2005

Sieve: A Tool for Automatically Detecting Variations Across Program Versions

Murali Krishna Ramanathan

Ananth Y. Grama
Purdue University, ayg@cs.purdue.edu

Suresh Jagannathan
Purdue University, suresh@cs.purdue.edu

Report Number:
05-019

SIEVE: A TOOL FOR AUTOMATICALLY DETECTING VARIATIONS ACROSS PROGRAM VERSIONS

Murali Krishna Ramanathan
Ananth Grama
Suresh Jagannathan

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #05-019
October 2005

Sieve: A Tool for Automatically Detecting Variations Across Program Versions

Murali Krishna Ramanathan Ananth Grama Suresh Jagannathan

Department of Computer Science
Purdue University
West Lafayette, IN 47907
{rmk, ayg, suresh}@cs.purdue.edu

ABSTRACT

Revisions are an essential characteristic of large-scale software development. Software systems often undergo many revisions during their lifetime because 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 assumed invariants in the original are not violated. In order to avoid testing components that are unchanged across revisions, impact analysis is often used to identify those code blocks or functions that are affected by a change.

In this paper, we present a new solution to this general problem that uses dynamic programming on instrumented traces of different program binaries to identify longest common subsequences in the strings generated by these traces. Our formulation not only allows us to perform impact analysis, but can also be used to detect the smallest set of locations within these functions where the effect of the changes actually manifest.

Sieve is a tool that incorporates these ideas. Sieve is unobtrusive, requiring no programmer or compiler involvement to guide its behavior. We have tested Sieve on multiple versions of open-source C programs and find that the accuracy of impact analysis is improved by 10 - 30% compared to existing state-of-the-art implementations. More significantly, Sieve can identify the regions where the changes manifest, and discovers that for the vast majority of impacted functions, the locus of change is limited to often less than three lines of code. 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 [7]. Recent work on isolating and correcting software bugs [9, 14, 16, 22, 15] 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 similarities across program versions. We do so by using test results on older versions to automatically identify regions in newer versions that are affected by the changes that characterize their differences; it is precisely these regions that merit comprehensive review and testing. 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?*"

Our focus subsumes various dynamic impact analysis techniques that have been proposed previously. Execute-after sequences [2], path impact analysis [13] and coverage impact analysis [19] all attempt to identify functions that are potentially affected by a program change using program traces and test data. For example, in [2], Apiwattanapong *et. al.* describe an efficient and precise dynamic impact analysis based on the following thesis: "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 [20]. 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. Outside of the conservative approxima-

tions used to determine the set of affected functions, 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 data-flow behavior, new techniques are still required to use this information effectively to 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 imagine multiple versions of a program as analogous to a collections of mutations from an original source. One popular way to perform protein matching for the purpose of identifying mutations is to abstract it to the problem of finding an optimal alignment between two proteins by using dynamic programming. The optimal alignment problem is a dual of the popular longest common subsequence problem [6]. 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 this intuition, as a first step to detect and isolate variations in program versions, we abstract a program as a sequence of memory reads and writes. Test input is fed into two versions and the trace of memory operations is collected using binary instrumentation. A trace is a sequence of <*Operation, Value*> tuples, where *Operation* is either a read or write to memory and *Value* is the value read from or written into 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 [10] distance is calculated and the gaps [3] in the comparison recorded. (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 reverse engineering the tuples 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, then we regard the function in the newer version as unaffected by changes in the older version.

We have implemented a tool called Sieve that uses the above techniques for identifying regions of change across program versions. In our current implementation, the cost of an alphabet match is zero while the cost of a gap insertion is greater than zero. While more sophisticated cost functions can be developed based on program context, we find that even using this simple cost function leads to high efficacy. Over a range of benchmarks, the results of our experiments show a reduction of 10-30% 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 does not generate false positives: if a function in a later version is marked as impacted, there are indeed regions within that function that are influenced by changes made to the older version. However, Sieve can produce false negatives, i.e., functions which are actually affected can be undetected due to the quality of the test inputs. In this regard, it shares the limitations as other dynamic profile or test-driven techniques.

1.1 Our Contributions

This paper makes the following technical contributions:

1. **New Mechanism:** We propose a new 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 to help identify changes across program versions is needed.

2. **Improved 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 of our approach is based on the quality of the test inputs, as is the case with many comparable designs and testing methodologies.

3. **Identifying Changed Regions:** We identify the regions of code in affected functions at which the changes to the source manifest in the program.

4. **Sieve:** We have implemented a tool using our approach that has been tested on a number of realistic 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 to use it.

