Towards Locating Execution Omission Errors

Xiangyu Zhang †‡
†Purdue University, Department of Computer Science,
West Lafayette, Indiana 47907
xyzhang@cs.purdue.edu

Sriraman Tallam Neelam Gupta
Rajiv Gupta
‡The University of Arizona, Department of Computer
Science, Tucson, Arizona 85721
{tmsriram,ngupta,gupta}@cs.arizona.edu

Abstract

Execution omission errors are known to be difficult to locate using dynamic analysis. These errors lead to a failure at runtime because of the omission of execution of some statements that would have been executed if the program had no errors. Since dynamic analysis is typically designed to focus on dynamic information arising from executed statements, and statements whose execution is omitted do not produce dynamic information, detection of execution omission errors becomes a challenging task. For example, while dynamic slices are very effective in capturing faulty code for other types of errors, they fail to capture faulty code in presence of execution omission errors. To address this issue relevant slices have been defined to consider certain static dependences (called potential dependences) in addition to dynamic dependences. However, due to the conservative nature of static analysis, overly large slices are produced. In this paper, we propose a *fully dynamic* solution to locating execution omission errors using dynamic slices. We introduce the notion of *implicit dependences* which are dependences that are normally invisible to dynamic slicing due to the omission of execution of some statements. We design a dynamic method that forces the execution of the omitted code by switching outcomes of relevant predicates such that those implicit dependences are exposed and become available for dynamic slicing. Dynamic slices can be computed and effectively pruned to produce fault candidate sets containing the execution omission errors. We solve two main problems: verifying the existence of a single implicit dependence through predicate switching, and recovering the implicit dependences in a demand driven manner such that a small number of verifications are required before the root cause is captured. Our experiments show that the proposed technique is highly effective in capturing execution omission errors.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Testing tools, Tracing; D.3.4 [*Programming Languages*]: Processors—Debuggers

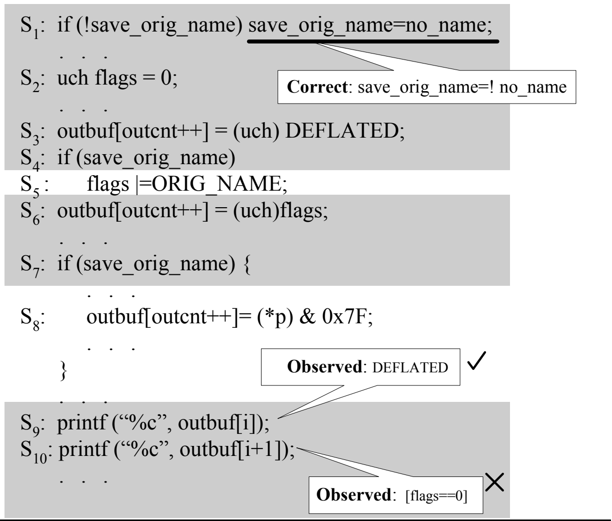
General Terms Algorithms, Measurement, Reliability

Keywords debugging, execution omission, relevant slicing, implicit dependence, potential dependence, and predicate switching.

1. Introduction

Once a software error manifests itself at runtime as a failure, dynamic program analysis, by observing the runtime symptoms of the failure, serves as a very natural means for identifying the error. In recent years, a wide variety of dynamic analysis have been proposed for the purpose of debugging runtime errors [11, 9, 13, 7, 5, 10, 4, 6, 17]. Many employ machine learning or statistical techniques to observe runtime deviations from certain invariants and raise alarms for any anomalies. Others try to understand bugs by searching the program state space. These techniques typically produce a fault candidate set, which is basically a prioritized set of statements that includes the faulty code. Proposed as a debugging aid, dynamic slicing [8] has the advantage of capturing the cause-effect relations between the faulty code and the failure through dynamic dependences. Recently, it has been shown that dynamic slicing based techniques are quite effective in locating program errors [19, 20].

Common situations in which a dynamic slice captures a faulty statement are those where execution of a faulty statement corrupts part of the program state, the effects of this corruption are propagated along dynamic dependences, and eventually a failure is detected due to the output of an incorrect value or the crashing of the program. However, an error may cause the execution of certain critical statements to be omitted, resulting in a failure. This type of error is called **execution omission error**. Dynamic slicing fails to capture execution omission errors.



```
S1:  if (!save_orig_name) save_orig_name=no_name;
     . . .
S2:  uch flags = 0;
     . . .
S3:  outbuf[outcnt++] = (uch) DEFLATED;
S4:  if (save_orig_name)
S5 :      flags |=ORIG_NAME;
S6:  outbuf[outcnt++] = (uch)flags;
     . . .
S7:  if (save_orig_name) {
          . . .
S8:       outbuf[outcnt++]= (*p) & 0x7F;
          . . .
     }
     . . .
S9:  printf ("%c", outbuf[i]);
S10: printf ("%c", outbuf[i+1]);
     . . .
```

Correct: save_orig_name=! no_name
Observed: DEFLATED ✓
Observed: [flags==0] ✗

Figure 1. An example of execution emission error (gzip).

Figure 1 gives such an example. The code is taken from version `v3` run `r1` of `gzip` available at the website [1]. The shaded statements are the ones that get executed. As shown by the figure, the error resides in the assignment to `save_orig_name`. Since `save_orig_name` contains the wrong value `false`, branch S_4 is not taken and thus `flags` has the wrong value `0` while it should have been defined as `ORIG_NAME` at S_5. This wrong `flags` value is eventually observed at S_{10}. Let us assume we try to locate the error by computing the dynamic slice of the wrong value, i.e. find the set of executed statements that affect the wrong value through data and/or control dependence. The resulting dynamic slice contains $\{S_2, S_3, S_6, S_{10}\}$, from which the root cause, S_1, is missing. This happens because dynamic slicing is able to capture the fact that the definition at S_2 reaches S_6 but fails to discover that the branch outcome of S_4 also affects the value of `flags` at S_6. The difficulty of analyzing an execution omission error is inherent limitation of dynamic analysis as the analysis is based on information collected from executed statements, not the ones whose execution was incorrectly omitted.

To the best of our knowledge, *relevant slicing* [3] is the first and also the only automatic technique which tries to tackle the problem of debugging execution omission errors. The basic idea of relevant slicing is to introduce a *potential dependence* between predicate S_4 and assignment S_6. Now the root cause becomes reachable from the wrong output through dependence edges such that it is included in the dynamic slice. Potential dependence is essentially a *static* concept in the sense that an edge is conservatively added if potentially there is a dependence. For example, S_{10} potentially depends on S_7 as well because the definition at S_8 could reach S_{10}, even though in fact S_7 does not affect the value at S_{10}. Furthermore, potential dependence edges are uniformly introduced for all the nodes in a dynamic dependence graph before any relevant slices can be computed. Therefore, the conservative effects accumulate resulting in much larger slices being computed.

