explain the significance of the other columns later in the section. The test cases are randomly generated for `bzip2`, `bunzip2` and `wget` and existing test suites are used for the rest of the benchmarks. We perform our tests on a Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on an Intel(R) Pentium(R) 4 CPU 3.00GHz with 1GB memory. The version of the PIN [19] tool used was a special release 1819 (2005-04-15) for Gentoo Linux. The sources were compiled using GCC version 3.3.4.

PIN, the instrumentation tool, yields a sequence of line numbers and a hash value associated with each line number for each function that is instrumented. We assume that older versions of a program are instrumented and traces obtained for the corresponding test cases are presented as input. By executing PIN on the newer version of the program, we obtain a new set of traces that need to be compared with an existing set of traces for the older version. In Sieve, we apply dynamic programming as described earlier on the obtained trace for pairs of functions from the two versions respectively; we obtain the regions (in the form of line numbers) in the newer version that differ from the older version.

Sieve is highly customizable, and users can choose the granularity of memory operations they wish to collect. We present two such instantiations. The first (Sieve$_h$) records information about all operations to heap allocated data. The second (Sieve$_m$), in addition, also records operations to stack locations. As we show below, there is modest improvement in the size of impact sets when stack operations

Benchmark	Old Version	New Version	LoC (in K)	Total Functions	Total Tests	EAS		Sieve_h		Sieve_m	
						IT	AT	IT	AT	IT	AT
`bzip2`	0.9.5d	1.0.2	9	107	101	189	< 1	516	1	1026	< 1
`bunzip2`	0.9.5d	1.0.2	9	107	101	146	< 1	422	< 1	959	< 1
`flex`	v0(orig)	v1(orig)	12	162	525	1265	< 1	2084	< 1	2935	8
`gawk`	3.1.3	3.1.4	41	522	133	298	< 1	572	1	1763	< 1
`grep`	v0(orig)	v1(seed)	11	123	470	842	< 1	1193	< 1	3292	10
`gzip`	v0(orig)	v1(seed)	6	81	217	216	< 1	773	< 1	1189	6
`wget`	1.6	1.7	28	312	105	367	< 1	466	< 1	808	8

Table 3. Benchmark Information and Results (Time in seconds).

are also considered with corresponding increase in instrumentation time. The degree of granularity chosen depends upon a variety of factors, including test case quality, the presumed disparity among two versions, knowledge about program structure (e.g., heap/stack intensive), etc. As mentioned earlier, the granularity level chosen may impact the safety of the analysis if variations between different versions of a function manifest via operations that are not tracked.

4.2 Results

Our experimental results allow us to answer the following questions about our approach:

- If a function is impacted, what are the sizes of regions in the function that are affected?
- Is there any reduction in the number of impacted functions reported using Sieve, compared to EAS?
- Is there a significant change in the rate of detection of impacted functions with increase in the number of test cases?
- What is the time overhead of Sieve compared to EAS?

Figure 5(a) characterizes functions found in the benchmarks with respect to the number of heap read and write instructions they perform. For example, in `bzip2`, roughly 35% of all functions perform fewer than six operations on the heap, and in `wget` roughly 10% of all functions perform more than 18 operations involving the heap.

Figure 5(b) presents, for those functions in a newer version impacted by a change, the size of the affected regions within those functions. For example, in `flex`, we observe that over 43% of all impacted functions have changes limited to three or fewer lines of code. Indeed, for all the applications in our benchmark suite, more than 50% of all impacted functions have fewer than three lines of code impacted by a change and 80% have fewer than 10 lines of code changed (except for `bzip2`).

To quantify Sieve's utility, we faithfully implemented the EAS algorithm, path impact analysis, as described in [1]