2. MOTIVATION

Maintaining programmer-defined invariants in large-scale software systems is challenging as the system undergoes revisions. It is often the case that when a component in such a system changes, other components are affected as well, sometimes unintentionally. Determining what these components are, and where their behavior changes, is the focus of this paper. By identifying and localizing the targets of a revision, more focussed test suites can be constructed, and programmers can more easily determine whether an intended change indeed occurred, or whether an unintended change was benign or erroneous.

Some 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, focussing test cases on those 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.

As an analogy, when comparing genes from mutations of a species, it is useful to detect exactly where a mismatch happens. This knowledge can give the biologist further insight into the characteristics of the mutation. Similarly, in our case, it is useful for a programmer to detect the locations at which changes to an older version lead to different behavior in the newer one. Armed with this knowledge, the programmer can use various slicing techniques [1, 23], for example, to comprehend the behavior of the new version isolated with respect to these changed regions. Sieve provides this degree of functionality. Our technique is similar to solutions for related problems in the area of computational biology. More specifically, sequence alignments of novel sequences with previously characterized genes can help in characterizing proteins [3]. The approach adopted to detect sequence alignments is dynamic programming. The problem of finding a maximum length subsequence of two or more strings is defined as the longest common subsequence problem. The solution to this problem [6] 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. A space is introduced into an alignment to compensate for insertions and deletions in one sequence relative to another is defined as a *gap* [3].

For example, given two strings aabcabcd and abacbd, the longest common subsequence is aacbd. One possible alignment for the example given above is as follows: 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. Gaps in an alignment 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.

3. SIEVE

3.1 Example

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

Using our approach, memory traces associated with the

```
void main(){
  ...
  old(s);
  f(s);
  g(s);
  h(s);
  ...
}

void old(LIST *s){
  LIST *t;

  t = s->next;

  while(s != NULL){

     print(s->val);
     s = s->next;
  }


  if(t->val > NUM)
     print("error");
}
```

```
void main(){
    ...
    new(s);
    f(s);
    g(s);
    h(s);
    ...
}

void new(LIST *s){
    LIST *r, *p;
    r = (LIST *)malloc(LIST);
    p = s->next;
    s->next = r;
    for(r->next = p;
        r != NULL;r=r->next){
       print(s->val);

    }

    s = delete_r_from_list(s);
    if(p->val > NUM)
      print("error");
}
```

Figure 1: Example of functions from two versions

invocation of these functions on the same test input are first obtained. Suppose the list referenced by s contains pointers to cells $\{x,y,z\}$, where x holds 10, y holds 15, and z holds 20. Furthermore, assume reference y is supplied as the argument to these functions in the test cases. The memory trace generated is shown in Figure 2.

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

 old: <R, z>, <W, z>, <R, 15>, <R, z>,
      <W, z>, <R, 20>, <R, NULL>, <W, NULL>, <R, 20>

 new: <W, q>, <R, z>, <W, z>, <R, q>,
      <W, q>, <R, 15>, <R, z>, <W, z>, <R, 20>,
      <R, NULL>, <W, NULL>, <R,  y>, <W, y>, <R,  20>
```

Figure 2: Memory Trace associated with the functions in Figure 1

By applying dynamic programming, we can match these these traces to get an optimal alignment. The alignment is shown in Figure 3. The gaps are represented by a hyphen.

Consequently, the regions in the actual source can also be aligned. Figure 1 roughly presents this alignment [1]. For example, the statement s->next = r in new does not have a corresponding statement in old. This is called a gap in sequence alignment. Similarly, other gaps are present for the newly allocated cell, and the call to delete_r_from_list. Renaming variables (e.g.,t is renamed as p), restructuring the code (e.g., the while loop is rewritten as for loop), etc., do not trigger an alignment mismatch because their effects remain unchanged.

If this were the only change in the program, our approach would identify functions new and delete_r_from_list as po-

[1]Note that s=s->next is aligned with r=r->next, though not shown aligned in the figure.

```
old: -, <R, z>, <W, z>, -, -, <R, 15>,
     <R, z>, <W, z>, <R, 20>, <R, NULL>, <W, NULL>,
     -, -,  <R, 20>