In this paper, we propose a purely dynamic solution for handling execution omission errors. The essence of this solution lies in adding a dependence edge between the *predicate* and the *use* only if the dependence *can be made observable* rather than just being *potential*. A dependence is observable only if changing the value at the predicate affects the use. Therefore, the basic idea of our approach is as follows. Given a predicate `p` and a use `u` such that `p` precedes `u`, and `u` does not directly or indirectly data/control depend on `p`, we reexecute the program with the same input and then *switch* the branch outcome of the predicate. If the point that corresponds to `u` is affected, we conclude that there is a dependence between `p` and `u` *in the original execution*. We also refer to such dependence as an *implicit dependence* because even though this dependence exists, it is not established via a chain of explicit data and/or control dependences. Let us consider the example in Figure 1. To verify if there is a dependence between S_4 and S_6, we reexecute `gzip` with the same input, and then switch the execution to take the `true` branch at S_4. We observe that the value at S_6 is altered by switching the predicate. Thus, we conclude that there is a dependence between S_4 and S_6. Likewise, we find that there is no dependence between S_7 and S_{10}.

In order to realize the above idea, we need to address two major issues. The first issue is to *recognize whether there exists an implicit dependence given a use and a predicate*. We introduce the concept of *implicit dependence*, which is defined by observing the switched execution. The main difference between implicit dependence and potential dependence is that the former is *observable* while the latter is only *potential*. The key challenge of this issue is *to identify the point in the switched execution that corresponds to* `u`. Switching the predicate may significantly alter the execution, and hence finding the corresponding instance of `u` in the new execution is no longer straightforward. For example, switching the predicate at S_7 may change the control flow of the execution so much that S_{10} is not even reached or the same use of `outbuf[i+1]` may occur at a different execution point. In such cases it becomes unclear how to observe whether the original value is affected by switching the predicate or not. We propose a novel execution alignment algorithm which is able to align two executions based upon execution regions instead of individual executed statements such that we are able to find the point in the new execution that corresponds to a particular point in the original execution if there is one.

The second issue is to *efficiently discover implicit dependences such that the root cause can be found with a minimal number of reexecutions*. Verifying one implicit dependence requires reexecuting the program once. Potentially, we need to verify the dependences between any use and each of its preceding predicates, which is apparently not an option due to the prohibitively high runtime overhead. We propose a *demand driven* strategy which gradually adds new statements to the dynamic slice through detected implicit dependences, and in the mean time the new dynamic slice is pruned using a technique developed in [19, 20].

The contributions of this paper are as follows:

- We introduce the concept of *implicit dependence*. It compensates for the inability of traditional dynamic data and control dependences to capture the execution of omission errors. Since implicit dependence is a completely *dynamic concept*, it is a more appropriate basis for dynamic analysis than the hybrid concept of *potential dependence* in [3].
- We propose a demand driven strategy to reduce the overhead caused by reexecutions that are required to detect implicit dependences. This demand driven strategy is integrated with a pruning technique introduced in [19] to achieve minimal increase in the size of the fault candidate set.
- We experimentally evaluate our technique. The results show that execution omission errors can be captured by performing a small number of verification and adding very few implicit dependence edges.

The remainder of the paper is organized as follows. In section 2 we discuss the existing solution – *relevant slicing* and describe its limitations. In section 3 we present the proposed technique in detail. Experimental results are presented in section 4. Some issues about the technique are discussed in section 5. Related work is described in section 6 and conclusions are given in section 7.

2. Relevant Slicing

Relevant slicing [3] was proposed to handle execution omission errors. Given a failed execution, relevant slicing first constructs a dynamic dependence graph in the same way as classic dynamic slicing does. In addition, it augments the dynamic dependence graph with *potential dependence* edges. A relevant slice is computed by taking the transitive closure of the wrong output on the augmented dynamic dependence graph.

In relevant slicing, a use `u` *potentially* depends on a preceding predicate `p` if a different definition could potentially reach `u` if `p` were to evaluate differently, i.e. take the opposite branch. A more formal definition of potential dependence is given below:

DEFINITION 1. *Given a use* `u`*, the* potentially depends *set* `PD(u)` *contains members of the form that specify predicates and their outcomes (i.e.,* p^T *or* p^F*). Predicate* p^T *(*p^F*) presents in* `PD(u)` *iff*

(i) the execution of `p` *precedes that of* `u`*;*
(ii) `u` *is not control dependent on* `p`*;*
(iii) the definition reaching `u` *occurs before* `p`*;*

(iv) a different definition could potentially reach `u` *if* `p` *were to evaluate to* `F` *(*`T`*).*

Dependence edges are added between any `u` and each element in `PD(u)`. For the example in Figure 1, the potential dependence edges are as follows.

$$PD(outbuf[i+1]@S_{10}) = \{save\_orig\_name@S_7^T\}$$
$$PD(flags@S_6) = \{save\_orig\_name@S_4^T\}$$

Conditions (i), (ii) and (iv) are straightforward. The third condition excludes the case illustrated below, in which `p` takes the `F` branch. Since `x@4`, the reaching definition of use `x@6`, occurs after predicate `p@1`, definition `x@2` would get killed even if `p@1` were evaluated to `T`. Therefore, $\texttt{p@1}^T$ is not in `PD(x@6)`.

```
1:  if (p) {
2:        x=...;
3:  }
4:  x=...;
5:  . . .;
6:  ...=x...;
```

In Figure 1, three potential dependences are added: $S_4 \rightarrow S_6$, $S_7 \rightarrow S_9$, and $S_7 \rightarrow S_{10}$. The relevant slice of the wrong value is computed as $\{S_1, S_2, S_3, S_4, S_6, S_7, S_{10}\}$, which contains the root cause S_1.

Despite the fact that relevant slicing reveals a very promising direction to overcome the problem of debugging execution omission errors, it has its inherent limitations. First of all, condition (iv) in Definition 1 implies that static points-to analysis has to be conducted to disclose possible reaching definitions for a use. The conservative nature of such a static analysis inevitably gives rise to false dependences. In our motivation example, since `DEFLATED` and `flags` are printed at S_9 and S_{10} respectively, the definition at S_8 can only reach some output statement which gets executed after S_{10}. In other words, the definition at S_8 can never reach S_9 and S_{10}. Unfortunately, a static points-to analysis fails to capture this fact and hence false dependences such as $S_7 \rightarrow S_9$ and $S_7 \rightarrow S_{10}$ are introduced.

Second, the effects of the conservative nature of static analysis accumulate. The initial introduction of false dependences creates new opportunities to bring in more and more false dependences. Eventually, the excessively expanded dependence graph might result in over sized relevant slices. In some earlier studies [3, 15], it was reported that relevant slicing only inflates the sizes of classic dynamic slices by a very small ratio. However, those study only reported the number of unique static statements in a relevant slice, which may barely increase. In contrast, the number of dynamic statements often increases by orders of magnitude. For example, let us assume an erroneous predicate is executed for `100` times and the fault gets exercised at the last execution instance. Ideally, the slice should contain only the faulty instance. However, relevant slicing often includes all the `100` instances due to its static nature. Therefore, even though the number of static statements only increases by one, the information that the programmer has to go through in order to figure out the failure inducing relations may increase by a much larger factor.