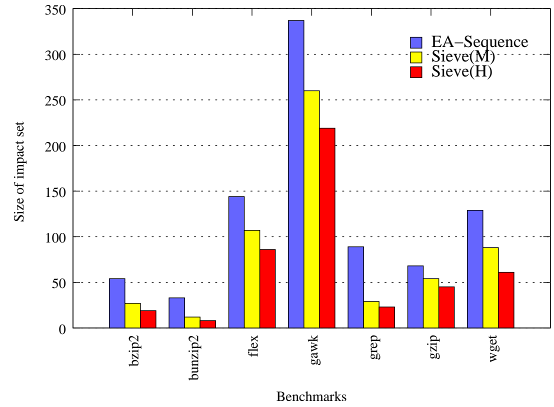
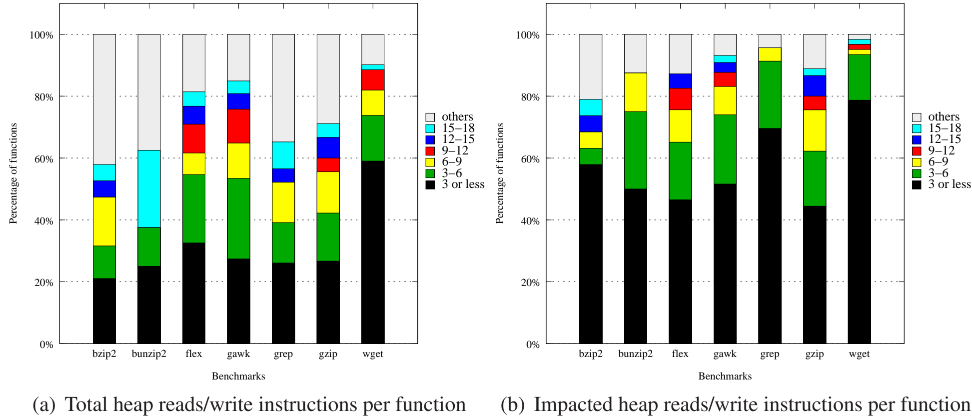


Figure 6. Histogram showing the number of impacted functions detected by EAS and Sieve

for C programs. Typically, functions are compared across versions and marked as (un)changed. A function that follows a changed function in any execution is labeled as affected. Figure 6 presents the the number of functions found to be impacted using Sieve as compared to a EAS. The number of impacted functions identified ranges from 8 for `bunzip2` to 220 for `gawk` under Sieve_h and range from 12 for `bunzip2` to 260 for `gawk` under Sieve_m , a proper subset of the functions identified as affected by EAS. A reduction from 30% to 60% in the size of the impacted set is observed across our benchmark set when comparing Sieve_h (or Sieve_m) with EAS. Interestingly, for some benchmarks tested, we found that the versions syntactically differ (without considering the complexity of the change) at `main` and therefore the list of functions covered by the test suite immediately becomes part of the impact set using EAS. The implication of this result is that when functions present near the root of the call graph are changed, the utility of EAS-like approaches significantly reduce. On the other hand, Sieve is independent of the location of a function in the call graph and does not consider syntactic changes to functions specifically. Furthermore, another consequence of this observation is that the focus of regression testing can be improved be-



(a) Total heap reads/write instructions per function (b) Impacted heap reads/write instructions per function

Figure 5. Histogram (a) shows that most functions in these benchmarks perform a non-trivial number of heap-related operations. Histogram (b) shows that for approximately 60 % of the functions in every benchmark, three or fewer lines within these functions are impacted. The results shown here are based on Sieve$_h$.

cause the set of impacted functions that must be examined, i.e., the set of functions that truly exhibit different runtime behavior across revisions observed by our instrumentation mechanism, is reduced compared to impact analyzers that do not leverage this degree of precision.

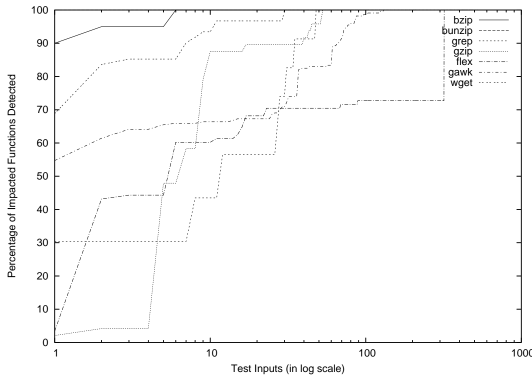


Figure 7. Number of impacted functions found Vs Number of test cases (in log scale)

While executing all test cases may result in precise data, in many instances, it is preferable that the set of impacted functions in the newer version is obtained by executing a smaller number of test cases. Figure 7 presents our experimental results on the number of impacted functions identified, corresponding to the number of test cases executed. Interestingly, a majority of the affected functions are identified by executing the binary on 10 or fewer inputs. For a majority of the benchmarks (except `flex`), the number of impacted functions saturates after executing 20% of the total number of test cases in an ad hoc order. This highlights the fact that test prioritization techniques can play a significant role in reducing this overhead.