new: <W, q>, <R, z>, <W, z>, <R, q>,
     <W, q>, <R, 15>, <R, z>, <W, z>,
     <R, 20>, <R, NULL>, <W, NULL>, <R, y>,
     <W,  y>, <R,  20>
```

Figure 3: Alignment for the traces shown in Figure 2. The gap cost is 5.

tentially affected. In contrast, path impact analysis [13], 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`, `g` and `h` would be recorded as affected by these changes.

3.2 Implementation

Sieve is a tool that consists of two components viz., 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 [17], and execute on the same test input. The effect of the instrumentation yields memory traces on selective operations. These traces are then compared using dynamic programming, and optimally aligned depending on the user defined cost function. A block diagram of this process is given in Figure 4.

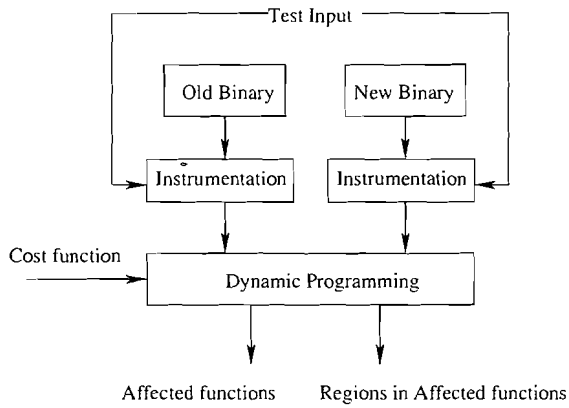


Figure 4: Block Diagram for Sieve.

Gaps in the alignment help detect operations performed by the newer version 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. Sieve employs a number of optimizations and heuristics, described below, to make comparison of complete traces practical on realistic inputs. 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. A detailed algorithm is given in Figure 5. The procedures INSTRUMENT and DYNAMIC referenced in the algorithm are given in Figures 6 and 9 respectively.

3.3 Instrumentation Tool Using PIN

procedure COMPARE
▷ **Input** B_o: Older version of a program binary
▷ **Input** B_n: Newer version of a program binary
▷ **Input** T: Set of test inputs
▷ **Output** S: Set of function tuples $< f_o, f_n >$ where f_o and f_n exactly match
▷ **Output** L: Set of $< l, f_n >$ tuples, where f_n is a function in the newer version, l is a line number in f_n

```
1  L ← {}
2  F_old is a set of function names referenced in B_o
3  F_new is a set of function names references in B_n
4  S ← {< f_o, f_n >, ∀f_o ∈ F_old, ∀f_n ∈ F_new}
5  for each t ∈ T
6      M_o ← INSTRUMENT(B_o, t)
7      M_n ← INSTRUMENT(B_n, t)
8      for each tuple < f_o ∈ F_old, f_n ∈ F_new >
9          M_fo ← Data associated with f_o
10         M_fn ← Data associated with f_n
11         Z ← DYNAMIC(M_fo, M_fn)
12         if |Z|> 0 then S ← S - < f_o, f_n >
13         L ← L ⋃ Z
```

Figure 5: Comparing two versions of a program.

We use PIN [17], 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. The PIN API provides a number of operations useful for our purposes. For example, the call `INS_IsMemoryRead(Ins)` can be used to query whether an instruction is a memory read or not. For any instruction in a binary compiled with a debug option, PIN provides a procedure that takes the address of the instruction and outputs the line number and file in the source that generated the instruction. We have used these operations in implementing Sieve's instrumentation module.

Instrumentation code can be inserted at desired locations in the binary. For our current implementation, we track all heap related operations ignoring other instructions, including reads or writes to the stack. Stack related operations are ignored for two reasons: (i) the changes in the newer version with respect to the stack operation is likely to eventually manifest itself as a change in some heap operation at some other location. Of course, the downside to this approximation is that the programmer may sometimes need to backtrack from the heap operation where a change is noticed to the actual stack operation instruction that caused the change; (ii) not tracking stack accesses reduces the overall time for instrumentation, which is the primary overhead in our experiments (see Section 4), as well as the time taken for dynamic programming. As part of future work, we intend to explore ways to instrument stack related instructions without incurring excessive cost, and to calculate the trade-off between precision and performance.

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

```
procedure INSTRUMENT
    ▷ Input B: Binary to be instrumented
    ▷ Input t: Input to the binary
    ▷ Output M: List of tuples < o, v, l, f >, where
            o is the operation (read or write)
            v is the value
            l is the line number in the source
            f is the function
1   M ← {}
2   Execute the binary B on input t using PIN.
3   for each instruction I executed
4       if I is not a memory read or write then continue
5       if I is a memory read then o ← R else o ← W
6       v ← Value being read or written to memory
7       l ← Line number of I in the source
8       f ← Function immediately enclosing I
9       M ← M + {o,v,l,f}
10  return M
```

Figure 6: Instrumenting a program binary using PIN.