3. Dynamic Detection of Implicit Dependences

As pointed out earlier, relevant slicing is a hybrid analysis which has both static and dynamic components. The conservative nature of static analysis inevitably results in over sized slices, which is contrary to the goal of providing a minimal fault candidate set containing the root cause. In this paper, we propose an aggressive solution. The essence of this solution lies in adding a dependence edge between a predicate and a use only if the dependence is *observable* rather than just being *potential*. To test whether a dependence is observable, we switch the branch outcome of the predicate in a second execution and then observe if the use is affected.

As mentioned earlier, we are confronted by two challenges: *how to recognize whether there exists an implicit dependence given a use and a predicate* and *how to efficiently discover implicit dependences such that the root cause can be found with a minimal number of reexecutions*. In this section, we describe how these problems are solved.

3.1 Implicit Dependence

In classical dynamic slicing, a data dependence represents the flow of a value from the statement execution that defines it to the statement execution that uses it such as the dependence between $\texttt{1}^{(1)}$ and $\texttt{6}^{(1)}$ in Figure 2, denoted as $\texttt{1}^{(1)} \xrightarrow{dd} \texttt{6}^{(1)}$. A control dependence between two statement executions represents that the execution of one statement depends on the branch outcome of a predicate in the other statement. For example, statement execution 14 is control dependent on 13 in execution (1), denoted as $13^{(1)} \xrightarrow{cd} 14^{(1)}$. These dependences are *explicit*, or in other words, they are observable and captured during the program execution.

Execution omission errors happen when certain code should have been executed while it did not due to the error. The barrier of locating an execution omission error lies in the dependences that are essential to the failure are not explicit from the execution. For example in Figure 2, the dependence from 15 to 2 in execution (1) is *implicit*. We have to capture this type of dependence in order to handle execution omission errors. Before we proceed, let us first introduce the definition of this type of dependence.

Theoretically, we can define a dependence exists between two statement executions if and only if disturbing the execution of one statement affects the execution of the other. This definition subsumes the previous definitions of data dependence and control dependence. However, tracking such dependence at runtime is not possible as information required must come from statements that are not executed. Next we define the new type of dependence, *implicit dependence*, and show how to detect them.

DEFINITION 2. *Given an execution E, a predicate p, and a use u s.t. there is no explicit dependence path between p and u, let E' be the reexecution of the same program with the same input as E except the branch outcome of p' being switched, p' and u' be the execution points in E' that match p and u in E, respectively, u* **implicitly** *depends on p, denoted as $p \xrightarrow{id} u$, iff*
(i) u' is not found in E', or,
(ii) there is an explicit dependence path between p' and u',

Explicit dependence is a dependence that can be observed during the execution including data dependence and control dependence. Note that *implicit* dependence is defined in terms of `p` and `u` in the original execution even though it is verified in the switched execution.

The key challenge here is to find the point during the switched execution that corresponds to `u`, a use in the original execution. Let us illustrate the problem using an example in Figure 2. The source code is presented in the left column and execution traces are presented in the right column. Let us assume that we are interested in verifying the dependence between the use of `x` at 15 in execution (1), denoted as $\texttt{15}^{(1)}$, and predicate `P` at $\texttt{2}^{(1)}$. According to our solution, we reexecute the program with the same input (the new execution is referred to as execution (2) in Figure 2 and then we switch the branch outcome of `P` at statement $\texttt{2}^{(2)}$. In order to find out whether switching `P` affects the use of `x`, we first need to find the corresponding use in execution (2).

```
1:  i = t = x = P = C1 = C2 = 0;
2:  if (P) {
3:        t = 1;
4:        x=...;
5:  }
6:  while (i < t) {
7:        ...;
8:        if (C1) {
9:              ...;
10:       }
11:       i = i + 1;
12: }
13: if (...) {
14:       if (C2 == 0) {
15:             ...=x;
16:       }
17:       ...;
18: }
```

(1) Original execution trace:
1, 2, 6, 13, 14, 15, 16, 17, 18.

(2) Execution trace with the branch outcome of P being switched:
1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 6, 13, 14, 15, 16, 17, 18.

(3) Execution trace with P switched and statement 3 is $t = C_2 = 1$:
1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 6, 13, 14, 17, 18.

Matching 15 in traces (1) and (2):

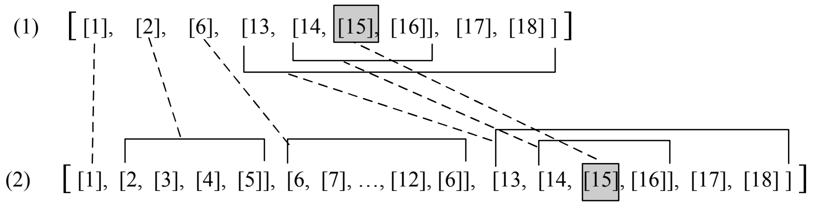


Figure 2. An example of implicit dependence

In general, this problem is undecidable because deciding whether the execution terminates after switching is a undecidable problem. Nonetheless, we are more interested in the cases that switched executions terminate. Therefore, in this paper we assume *switched executions always terminate*. In practice, we set a timer which if expires, we aggressively conclude the verification fails and thus there is no dependence between the predicate and the use.

Even with the assumption of termination of the switched execution, finding the corresponding instance of `u` in the *switched execution* is not easy. In execution (2) of the example, switching `P` changes `t`, resulting in the statements inside the `while` loop to be executed. Let us assume statement 7 makes a recursive self call and then the resulting trace has the following form:

1, 2, 3, 4, 5, 6, 7, **1, 2, ..., 15,...** 8, 11, 12, 6, 13, 14, 15, 16, 17, 18.
A simple strategy that looks for the first appearance of 15 after statement 2 does not work in this case.

In a more complicated example such as execution (3) in Figure 2, $\texttt{15}^{(3)}$ does not get executed as the result of `P` at $\texttt{2}^{(3)}$ getting flipped. Let us further assume statement $\texttt{17}^{(3)}$ recursively calls the function and thus the trace produced becomes the following:

1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 6, 13, 14, 17, **1, 2, ..., 15, ...** 18.
Apparently, a well designed matching algorithm should clearly recognize that there is no match of the original $\texttt{15}^{(1)}$ in the new run rather than associating 15 in this run with $\texttt{15}^{(1)}$.

The difficulty stems from the fact that we were trying to align individual statement executions while statement executions may vary significantly due to the predicate switch. We observe that the two executions can actually be aligned in larger units. For instance in Figure 2, statement $\texttt{6}^{(1)}$ corresponds to or *matches* the sequence of executed statements $[\texttt{6,7,8,11,12,6}]^{(2)}$ in the switched execution. In other words, rather than matching the executions *inside* the `while` statements, we align them as a whole instead. Similarly, sequence $[\texttt{13,14,15,16,17,18}]^{(1)}$ matches $[\texttt{13,14,17,18}]^{(3)}$, and then $[\texttt{14,15,16}]^{(1)}$ matches $\texttt{14}^{(3)}$. Therefore, we conclude that $\texttt{15}^{(1)}$ has no corresponding match in execution (3).

Next we develop the algorithm for the above matching process. We begin by defining the concept of a region which is the basis for performing matching.