In Table 3, we show the instrumentation time (IT) and analysis time(IT) for EAS, Sieve$_h$, and Sieve$_m$, respectively. The analysis time (time taken for dynamic programming or analysis EA-Sequences) is similar, and is less than a second for most of the benchmarks. However, the instrumentation time for Sieve$_h$ (or Sieve$_m$) is modestly higher than that of EAS. This is expected, since Sieve tracks program execution more accurately, as compared to EAS. For a modest increase in instrumentation time, there is substantial improvement with respect to identifying impacted regions. The amount of memory required for Sieve is approximately 4MB for the benchmarks studied.

Both Sieve$_h$ and Sieve$_m$ use hashing to make the analysis scalable. If hash collisions occur frequently, such an optimization would impact Sieve's effectiveness. To examine this issue, we consider instantiations of Sieve$_h$ and Sieve$_m$ that perform no hashing. If the obtained sequences from these match in both versions for all the test cases, then the procedure in the newer version is considered unaffected.

By performing this process on all the benchmarks, we notice that the number of affected procedures with the optimized version of Sieve$_h$ is equal to the number of affected procedures detected using the above mentioned process. We also notice no difference in the number of affected procedures when the above process is repeated for Sieve$_m$ for all benchmarks except `gawk`. In the case of `gawk`, we observe that the number of affected procedures obtained using a version of Sieve$_m$ that employs hashing is two less than that obtained using the above mentioned matching technique. This is because of a hash collision in different versions of

two procedures that results in their versions as being flagged as equivalent (i.e., two different sequences hashing to the same hash value) when the optimized version of Sieve is used. Using more effective hash functions that decrease the probability of collisions will help to alleviate such issues.

4.3 Effectiveness of Sieve

The benchmarks obtained from [9] (`gzip`, `grep`, and `flex`) have versions for which user-defined faults can be easily introduced. Figure 8 shows a fragment from the function `updcrc` from the original base (v0) version of `gzip`. The purpose of `updcrc` is to run a set of bytes through the `crc` register and return the `crc` value.

```
3816      if (s == NULL) {
3817          c = 0xffffffffL;
3818      } else {
3819          c = crc;
3820          while (n--) {
3821            c = crc_32_tab[((int)c ^ (*s++))
                        & 0xff] ^ (c >> 8);
3822          }
3823      }
3824      crc = c;
3825      return c ^ 0xffffffffL;
```

Figure 8. Code fragment from gzip, v0(orig)

```
6085      if (s == NULL) {
6086          c = 0xffffffffL;
6087      } else {
6088          c = crc;
6089          if (n) do {
6090           c = crc_32_tab[((int)c ^ (*s++))
                       & 0xff] ^ (c >> 8);
6094          } while (n--);
6096      }
6097      crc = c;
6098      return c ^ 0xffffffffL;
```

Figure 9. Code fragment from gzip, v1(seed)

In the new version of `gzip`, the `while` loop is restructured as a `do...while` loop. The corresponding fragment is shown in Figure 9 [1]. We obtained line numbers `6090` and `6094` as affected. The flaw with the newer version, is that the corresponding lines are executed once more than the allowed limit. For many other faults (which are categorized for some benchmarks), we are able to locate the dependent regions as affected in a similar fashion.

[1] Four lines from the original source are removed from the Figure for ease of understanding. The semantics are still preserved.

5 Related Work

Law and Rothermel define the problem of dynamic impact analysis in [16] and presented a solution, PathImpact, based on whole program path profiling. In [1], Apiwattanapong *et al.* provide an efficient and precise dynamic impact analysis using execute-after sequences. They improve on existing dynamic impact analysis approaches [16, 22]. In their approach, functions that follow a modified function in some execution path are added to the impact set. One of their reasons for using dynamic impact analysis is to reduce the parts of the program that need to be retested while performing regression testing. Ren *et al.* present a technique for change impact analysis of Java programs in [25]. In their approach, a set of changes responsible for a modified test's behavior and the set of tests that are affected by a modification are identified. The differences between two versions are decomposed into a set of atomic changes and, based on static or dynamic call graph sequences, the above mentioned details are estimated.