```
557 void compressBlock(EState* s, Bool is_last_block)
558 {
559   if(s->nblock > 0) {
561     BZ_FINALISE_CRC(s->blockCRC);
562     s->combinedCRC = (s->combinedCRC<<1) |
                                 (s->combinedCRC>>31);
563     s->combinedCRC ^= s->blockCRC;
564     if (s->blockNo > 1) s->numZ = 0;
566     if (s->verbosity >= 2)
          ...
572     }
574     s->zbits = (UChar*) (&((UInt16*)s->arr2)[s->nblock]);
577     if (s->blockNo == 1) {
          ...
582       bsPutUChar ( s, (UChar)('0' + s->blockSize100k) );
583     }
585     if (s->nblock > 0) {
          ...
592       bsPutUInt32 ( s, s->blockCRC );
          ...
605       bsW ( s, 24, s->origPtr );
          ...
608     }
612     if (is_last_block) {
          ...
617       bsPutUInt32 ( s, s->combinedCRC );
618       if (s->verbosity >= 2)
              ...
621     }
622 }
```

Figure 7: Example of instrumentation.

input with dynamic instrumentation, a list of tuples is generated. The elements in the tuple include the type of operation (read or write), its 32 bit value (read or written), the line number and the function in which the instruction was generated. A precise description of this process is given in Figure 6.

Figure 7 shows a program fragment from the `compress.c` program in the `bzip2` benchmark. Including comments, there are approximately 55 lines in the function `compressBlock`. Most of the lines shown in the figure perform heap related operations. By instrumenting `bzip2` on a test sample, we obtain the data related to `compressBlock` shown in Figure 8. A single line in the source code can map to multiple heap related assembly instructions as shown in the figure. (The numbers shown in the left of the figure correspond to line numbers in the source.) The same function in a newer version of `bzip2` was syntactically different from the one shown above due to renaming of variables, function names and adding new variables. However, in both versions, the operations and values generated were the same.

3.4 Comparison Tool Using Dynamic Programming

The comparison module(see Figure 9) 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 considered a string, the equivalence of an alphabet in the string here is a tuple <*Operation, Value*>. A dynamic programming table is constructed with an extra row and column up front. The extra row and column contains values equivalent to the column and row indices respectively. 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} is calculated as follows. If alphabets i and j are equal, i.e., the tuples are equivalent, then the cost d_{ij}, computed in line 10 of Figure 9, is the minimum of d_{i-1j-1}, $d_{i-1j}+1$ and $d_{ij-1}+1$. 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, computed in line 10a, gives the alignment of the two traces.

To illustrate how the comparison module works, we provide a sample from the `wget` benchmark. Figure 10 shows an extract of function `make_connection` from file `connect.c` in `wget`. Syntactically, this function is the same in the two versions (1.6 and 1.7) we consider. Since this function appears in a low-level networking module, we would expect it to be reasonably insulated from changes to higher-level modules in the application. When run on a sample test input, we obtain a sequence of <*operation, value*> tuples as follows:

```
W d, R d, R 0, R d, R d
```

and

```
W d, R d, R 0, R d, R 0
```

Lines 87, 88, 90, 99 and 106 in versions 1.6 and 1.7 respectively constitute the set of heap-related operations for this function. As before, `W` denotes a write operation, `R` denotes a read operation, and `d` represents a memory location.

As is evident from Figure 10, `DEBUGP` is the cause for the difference. The definition of `DEBUGP` for both versions is shown in Figure 11 and Figure 12. As can be observed from these definitions, a new conditional variable `opt.debug` was added and this variable was set to 0. This results in a read of 0 in the newer version as compared to the unconditioned read of `*sock` in the previous version.

3.5 Heuristics

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. To make our approach scalable,

```
559 CMPL $0x00 0x00000044(eax)    R 6d3
561 MOVL 0x00000260(eax) eax      R cf5c545
561 MOVL eax 0x00000260(edx)      W f30a3aba
562 MOVL 0x00000264(eax) eax      R 0
562 MOVL eax 0x00000264(edx)      W 0
563 MOVL 0x00000260(eax) eax      R f30a3aba
563 XORL 0x00000264(edx) eax      R 0
563 MOVL eax 0x00000264(ecx)      W f30a3aba
564 CMPL $0x01 0x0000026c(eax)    R 1
566 CMPL $0x01 0x00000268(eax)    R 0
574 MOVL 0x00000044(eax) eax      R 6d3
574 ADDL 0x00000014(edx) eax      R a6589008
574 MOVL eax 0x0000002c(ecx)      W a6589dae
577 CMPL $0x01 0x0000026c(eax)    R 1
582 MOVZBL 0x00000270(eax) eax    R 9
585 CMPL $0x00 0x00000044(eax)    R 6d3
592 MOVL 0x00000260(eax) eax      R f30a3aba
605 MOVL 0x0000001c(eax) eax      R 247
617 MOVL 0x00000264(eax) eax      R f30a3aba
618 CMPL $0x01 0x00000268(eax)    R 0
```

Figure 8: Instrumentation output for the function in Figure 7: Line number, Assembly instruction, operation (R/W), and value read from or written into memory.

---

procedure DYNAMIC
▷ **Input** R: Memory trace with older version
▷ **Input** C: Memory trace with newer version
▷ **Output** U: Set of tuples $< l, f >$, where
l is the line number in the source
f is the function