DEFINITION 3. *An execution can be decomposed into* **regions**. *A statement execution* `s` *and the statement executions that are control dependent on* `s` *form a region. The following grammar captures the form of a region.*

$$\begin{aligned} \textit{Region} &::= \texttt{s}\ \textit{CD} \\ \textit{CD} &::= \epsilon \ \mid\ \textit{Region} \ \mid\ \textit{CD Region} \end{aligned} \qquad (1)$$

where s is a statement execution and given any statement execution $e \in CD$, e is control dependent on s.

For instance, $[\texttt{13,14,15,16,17,18}]^{(1)}$ forms a region, inside which $[\texttt{14,15,16}]^{(1)}$ forms another region.

Given the definition of region, the algorithm is described by Algorithm 1. `E` and `E'` denote the original execution and the switched execution, respectively. The algorithm finds `u'` in `E'` that matches `u` given the matching predicates of `p` and `p'`. In the body of `Match()`, the algorithm first tries to identify the immediate surrounding region `r` that includes both `p` and `u` and its corresponding instance `r'` in `E'`. `Region(s/r)` gives the immediate surrounding region of a statement execution `s` or a region `r`. For example, `Region([15])` = `[14,15,16]` in execution (1). Since the two executions are identical till they reach the points of `p` and `p'`, we can easily align `r` and `r'`, the surrounding regions of `p` and `p'`, respectively. Next the algorithm calls `MatchInsideRegion()` to match `u` given `r` and `r'`. In `MatchInsideRegion()`, the algorithm begins with matching the first immediate subregions. `FirstSubRegion(r)` returns the first immediate subregion that is inside `r`. For instance, `FirstSubRegion` $(\texttt{[2, 3, 4, 5]}^{(2)})$ = $\texttt{[3]}^{(2)}$. The `while` loop in lines 17-20 traverses through the siblings to locate the subregion `r` that contains `u`.

For instance, the sibling region of $\texttt{[3]}^{(2)}$ is $\texttt{[4]}^{(2)}$. Given the matching subregions `r` and `r'`, at line 23, the algorithm checks if the same branches are taken. If not, switching `p` actually alters the branch outcome of a predicate that `u` is control dependent upon, which implies that we are not able to find `u'`. Hence the algorithm returns failed. We will further illustrate this case with an example later on. If the same branches are taken, the algorithm recursively calls `MatchInsideRegion()` to further match subregions that are one level below until `u'` is found. Next let us show how to find the match of $\texttt{15}^{(1)}$ in (2) using the algorithm. Both (1) and (2) are first parsed into regions as depicted in Fig-

Algorithm 1 *Matching Algorithm*

Description: *Match()* finds the use in E' that corresponds to u in E given p in E corresponds to p' in E' and u is not in *Region*(p). *MatchInsideRegion()* finds the use in region R' that matches u in R.

```
1:  Match ($p$, $u$, $p'$)
2:  {
3:        $r$=Region($p$);
4:        $r'$=Region($p'$);
5:        while (!InRegion ($u$,$r$)) {
6:              $r$ = Region($r$);
7:              $r'$ = Region($r'$);
8:        }
9:        return MatchInsideRegion(r, u, r');
10: }
11:
12: MatchInsideRegion ($R$, $u$, $R'$)
13: {
14:       $r$=FirstSubRegion($R$);
15:       $r'$=FirstSubRegion($R'$);
16:       if ($r'$ == NULL ) return NULL;
17:       while (!InRegion ($u$,$r$)) {
18:             $r$ = SiblingRegion($r$);
19:             $r'$ = SiblingRegion($r'$);
20:             if ($r'$ == NULL) return NULL;
21:       }
22:       if (FirstStmt($r$) == $u$) return $r'$;
23:       if (Branch ($r$) != Branch ($r'$) ) return NULL;
24:       return MatchInsideRegion ($r$, $u$, $r'$);
25: }
```

ure 2. According to the algorithm, we first look for the region that covers both `2`$^{(1)}$ and `15`$^{(1)}$, which is the whole execution region, and its corresponding instance in (2). Then we match the subregions of the whole execution regions as shown by the dotted lines in the figure till we reach the subregion that contains `15`$^{(1)}$, which is `[13,14,15,16,17,18]`$^{(1)}$. The matching subregion is `[13,14,15,16,17,18]`$^{(2)}$. The same branches are taken at statement 13 in both the executions, and thus we move forward to match 15 in the subregions of `[14,15,16]`$^{(1/2)}$ and so on. Eventually, 15 is located in (2).

The above example demonstrates how the algorithm works when the corresponding use exists. Now let us take a look at another example that shows how it works for the case that the corresponding use does not exist. In this example, we try to match `15`$^{(1)}$ in (3). We are able to match in a similar manner till we reach the following two subregions: `[14,15,16]`$^{(1)}$ and `[14]`$^{(2)}$. By looking at the branch outcomes at statement `14`$^{(1/2)}$, we conclude that we are not able to find 15 in (3).

The algorithm has to take care of some special cases. In `MatchInsideRegion()`, the sub-regions always match if their parents are single-entry-single-exit. In the case that the parents have a single-entry-multiple-exit structure, different sequences of subregions may be exercised inside the parents which increases the complexity of execution alignment. For example, in Figure 3, `R` and `R'` match and we are looking for the match of 7. Since in the new execution, C_0 takes the true branch such that the control flow exits the loop by a break. Line 16 and 20 in Algorithm 1 handle such cases. If a different exit point is taken in the new run before a subregion contains `u` is reached, a sibling region cannot be found. In the example, we can see that the sibling of region `[4]` is *NULL* in the second run, which implies region `R'` terminates. We conclude that the match of 7 is not found.

```
1:  if ($P$) ...
2:  ...
3:  while ($i < t$) {
4:        if ($C_0$)
5:              break;
6:        if ($C_1$)
7:              ...=$x$;
8:        $i = i + 1$;
9:  }
10: ...;
```

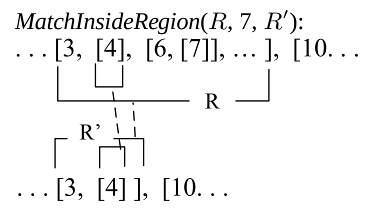


Figure 3. An example of the single-entry-multiple-exit case.

Finally, according to Definition 2, and the results of applying the alignment algorithm to execution (1) and (2), we conclude that $2^{(1)} \xrightarrow{id} 15^{(1)}$ because $2^{(2)} \xrightarrow{cd} 4^{(2)} \xrightarrow{dd} 15^{(2)}$. If statement 3 is changed to "t=C_2=1;", $2^{(1)} \xrightarrow{id} 15^{(1)}$ holds since predicate at 14 takes the false branch resulting in 15 does not get executed as shown in execution (3).

During the procedure of debugging, it is often the case that the programmer knows what the correct value v_{exp} when he observes the wrong value at o. This information can be used to *strongly* suggest the implicit dependence edge that leads to the execution omission error.

DEFINITION 4. *Given an execution E and an expected value v_{exp} at an execution point of o, let o' be the point in the switched execution E' that corresponds to o, VALUE(o') be the right hand side value at o', there is a* **strong implicit dependence** *between a predicate p and a use u, denoted as $p \xrightarrow{sid} u$, iff*
(i) $p \xrightarrow{id} u$;
(ii) v_{exp} = VALUE(o').