Many static techniques also exist for performing impact analysis [2, 3]. Static analysis techniques are naturally more conservative than existing dynamic impact analysis techniques. A comprehensive discussion of the advantages of dynamic impact analysis over static analysis techniques is presented in [1]. The technique presented in this paper is more closely related to dynamic impact analysis techniques; we differ primarily in our attempt to precisely determine the regions within procedures that are affected by changes.

In [15], Law and Rothermel present a useful incremental dynamic impact analysis technique for evolving software systems. The method presented in [16] is extended to handle the problem of dynamic impact analysis across succession of system releases in an incremental fashion. In this paper, we essentially present a method to identify impacted regions in a new version compared to an existing version based on longest common subsequence. Extensions of this solution to multiple versions translate to the problem of finding longest common subsequence across multiple strings. Even though the latter problem is known to be NP-Hard, the size of the strings in the related dynamic impact analysis problem is small. Therefore, we believe that this approach is practical, and can be applied to identify modified regions across a succession of system releases.

Tools like `diff` can only identify syntactic changes across two different program versions. More sophisticated tools like MOSS [20], which are used in detecting plagiarized code fail in the presence of smartly refactored code. Horowitz identified the importance of tools that can recognize semantic changes across program versions. In [14], three different algorithms for comparing program versions by identifying various textual and semantic changes are presented. Sieve is a tool specially designed for tracking se-

mantic changes across versions. It gives qualitatively better results than `diff` or MOSS. Zhang and Gupta [28] present a novel method for matching dynamic execution histories across program versions for detecting bugs and pirated software, whereas we are interested in detecting the locations of impact within an impacted function. It is not clear if their method can be generalized for this purpose.

Many interesting techniques have been devised for bug detection in software systems [12, 17, 27, 18]. For example, in [12], Godefroid *et al.* present a technique to automatically generate test cases so that the coverage of the program is increased. In [17], the source of the software is mined to detect commonly occurring patterns and the deviants are identified as bugs. We view our contribution as a complementary technique to existing single program bug detection techniques.

6 Conclusions

This paper describes Sieve, a tool to detect variations across program versions. Sieve examines the execution of two binaries on the same test input to yield the affected functions in the newer version, along with the regions in these functions where the change manifests. Experimental results on a number of open source programs shows that Sieve reduces the size of the impact set. We also find that affected regions in the impacted functions tend to be relatively small.

7 Acknowledgements

We are grateful to the anonymous ASE referees for their detailed comments. We thank Robert Cohn of the PIN Project for providing specifics on PIN. We also thank Alessandro Orso for elucidating certain details related to impact analysis, Gregg Rothermel and Alex Kinneer for helping us with some of the benchmarks, and Cristian Ungureanu for his useful comments.

References

[1] T. Apiwattanapong, A. Orso, and M. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 432–441, 2005.

[2] R. Arnold and S. Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, July 1996.

[3] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, 1997.

[4] http://www.ncbi.nlm.nih.gov/education/ blastinfo/information3.html.

[5] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[6] B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of International Conference on Software Maintenance(ICSM)*, 2004.

[7] http://www.bzip.org.

[8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1990.

[9] H. Do, S.G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[10] *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and technology, Planning Report 02-3, May 2002.

[11] http://www.gnu.org/software/gawk/gawk.html.

[12] P. Godefroid, N. Klarslund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, Il, 2005.

[13] D. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of ACM, 24(4)*, pages 664–675, 1977.

[14] Susan Horowitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, 1990.

[15] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of 14th International Symposium on Software Reliability Engineering (ISSRE)*, 2003.

[16] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[17] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, pages 306–315, Sep, 2005.

[18] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, Chicago, Illinois, 2005.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[20] MOSS. http://www.cs.berkeley.edu/ aiken/moss.html.

[21] A. Orso. Personal communication.

[22] A. Orso, T. Apiwattanapong, and M. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137, 2003.

[23] http://www.cs.cmu.edu/ hakim/software/.

[24] M.K. Ramanathan, S. Jagannathan, and A. Grama. Trace based memory aliasing across program versions. In *FASE '06: Proceedings of the Fundamental Approaches to Software Engineering, as part of ETAPS*, pages 381–395, 2006.

[25] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, 2004.

[26] http://www.gnu.org/software/wget/.

[27] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, pages 273–288, San Francisco, CA, 2004.

[28] X. Zhang and R. Gupta. Matching execution histories of program versions. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, pages 197–206, Sep, 2005.