```
1   U ← {}
2   for i ← 1 to | R | + 1
3    for j ← 1 to | C | + 1
4     if i = 0 then d_ij ← j
5     else if j = 0 then d_ij ← i
6     else
7       p ← user defined penalty
8       if R[i-1].o = C[j-1].o and
          R[i-1].v = C[j-1].v then
9         p ← 0
10      d_ij ← min(d_{i-1 j-1}+p, d_{i-1 j}+1, d_{i j-1}+1)
10a     z_ij ← any(diagonal, left, top) based on the result of 10
11  while(i ≠ 0 or j ≠ 0) do
12   if z_ij = diagonal then
13    i ← i - 1, j ← j - 1
14   else if z_ij = top then
15    U ← U ⋃ < C[j].l,C[j].f >, j ← j - 1
16   else U ← U ⋃ < −R[i].l,C[j].f >, i ← i - 1
17  while(i ≠ 0) do
18   U ← U ⋃ < −R[i].l,C[j].f >, i ← i - 1
19  while(j ≠ 0) do
20   U ← U ⋃ < C[j].l,C[j].f >, j ← j - 1
21  return U
```

Figure 9: Dynamic Programming

```
63 make_connection(int *sock,char *hostname,unsigned short port) {
     ...
87   if ((*sock = socket (AF_INET, SOCK_STREAM, 0)) == -1)
88      return CONSOCKERR;
90   if (opt.bind_address != NULL)
       ...
99   if(connect(*sock,(struct sockaddr *)&sock_name,sizeof(sock_name)))
       ...
106  DEBUGP(("Created fd \%d.\n", *sock ));
       ...
108 }
```

Figure 10: Program fragment of function `make_connection` in `connect.c` from `wget`.

```
/* Print X if debugging is enabled; a no-op otherwise.  */
#ifdef DEBUG
# define DEBUGP(x) do { debug_logprintf x; } while(0)
#else  /* not DEBUG */
# define DEBUGP(x) DO_NOTHING
#endif /*not DEBUG */
```

Figure 11: Definition of DEBUGP in wget.h (version 1.6).

we employ a heuristic that performs dynamic programming piecemeal to smaller substrings.

The heuristic is based on the following observation. If two functions are unrelated, then their memory traces are likely to yield large gaps as an alignment is computed. If the functions are related, i.e., one is a version derived from the other, then there are likely to be relatively few gaps in the alignment of their respective traces; in other words, there is likely to be sufficient locality to apply dynamic programming on the strings yielded by subtraces to yield a good, if not necessarily optimal, alignment.

More precisely, our heuristic works as follows:

1. Obtain a prefix of fixed length r from both traces.
2. Apply dynamic programming on the prefixes obtained.
3. Find the farthest location in each prefix respectively after which there is no alignment between the prefixes.
4. Obtain a prefix of r starting from these locations respectively from each trace and repeat the process from Step 2.

We use the example from Section 2 to explain the heuristic. Recall that the two strings being compared are `aabcabcd` and `abacbd`. Fix r to be three. In the first step, prefixes `aab` and `aba` are extracted. Aligning these prefixes, we get `aab-` and `-aba`. In the next step, we extract `cab` from the first string and `acb` from the second string. Aligning the prefixes, we get `-cab` and `ac-b`. Subsequently, we extract `cd` and `d`