In other words, if the expected correct value can be produced at o', which corresponds to o in the original execution, the implicit dependence is a strong implicit dependence. We will see that a high priority is given to strong implicit dependences in producing the fault candidate set.

3.2 Demand Driven Strategy

According to the definition of implicit dependence, verifying one implicit dependence requires reexecuting the program once. If an error is not captured by the dynamic slice because of the missing implicit dependences, potentially, we need to verify the implicit dependences between each use in the slice and its preceding predicates. Therefore, the technique may become impractical without a sophisticated design.

We reduce the number of verifications through a demand driven strategy consisting of two iterative steps. First, we *prune* a dynamic slice as much as possible before we start verifying the related implicit dependences. Second, we pick one `u` from the pruned slice that is considered as the most promising one leading to the root cause and expand the slice by adding the executed statements that `u` implicitly depends on. These two steps are repeated until the root cause is found.

Confidence Analysis Based Pruning. In [19], a technique is proposed to compute for each executed statement the likelihood of it being faulty. The basic idea is derived from the observation that some of the statements used in computing an incorrect value may also have been involved in computing correct values (e.g., a value produced by a statement in the dynamic slice of the incorrect value may also have been used in computing a correct output value prior to the incorrect value). Hence, it is possible to estimate the

10.	`a=1;`	$C = f(range(A))$	?
	`...`		
20.	`b=a % 2;`	$C = 1$	$\surd$
	`...`		
30.	`c=a+2;`	$C = 0$	$\times$
	`...`		
40.	`printf("%d",b);`	$C = 1$	$\surd$
41.	`printf("%d",c);`	$C = 0$	$\times$

Figure 4. An example of confidence analysis

likelihood of a statement execution being faulty by looking at its relations to both the correct output and the wrong output. Such a likelihood is represented by a *confidence value* ranging from 0 to 1 – a higher confidence value corresponds to greater likelihood that the statement execution produced a correct value. This technique is also called *confidence analysis*.

For example in Figure 4, let us assume that the user observes that statement 40 outputs a correct value while 41 outputs a wrong one so that they are associated with the the confidence values of 1 and 0, respectively. The goal of confidence analysis is to automatically infer the confidence values of other statement executions. The definition at statement 30 cannot reach the correct output, which can be interpreted as there is no evidence that indicates it produces a correct value. Therefore, it has a confidence of 0. The definition at 20 reaches both the correct and the wrong outputs. From the fact that `b` at 41 is observed to be correct, we can infer that 20 produces a correct value and hence its confidence is 1. However, from 20 being correct, we cannot infer that 10 is correct because the computation at 20 represents a many-to-one mapping from `a` to `b`. For example, `a=3,5,7,...`, and so on produces the same correct value at 20. Therefore, the confidence of 10 is computed based upon the range of `a`, which can be approximated by the value profile.

Confidence values are used in both the iterative steps mentioned earlier. In the first step, the executed statements having a confidence value of 1 can be pruned from the dynamic slice since they are not fault candidates. In the second step, we rank the executed statements in the pruned slice based on their confidence values and their dependence distances to the failure point, and then pick the one with the highest rank to perform implicit dependence verification. After implicit dependences are verified, they are added to the dependence graph and then confidence values are recomputed to incorporate the new dependence edges. A new ordered fault candidate set is produced accordingly.

Demand Driven Procedure. Algorithm 2 presents the demand driven procedure. `PruneSlicing ()` was an implementation of the confidence analysis [19], which is able to compute a pruned and ranked slice. It is also an interactive procedure, in which the system presents the statement instances in the slice in an order and the programmer gives feedback to the system if he considers the presented statement instance contains benign program state. This procedure terminates after a few interactions with the programmer such that the remaining statement instances in the pruned slice have only corrupted program state. Note that a statement whose instances have corrupted program state is not necessarily the error. The purpose of calling `PruneSlicing ()` is to have the smallest slice before the expansion along implicit dependence edges during each step.

In each iterative step of the `while` loop, a use `u` is selected from the pruned slice based on its ranking. The algorithm makes use of the concept of potential dependence from relevant slicing. The set of statement executions on which `u` potentially depends on, denoted as `PD(u)`, are used as candidates on which verification is performed. Lines 7-8 group the predicates in `PD(u)` based on the verification results. The existence of strong implicit dependences

Algorithm 2 *Demand-Driven Algorithm*

Description: *LocateFault()* locates the root cause by adding implicit dependences on demand, given G, $O_{\surd}$, $o_{\times}$, v_{exp} as the dynamic dependence graph, the set of correct output, , the wrong output, and the expected correct value at $o_{\times}$, respectively. *Verify ()* tests the relation between p and u. It returns $STRONG\_ID$ / ID if $p \xrightarrow{sid/id} u$, otherwise NOT_ID.

```
1:  LocateFault ($G$, $O_{\surd}$, $o_{\times}$, $v_{exp}$)
2:  {
3:      $PS$=PruneSlicing ($G$, $O_{\surd}$, $o_{\times}$);
4:      while (the root cause is not found) {
5:          select a use $u$ from $PS$;
6:          $S[...] = \{\phi, \phi, \phi\}$;
7:          foreach $p$ in PD ($u$) {
8:              $S$[VerifyDep ($p$,$u$,$o_{\times}$, $v_{exp}$)] + = $p$;
9:          }
10:         if ($S[STRONG\_ID] \neq \phi$) $type = STRONG\_ID$;
11:         else $type = ID$;
12:         foreach $p$ in $S[type]$ {
13:             foreach $t$ s.t. $p \in PD(t)$ {
14:                 if (VerifyDep ($p$,$t$, $o_{\times}$, $v_{exp}$)==$type$) {
15:                     $G$=$G$+ $p \rightarrow t$;
16:                 }
17:             }
18:         }
19:         $S$=PruneSlicing ($G$, $O_{\surd}$, $o_{\times}$);
20:     }
21: }
22:
23: VerifyDep ($p$, $u$, $o_{\times}$, $v_{exp}$)
24: {
25:     $E'$ = reexecute $E$ with $p$'s branch outcome altered;
26:     $p'$ = the match of $p$ in $E'$;
27:     $o'$= Match ($p$, $o_{\times}$, $p'$);
28:     if ($o'$ $\neq$NULL && Value ($o'$) == $v_{exp}$) return STRONG_ID;
29:     $u'$= Match ($p$, $u$, $p'$);
30:     if ($u'$==NULL) return ID;
31:     else {
32:         $d'$=the definition of $u'$;
33:         if (InRegion($d'$,Region($p'$)))
34:             return ID;
35:     }
36:     return NOT_ID;
37: }
```

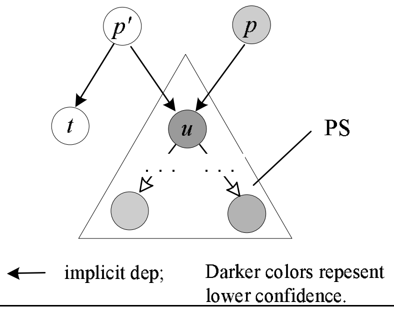


Figure 5. Enable pruning by verifying more related implicit dependences.

overrides the existence of implicit dependences, meaning that only strong implicit dependence edges will be added if there are any. Note that for each candidate p, the algorithm verifies not only the dependence between p and u, but also the dependences between p and any other uses that potentially depend upon p. The goal is to enable more pruning during computation of the new fault candidate set. For example in Figure 5, the triangle represents the the current pruned slice PS. Assume it does not contain the root cause, according to the algorithm, we need to verify the potential dependences that leads to u. Let us further assume $\texttt{p} \xrightarrow{id} \texttt{u}$ and $\texttt{p'} \xrightarrow{id} \texttt{u}$ are verified. If we do not verify the dependence $\texttt{p'} \xrightarrow{id} \texttt{t}$ and add it to the dependence graph, the high confidence of t can not be propagated to p' and thus p' has to be considered as a fault candidate.