```
/* Print X if debugging is enabled; a no-op otherwise.  */
#ifdef DEBUG
# define DEBUGP(x) do {if (opt.debug) {debug_logprintf x;}}while (0)
#else  /* not DEBUG */
# define DEBUGP(x) DO_NOTHING
#endif /* not DEBUG */
```

Figure 12: Definition of DEBUGP in wget.h (version 1.7).

and align them as `cd` and `-d` respectively. The final alignment is `aab-cabcd` and `-abac-b-d`. Compare this alignment with the alignment (`a-abcabcd` and `aba-c-b-d`) obtained in Section 2 using the normal process. Coincidentally, in this case we have also obtained an optimal alignment (i.e., an alignment with the smallest number of gaps) using our heuristic. In general the alignment obtained through this heuristic is not always optimal. However, we show in Section 4 that this heuristic performs surprisingly well for all the benchmarks we consider.

4. EVALUATION

4.1 Experimental Setup

We have examined Sieve using two versions of the following software packages: `bzip2` [5], `bunzip2` [5], `gawk` [8], `htmldoc` [12] and `wget` [21]. 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 1. We explain the significance of the other columns below. The test cases used for the benchmarks are either randomly generated or are from standard test suites available for them.

We perform our tests on Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on a Intel(R) Pentium(R) 4 CPU 3.00GHz with 1GB memory. The version of the PIN [17] tool used was a special release 1819 (2005-04-15) for Gentoo Linux. The sources were compiled using GCC version 3.3.4.

4.2 Results

To improve the analysis time of the current implementation, a list of functions that need to be instrumented and the pair of functions to be considered for comparison are also provided. The number of memory reads and writes, the associated values yielded, and the line in the source responsible for such an action is given as output of the instrumented program executed under PIN. By performing this process for both versions, we have two traces of heap reads and writes, and corresponding information that is provided as input to the comparison module.

Unless otherwise stated, the results are obtained using blocks of size equal to 50 (i.e., the length r defined in the heuristic in Section 3.5) in the dynamic programming process. On performing the whole process as mentioned above for each test case, we obtain the regions (in the form of line numbers) in the newer version that differ from the older version.

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

- If a function is impacted, what regions in the function are affected?
- Is there any reduction in the number of impacted functions reported using our approach compared to state-of-the-art techniques?
- How does the heuristic of varying the block size affect the accuracy and performance of our approach?
- What are the performance overheads viz., memory and time taken associated with our technique?

Figure 13(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 45% of all functions perform fewer than six operations to the heap, and in `wget` roughly 15% of all functions perform more than 18 operations involving the heap.

Figure 13(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 `bzip2`, we observe that over 60% of all impacted functions have changes limited to three or fewer lines of code. Indeed, for all the applications in our benchmark suite, greater 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.

Figure 13(c) shows the cumulative effect of Figures 13(a) and 13(b). It gives details on the percentage of code within impacted functions that are influenced by changes due to revisions between versions. For example, in `gawk`, roughly 20% of all impacted functions had changes that affected less than 15% of their code. On the other hand, in `bunzip2` nearly 75% of all impacted functions had changes that were manifest within less than 30% of their code size.

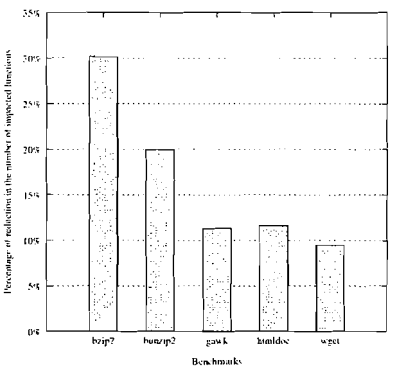


Figure 14: The percentage of functions that were found to be not impacted by our approach as compared to path impact analysis.

Figure 14 presents the reduction in the number of functions found to be impacted using our approach as compared to a state-of-the-art impact analysis [2]. The number of impacted functions identified by our approach range from 24 for `bunzip2` to 298 for `gawk`.

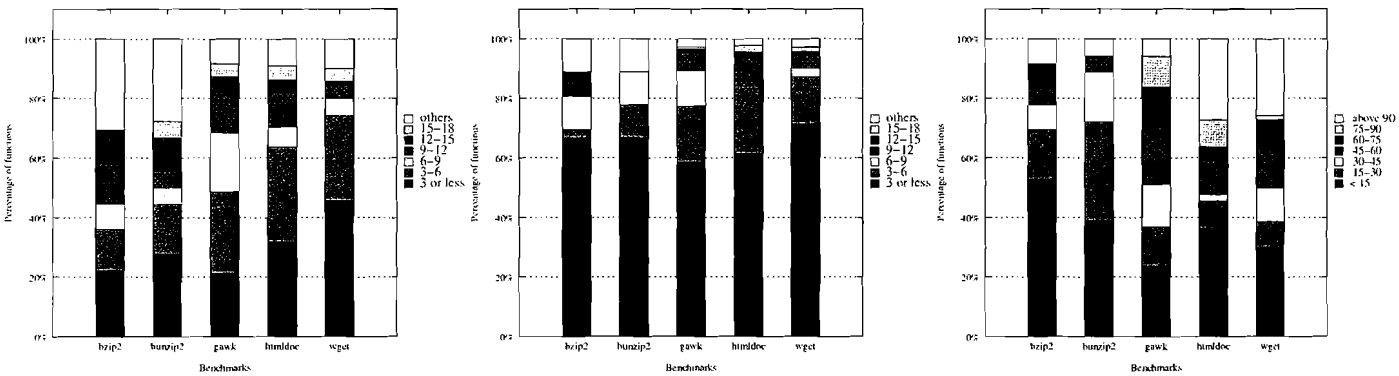
To quantify Sieve's utility, we implemented path impact analysis as described in [2] for C programs. Typically, the functions are compared across versions and marked as (un)changed. A function that follows a changed function in any execution is labeled as affected. A reduction from 10% to 30% in the size of the impacted set was observed across our benchmark set when comparing our technique with this strategy.

The implication of this result is that the focus of regression testing can be improved because 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 analyzes that do not leverage this degree of precision.

In Table 1, we provide the specifics of our benchmarks and the results obtained using our technique. The number of lines of code varies from 9K to 65K with the number

Benchmark	Old Version	New Version	LoC (in K)	Total Functions	Longest Trace (10^3)	Total Tests	Instr. Time	Analysis Time	Memory (in MB)	% affected	
										Static	Dynamic
bzip2	0.9.5d	1.0.2	9	107	6099	107	2600	591	351	25.4	31.8
bunzip2	0.9.5d	1.0.2	9	107	1839	107	1341	181	89	26.6	13.6
gawk	3.1.3	3.1.4	41	522	3598	133	1408	88	670	41.7	25.7
htmldoc	1.8.23	1.8.24	65	246	1399	138	4474	646	84	48.4	84.1
wget	1.6	1.7	28	313	158	207	954	17	16	44.4	33.6

Table 1: Benchmark Information and Results (Time in seconds).



(a) Total heap reads/write instructions per function (b) Impacted heap reads/write instructions per function (c) Percentage of an impacted function changed

Figure 13: Histogram (a) shows that most functions in these benchmarks perform a non-trivial number of heap-related operations. Histogram (b) shows that for approximately 65 % of the functions in every benchmark, three or fewer lines within these functions are impacted; Histogram (c) is a combination of (a) and (b). It shows the percentage of change that occurs within impacted functions.

of functions varying from 100 to 500 approximately. The length of the trace represents the number of reads and writes to the heap in thousands of instructions. The longest trace observed was approximately 6 million for `bzip2`. The average memory used while significant is not problematic. This is expected for many dynamic analysis scenarios because precise information on heap operations is being gathered. The percentage of affected regions is also provided in the table. The static percentage reveals that a sizeable fraction of the newer version of a benchmark program is impacted by changes to the older, even though Fig 13 demonstrates that the absolute number of lines where the changes manifest is small in the majority of the cases. The dynamic percentage shows that in some cases (e.g., `htmldoc`), these changes are exercised often.

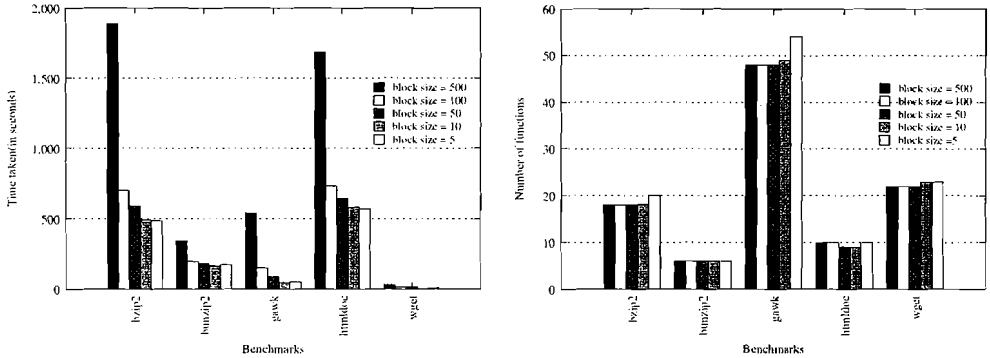
The time taken for our technique is composed of the instrumentation time of the binary and execution time of comparison module. It is obvious from the table that the main performance bottleneck is associated with instrumentation time. There are two reasons for the inefficiency of the instrumentation process. The first is because we use a dynamic binary instrumentation tool as opposed to static instrumentation. Therefore for each test case, time is taken to insert appropriate instrumentation code. We believe the time taken for this approach can be significantly reduced using alternative instrumentation strategies. Furthermore, Sieve currently tracks all heap related operations. This number can also play an important role in increasing instrumentation time. A correlation is present between the length of the trace and instrumentation time. For example, `wget` has a shorter trace and thus significantly smaller instrumentation time compared to `bzip2`. One way to reduce the number of heap operations tracked is to discard those operations found in regions already known to have been affected from previous test runs. In any case, the time taken for dynamic programming, the heart of our approach, is only a small fraction of the instrumentation time.

As discussed in Section 3.5, the accuracy and performance of our approach varies based on the block size (the prefix r in the heuristic description). Figure 15(a) shows the time taken for dynamic programming for different block sizes for each benchmark. Since the instrumentation time is independent of the block size, it is not shown in the figure. With decrease in the block size, the time taken to complete also decreases. However, a tradeoff exists between the performance and accuracy with respect to block size. As can be observed from Figure 15(b), the accuracy of our approach gradually decreases with decrease in block size. In the figure, the number of functions in the newer version that exactly match with their older counterparts is given. The number of impacted functions of our approach is the difference between the number of impacted functions using a generic impact analysis and the number of functions that exactly match. Based on the above results, we use blocks of size 50 for our experiments as it provides efficient execution times without sacrificing accuracy.

5. LIMITATIONS

We discuss two limitations in the current version of Sieve.

Aliasing: In our current implementation, we do not consider the memory addresses from which values are being



(a) Time taken for the dynamic programming process for different block sizes

(b) Number of (unaffected) functions that exactly match w.r.t. the memory trace with functions in the older version.

Figure 15: With decrease in block size, the time taken for dynamic programming reduces as seen in (a). However, a drop in accuracy is also noticed for block sizes less than 50 as shown in (b).

read/written. For example, multiple writes of value v into the same memory location in one program will be found equivalent to the same number of writes of v into consecutive (distinct) memory locations in another program. This is because only the operation performed and the value read or written are taken into account in the matching process: no consideration is given to the locations being effected. Zhang and Gupta present a work around to this problem in [24] in a related context. We intend to investigate the applicability of their approach, as well as other refinements to the comparison module, as part of Sieve's future development.

Instrumentation: As explained earlier, instrumenting the programs using a dynamic instrumentation tool seems to be a bottleneck. Currently, for each test case, instrumentation is added on the fly and the instrumented code is executed. The number of times the instrumentation is added is directly proportional to the number of test inputs. By using a static instrumentation tool, we believe that the time taken for instrumentation can be significantly reduced.

6. RELATED WORK