The procedure of expanding and then pruning repeats until the root cause is captured in the slice. At this moment, the dependence chains in the pruned slice clearly disclose the cause effect relations between the root cause and the failure.

VerifyDep() presents an algorithm to test whether or not there is a (strong) implicit dependence between a predicate *p* and a use *u* based upon Definition 2 and 4. The algorithm is derived directly from the definitions except in lines 32-34 it considers the implicit dependence is true only if a data dependence *edge* exists between u' and d' instead of an explicit dependence *path* as described by the definition. Considering paths but not edges substantially increases the number of fault candidates added during each iterative step, which is not desirable because the programmer can be easily overwhelmed. Moreover, only minor degradation results from considering edges, meaning that the error will still be contained eventually.

For example, let us assume statement 7 in Figure 2 is x=.... According to the definition, $2^{(1)} \xrightarrow{id} 15^{(1)}$, which matches our intuition, because of the explicit dependence path $2 \xrightarrow{cd} 3 \xrightarrow{dd} 6 \xrightarrow{cd} 7 \xrightarrow{dd} 15$ in the switched run. According to the algorithm, there is no implicit dependence between $2^{(1)}$ and $15^{(1)}$ because switching P does not introduce an explicit data dependence. However, the algorithm is able to identity $2^{(1)} \xrightarrow{id} 6^{(1)}$ and $6^{(1)} \xrightarrow{id} 15^{(1)}$. In other words, $2^{(1)}$ will eventually be included in the fault candidate set through these two edges.

We would like to point out that considering edges instead of paths in VerifyDep() makes this procedure unsafe under certain situations, meaning some executed statements may not be reachable from *u* even though *u* implicitly depends on them. For instance, let us assume statement 4 is C_1*=1;* and statement 9 is *x=....* The algorithm decides there is no implicit dependence either between $2^{(1)}$ and $15^{(1)}$ or between $2^{(1)}$ and $6^{(1)}$, therefore, $2^{(1)}$ is not reachable from $15^{(1)}$. However, according to the definition, $2^{(1)} \xrightarrow{id} 15^{(1)}$. The observation is that if switching a predicate changes the branch outcomes of nested predicates, the algorithm is not safe. A safe algorithm can be derived with the cost of much more verifications being performed in each iteration of the demand driven process, which sacrifices the merit of the process. We argue and later show by experimentation that these situations are rare and the unsafe algorithm is efficient and effective enough for the application of debugging.

Next let us revisit our motivation example in Figure 1 to demonstrate this algorithm. Given the correct output and the wrong output observed at S_9 and S_{10}, respectively, the computation steps are presented as follows.

(1). Prune the dynamic slice of the wrong output from $\{S_2, S_3, S_6, S_{10}\}$ to $\{S_2, S_6, S_{10}\}$. S_3 is removed from the dynamic slice because it has a one to one mapping to the correct output;

(2). S_{10} is selected for expansion. $PD(S_{10}) = \{S_7\}$. Since $VerifyDep(S_7, S_{10})$ returns NOT_ID, no dependence edges are added;

(3). This time S_6 is selected for expansion. $PD(S_6) = \{S_4\}$. Since $VerifyDep(S_4, S_6)$ returns STRONG_ID, dependence $S_4 \xrightarrow{sid} S_6$ is added. According to the algorithm, the statement executions that potentially depends on S_4 are also tested in order to facilitate pruning. In our case, there are other statement executions potentially depend on S_4.

(4). After adding the implicit dependence, a new pruned slice is computed as $\{S_1, S_2, S_4, S_6, S_{10}\}$, which contains the root cause and clearly explains how the failure is induced.

A plausible alternative to our technique is to directly combine relevant slicing and confidence analysis. Unfortunately, this straightforward solution is problematic. In relevant slicing, the edges added to the dynamic dependence graph represent potential dependences. Propagating confidence along these possibly false dependence edges may result in a faulty statement appearing non-faulty. For example in Figure 1, relevant slicing adds a dependence between S_7 and S_9, which is false. Such a false dependence enables propagation of the confidence value 1 from S_9 to S_7 and then to S_1, which eventually sanitizes the root cause. This suggests that confidence analysis can only be performed along verified *implicit* dependence edges. It also provides additional motivation for our technique of detecting implicit dependence through predicate switching.

4. Experimental Evaluation

The prototype consists of three components. The *online component*, which was built on top of valgrind-2.2.0 [14], constructs a dynamic dependence graph with control flow and timestamp annotations for an execution. In the *static component*, diablo-0.3 [2] was adapted to build the control flow graph from a x86 binary and compute static control dependences. A union dependence graph, which is static, is also construted by this component by unioning all the unique dependences that were exercised during the execution of a large number of test cases. Such a graph is used to compute potential dependences. The *debugging component* implemented the confidence analysis, the demand driven debugging process, and the implicit dependence verification.

Given a x86 binary, the prototype first executes the binary with a large set of test cases to construct the static dependence graph and collect value profile for the confidence analysis. Next, the prototype executes the specific failing run to construct the dynamic dependence graph which contains only explicit dependences. The offline debugging component takes the dynamic dependence graph, the static dependence graph, the set of correct output, and the first wrong output to start the debugging process.

Benchmarks. Table 1 presents the benchmarks used in our experimentation. These programs are medium-sized linux utilities that belong to the Siemens suite [12]. LOC represents the lines of source code. Each program has multiple versions and each version has a number of real or seeded errors. Test cases are also provided to expose these errors. We did not use the benchmark make in the suite because we were not able to expose any errors using the provided test cases.

Execution Omission Errors. We first investigated all the errors in the suite and identified the execution omission errors. More precisely, we filtered out all the errors that were captured by traditional dynamic slicing techniques. The remaining errors are execution

Table 1. Characteristics of benchmarks

Benchmark	LOC	# of procedures	Error type	Description
flex	10459	162	seeded	a fast lexical analyzer generator
grep	10068	146	seeded	a unix utility to print lines matching a pattern
gzip	5680	104	seeded	a LZ77 based compressor
sed	14427	255	real & seeded	a stream editor for filtering and transforming text

Table 2. Execution Omission Errors

Benchmark	Error	RS (static/dynamic)	DS (static/dynamic)	PS (static/dynamic)	RS/DS (static/dynamic)	RS/PS (static/dynamic)
flex	$V_1 - F_9$	963/88K	946/83K	13/31	1.02/1.06	74/2838
	$V_2 - F_{14}$	849/157K	714/27K	9/476	1.18/5.8	94/329
	$V_3 - F_{10}$	600/103K	80/6.8K	8/294	7.5/15.1	75/350
	$V_4 - F_6$	894/265K	629/29K	2/4	1.42/9.14	447/66250
	$V_5 - F_6$	108/915	104/873	9/15	1.04/1.05	12/61
grep	$V_4 - F_2$	489/32K	416/3K	416/3K	1.18/10.7	1.18/10.7
gzip	$V_2 - F_3$	48/618	6/9	3/5	8/68.7	16/123
sed	$V_3 - F_2$	575/392K	498/118K	18/76	1.15/3.32	31.9/5158
	$V_3 - F_3$	222/5.0K	202/3.8K	202/3.8k	1.10/1.32	1.10/1.32

omission errors as presented in Table 2. Column `error` displays the set of errors that are under study. Error $V_x - F_y$ denotes the yth error in the x version of the specific program. `RS` represents the relevant slice. `Static` and `dynamic` give the number of unique source code statements and the number of dynamic statement instances in the slice. Note that a static statement can be executed many times during an execution, resulting in multiple instances of the statement. `DS` and `PS` denote the *traditional* dynamic slice and the *pruned* dynamic slice, respectively. The last two columns compare their sizes.

From Table 2, we are able to make the following observations:

- `RS` captures all the execution omission errors, but the sizes of `RS` are very large, which simply make manual inspection infeasible. `DS` misses all the errors, and `PS`, which is the pruned version of `DS`, misses all the errors as well.
- The `static` sizes of `RS` and `DS` are comparable. However, the `dynamic` sizes of `RS` are substantially larger than those of `DS`, which implies much more manual effort be required in the case that instance information is essential to understand the cause-effect relations.
- The sizes of `PS` are significantly smaller than those of `RS`, which makes inspecting `PS` much easier. This strongly suggests that execution omission errors should be located by starting with small pruned slices and then gradually exploring implicit dependence edges.

Effectiveness. Table 3 shows the evaluation results of effectiveness. The column labeled '`# of user prunings`' presents the number of times that we had to tell the system that a specific statement instance is benign before the system can acquire the minimal pruned slice, in which all statement instances had corrupted program state. Zero user prunings indicates that the automatic pruning based on $O_{\surd}$ and $o_{\times}$ is able to produce the minimal pruned slice. The columns labeled '`# of verifications`', '`# of iterations`', and '`# of expanded edges`' present the number of verifications performed in order to identify the (strong) implicit dependence edges, the number of iterations before the error was located, and the number of (strong) implicit dependence edges added, respectively. `IPS` denotes the final pruned expanded slice that contains the error. `OS` is the failure-inducing dependence chain from the error to the failure. In other words, it is the lower bound for a slice that can be produced by dynamic slicing-based technique. We manually identified these chains in order to perform the evaluation.

The observations from Table 3 are as follows.

- The numbers of user interactions that are required to achieve minimal pruned slices are small, which implies that pruning is very effective. In order to reduce the subjective factor of the experiment, we first manually identified the `OS`, which is the failure inducing chain, and then statement instances not in `OS` were selected from the pruned slice in order as being benign.
- The numbers of verifications are reasonable, showing that pruning and the demand driven process successfully control the number of edges that we need to verify.
- The numbers of iterations and the numbers of added (strong) implicit edges are mostly very small. In most cases, we only need to expand the pruned slice once. This implies that after we reduce the slice to its minimal form, the execution omission errors can be contained by adding very few implicit edges in one expansion. Note that adding one implicit edge to the dependence graph can make a number of executed instances become reachable. In `sed`-$V_3 - F_2$, we expanded twice by adding two strong implicit dependences edges. Our experiment reveals that *most execution omission errors only propagate along very few implicit dependence edges before they manifest themselves.* The results support our proposed method of first reducing the slice to its minimal form and then expanding along implicit dependence edges.
- The sizes of `IPS` are very close to those of `OS`, meaning that we were able to acquire nearly optimal slices.
- While in Table 2, the `dynamic` sizes are mostly orders of magnitude larger than the `static` sizes. The `static` and `dynamic` sizes of `OS` in Table 3 are comparable. It implies that *manually investigating dynamic statement instances is feasible.* Instances contain much more prolific information such as values and addresses which can greatly facilitate debugging than static statements do. Previously people were reluctant to inspect instances because they believed that errors may propagate through too many instances and manually inspecting them is unrealistic. Our experiment supports the opposite.

Table 3. Effectiveness

Benchmark	Error	# of user prunings	# of verifications	# of iterations	# of expanded edges	IPS (static/dynamic)	OS (static/dynamic)
flex	$V_1 - F_9$	2	5	1	5	17/51	7/16
	$V_2 - F_{14}$	1	4	1	1	7/24	7/24
	$V_3 - F_{10}$	1	1	1	1	4/2	4/2
	$V_4 - F_6$	0	6	1	5	8/28	6/23
	$V_5 - F_6$	1	2	1	2	10/27	10/27
grep	$V_4 - F_2$	15	313	1	62	103/2177	93/1196
gzip	$V_2 - F_3$	2	1	1	1	5/7	5/7
sed	$V_3 - F_2$	9	36	2	2	25/74	23/69
	$V_3 - F_3$	10	115	1	1	26/74	26/74

- `Grep`-$V_4 - F_2$ is the most complicated error we have. The error was propagated for a long time before it was observed. As a consequence, a large portion of the program state was polluted and the resulting `OS` was quite large. That was actually decided by the characteristics of `grep`, which does not display any intermediate program state before it terminates. `Flex` and `gzip` demonstrate the other extreme: results are emitted gradually during the execution, which makes debugging a lot easier.

Table 4. Performance

Benchmark	Error	Plain (sec.)	Graph (sec.)	Verif. (sec.)	Graph /Plain
flex	$V_1 - F_9$	0.29	22.7	2.7	78.3
	$V_2 - F_{14}$	0.28	22.3	1.92	79.6
	$V_3 - F_{10}$	0.28	22.4	0.52	80
	$V_4 - F_6$	0.34	15.6	3.6	45.9
	$V_5 - F_6$	0.12	2.2	0.48	18.3
grep	$V_4 - F_2$	0.43	66.6	43.3	154.9
gzip	$V_2 - F_3$	0.41	13.5	0.68	32.9
sed	$V_3 - F_2$	0.26	11.4	16.6	43.8
	$V_3 - F_3$	0.14	4.7	32.2	33.6

Performance. The last experiment is about performance. The runtime cost of this technique mainly stems from two procedures: the *online dependence graph construction procedure* which also collects control flow and timestamp information in order to compute potential and implicit dependences, referred to as `Graph` in Table 4; and the *verification procedure* that entails reexecuting the program and producing a partial predicate trace, referred to as `Verif.` in Table 4. `Plain` presents the execution times on the `valgrind` engine without any instrumentation. The original executions took a few miliseconds, which were so small that they may skew the results because starting up the `valgrind` engine and dynamically instrumenting the program takes much more time than the original execution. Therefore, a more reasonable comparison should be performed between `Plain` and `Graph`.