In [2], Apiwattanapong *et. al.* provide an efficient and precise dynamic impact analysis using execute-after sequences. They improve on existing dynamic impact analysis approaches [13, 19]. 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 tool for change impact analysis of Java programs in [20]. 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. We share obvious similarities with these efforts, but differ both in the mechanisms used to identify impacted functions, and the ability to identify localized regions of change within these functions. Moreover, because our technique operates over binary execution, we are not reliant on program analysis of input sources or programmer annotations.

Zhang and Gupta [24] present a novel method for matching dynamic execution histories across program versions for detecting bugs and pirated softwares. They perform matching by looking at the control flow taken, values produced, addresses referenced and data dependencies exercised. In contrast, we abstract programs as a sequence of read and write operations into the heap and perform the comparison of two versions using a dynamic programming approach. Moreover, 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.

Dynamic programming, more specifically longest common subsequence techniques, are used in many applications. One such application in software engineering is described in [4]. The foundation of their approach is based on the thesis that for similar bugs, the call stack also shares similarities. Therefore, by pruning unnecessary information from the call stack, and comparing the resulting string representation with an existing signature, a score can be given to the match using a longest common subsequence algorithm. The similarity between their approach and ours is restricted to the underlying technique and its applicability in a software engineering context, but does not extend to impact analysis or variation detection across program revisions.

Trivially, tools like `diff` can only identify the syntactic changes across two different program versions. More sophisticated tools like MOSS [18] that 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 [11], she presents three different algorithms for comparing program versions by identifying various textual and semantic changes. Sieve is a tool specially designed for tracking semantic changes across versions and we believe gives qualitatively better results than `diff` or MOSS. Our experiments with binary versions of realistic programs shows that our method is practical.

Many interesting techniques have been devised for bug detection in software systems [9, 14, 16, 22, 15]. For example,

in [9], Godefroid *et. al.* present a technique to automatically generate test cases so that the coverage of the program is increased. In [14], the source of the software is mined to detect commonly occurring patterns and the deviants are identified as bugs. Our work focusses on an entirely new dimension – how to detect impacted regions in a revision of a program, which by implication can help in detecting whether the impact was by design or accidental. We view our contribution as a complementary technique to existing single program bug detection techniques.

7. CONCLUSIONS

This paper describes Sieve, a tool to detect variations between 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. This information can be used for debugging, and improved regression testing. Experimental results on a number of open source programs shows that Sieve improves the quality of impact analysis by 10-30% compared to existing approaches. We also find that affected regions tend to be relatively small. Besides addressing the limitations given in Section 5, we also plan to explore other interesting avenues for future work including integrating Sieve with existing dynamic program analysis and testing frameworks like DART [9].

8. ACKNOWLEDGEMENTS

We thank Robert Cohn of the PIN Project for answering the specifics on PIN. We also thank Alessandro Orso for clarifying certain details related to impact analysis. We thank Cristian Ungureanu for his useful comments.

REFERENCES

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990.

[2] 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.

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

[4] M. Brodie, S. Ma, G. Lohman, T. Syeda-Mahmood, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. An architecture for quickly detecting known software problems. In *ICAC 2005: Proceedings of the International Conference on Autonomic Computing*, 2005.

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

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

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

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

[9] 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.

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

[11] Susan Horwitz. 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.

[12] http://www.htmldoc.org/.

[13] 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.

[14] 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.

[15] 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.

[16] B. Livshits and T. Zimmermann. Dynamine: a framework for finding common bugs by mining software revision histories. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, Sep, 2005.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI '04: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, 2004.

[24] 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.