From Table 4, the online graph construction causes a slow down in execution by factors ranging from 18.3 to 154.9 due to the heavyweight instrumentation. Note that the dynamic instrumentation engine itself is slow to begin with. In the application of debugging, paying the high runtime cost once may be acceptable compared to the otherwise tedious manual efforts. The execution times presented in `Verif.` illustrate the cost of generating and aligning predicate traces. They are mainly decided by the number of verifications.

5. Discussion

Feasibility. One concern arises from the brute force predicate switching, which is about the feasibility of the switched path. Consider the code shown in Table 5(a). Let us assume that we are interested in finding the dynamic slice of `X` at `S3`. Further assume that the value of `A` was 15 and therefore `S1` is executed and `S2` is not executed before arriving at `S3`. In other words, the use of `X` at `S3` receives value of `X` defined at `S1`. By switching the outcome of predicate `P2`, we determine that a different value of `X` (the one defined at `S2`) reaches `S3`. As a result in our method it is assumed that an implicit dependence between `P2` and `S3` has been exposed. However, it seems that if `P1` evaluates to true, `P2` cannot evaluate to true. By forcing `P2` to evaluate to true we may introduce a spurious implicit dependence.

```
P1: if $A > 10$ then
S1:     X = ..
    endif
P2: if $A < 5$ then
S2:     X = ..
    endif
S3: .. = X
```

(a) Feasibility

```
S1: X = ..
S2: A = ..
P1: if $A > 10$ then
P2:     if $A > 100$ then
S3:         X = ..
        endif
    endif
S4: .. = X
```

(b) Soundness

Table 5.

Our argument is that we cannot completely exclude the possibility of `P1` or `P2` being the error. In other words, even though the path is infeasible in the faulty program, it may be feasible in the *correct* version of the program.

Soundness. We would like to point out that in general the proposed method is not sound. In particular, it may miss an implicit dependence. Now let us consider another example in Table 5(b), in which our method fails to expose an implicit dependence. Let us assume that the value of `A` computed at statement `S2` is 5, and as a result `P1` evaluates to false and `P2` is not executed. Therefore the value of `X` at `S4` comes from statement `S1`. When computing the dynamic slice of `X` at `S4` we try to expose implicit dependence by forcing the outcome of predicate `P1` to true. Forcing outcome of `P1` to true causes `P2` to execute but `P2` evaluates to false. As a result, `S3` is not executed, and thus no implicit dependence is found between `P1` and `S4`. If the value of `A` is incorrect, then we have actually failed to expose the implicit dependence between `P1` and `S4`. The cause of this problem is that the branch outcomes of nested predicates depend on the same definition. Switching one predicate at a time may not suffice.

While the example illustrates that we may fail to uncover an implicit dependence. We have not encountered such a case in our study. Furthermore, to fully overcome this problem, we have to either resort to a conservative solution such as relevant slicing or perturb the value of `A` instead of the branch outcome, which is much more expensive because `A` has an integer domain while a predicate has a binary domain.

6. Related Work

Dynamic slicing [8] is a technique that captures the executed statements that are involved in computation of a wrong value. Some previous work [20, 19] has shown that dynamic slicing is quite effective in locating many types of runtime errors. However, working by collecting data/control dependence information from executed statements, dynamic slicing is not capable of handling execution omission errors. Relevant slicing [3, 20] is a technique derived from dynamic slicing which conservatively adds dependence edges to the dynamic dependence graph if dependences could *potentially* happen between the omitted part and executed statements. As a result, spurious dependences are introduced and eventually the effectiveness of this technique is diminished. Our solution is based upon dynamic slicing as well. What distinguishes it from other work is that it verifies the existence of dependences that are *implicit* and edges are only added if the dependences are verified.

Predicate switching [18] is a dynamic analysis which proactively collects evidence about a software error. The basic idea is to switch the branch outcome of a predicate instance in the failed execution and then observe if the correct output can be produced *at the end of the execution*. If that happens, such a predicate is considered as *critical* to the error. In the proposed technique, we use predicate switching for a different purpose of disclosing implicit dependences. The switched execution does not need to run till the end and a small set of predicate are deliberately selected to switch in order to control the runtime overhead. In [16], Tao et al. proposed an path selection technique to expose software errors which is similar to predicate switching. They construct a successful program run which is closest to the failed run based on a distance metric. Evidence can be collected by comparing these two runs.

7. Conclusions

Execution omission errors are difficult to locate using traditional dynamic analysis because these analysis are typically designed to focus on what ever happened while execution omission errors are more related to what never happened. In this paper, we introduce the concept of *implicit dependences* which are dependences that are normally invisible due to the omission of execution of some statements. We design a novel dynamic method that enables detection of implicit dependences, which consists of reexecuting the program while switching a specific predicate instance, and aligning the original and switched executions. We also propose a demand driven process, which utilizes the confidence analysis to acquire the minimal pruned slice, and then identifies implicit dependences starting from the minimal slice, avoiding verifying a large number of potential dependences. Our results show that execution omission errors can be easily captured with the proposed techniques. Only a few implicit dependence edges need to be identified.

References

[1] http://www.cse.unl.edu/∼galileo/sir.

[2] http://www.elis.ugent.be/diablo/.

[3] Tibor Gyimothy, Arpad Beszedes, and Istan Forgacs. An efficient relevant slicing method for debugging. In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 303–321, Toulouse, France, 1999.

[4] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the International Conference on Software Engineering*, pages 291–301, Orlando, Florida, 2002.

[5] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 83–90, Montreal, Quebec, Canada, 1998.

[6] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, Boston, MA, USA, 2004.

[7] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the International Conference on Software Engineering*, pages 467–477, Orlando, Florida, 2002.

[8] Bogdan Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[9] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154, San Diego, California, USA, 2003.

[10] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 286–295, Lisbon, Portugal, 2005.

[11] Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *ASE '03: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, 2003.

[12] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transaction on Software Engineering*, 24(6):401–419, 1998.

[13] Joseph R. Ruthruff, Margaret Burnett, and Gregg Rothermel. An empirical study of fault localization for end-user programmers. In *ICSE '05: Proceedings of the International Conference on Software Engineering*, pages 352–361, St. Louis, MO, USA, 2005.

[14] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-gnu/linux. In *http://valgrind.kde.org/*.

[15] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *ICSE'04:Proceedings of the International Conference on Software Engineering*, pages 512–521, Edinburgh, United Kingdom, 2004.

[16] Tao Wang and Abhik Roychoudhury. Automated path generation for software fault localization. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 347–351, Long Beach, CA, USA, 2005. ACM Press.

[17] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, Charleston, South Carolina, USA, 2002.

[18] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceeding of the International Conference on Software Engineering*, pages 272–281, Shanghai, China, 2006.

[19] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 169–180, Chicago,IL, USA, 2006. ACM Press.

[20] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the International Symposium on Automated Analysis-driven Debugging*, pages 33–42, Monterey, California, USA, 2005.