Locating Software Faults Based on Minimum Debugging Frontier Set

Feng Li, Zhiyuan Li, Wei Huo, and Xiaobing Feng

Abstract—In this article, we propose a novel state-based fault-localization approach. Given an observed failure that is reproducible under the same program input, this new approach uses two main techniques to reduce the state exploration cost. Firstly, the execution trace to be analyzed for the observed failure is successively narrowed by making the set of trace points in each step a cut of the dynamic dependence graph. Such a cut divides the remaining trace into two parts and, based on the *sparse symbolic exploration* outcome, one part is removed from further exploration. This process continues until reaching where the fault is determined to be. Second, the cut in each step is chosen such that the union of the program states from the members of the cut is of the minimum size among all candidate cuts. The set of statement instances in the chosen cut is called a *minimum debugging frontier set* (MDFS). To evaluate our approach, we apply it to 16 real bugs from real world programs and compare our fault reports with those generated by state-of-the-art approaches. Results show that the MDFS approach obtains high quality fault reports for these test cases with considerably higher efficiency than previous approaches.

Index Terms—Fault localization, minimum debugging frontier set, sparse symbolic exploration, dynamic dependence graph

1 Introduction

Program debugging is an essential yet time-consuming task in software development and maintenance. In order to reduce the human effort spent on such a task, researchers have conducted extensive investigation in automatic techniques to locate faults in programs that cause execution errors. To this end, a methodology based on state alteration [1], [2], [3], [4], [5], [6] has gained much attention. Under this methodology, given a failed program run, intermediate program states (i.e., the values of certain variables and branch conditions) at chosen trace points are altered and the impact on the execution result is observed. Based on the observation, attempts are made to identify a set of program states that are potentially responsible for the originally observed execution error. It is well-known that a faulty program state may be due to either an incorrect branch decision or an incorrect variable value, and sometimes both. By definition, examining both branch conditions (a.k.a. branch predicates) and variable values will result in more accurate bug reports than examining the former alone. Hence, in this paper, we include both pieces of information in the examined program state.

In the absence of information on the correct program state, it is impractical to expect the debugging tool to recognize a faulty program state with absolute certainty, except at the failure point. In practice, little knowledge is known to the debugging tool about the expected program state. Therefore, fault localization techniques make best efforts to produce a bug report on suspected faulty states that are as close to the fault locations as possible under a reasonable cost, which is also the objective of this paper. We propose a new state alteration and exploration approach that has significant advantages, in terms of both accuracy and cost, over existing methods that are based on state alteration.

The central idea of our new approach is to iteratively select sets of trace points for efficient automatic state alteration and exploration. We first determine a set of dynamic data, control and summary control dependences (c.f. Section 3.1) which contributes to the unexpected state at the failure site. We then construct a dynamic dependence graph with statement instances involved in such depedences as nodes and those dependences as edges. We call such a graph the *initial analysis domain* and we successively subdivide it by making the set of trace points in each iterative step a cut of the corresponding graph. Such a cut divides the graph in two parts. The cut in each step is chosen such that the union of the program states from the members of the cut is of the minimum size among all candidate cuts. The set of statement instances in the chosen cut is called a *minimum debugging frontier set* (abbr. MDFS).

The concept of MDFS was introduced in our previous work [12] that led to a prototyping tool and a set of experimental results. In this paper, we present a theoretical foundation for MDFS and add important details of the main algorithm and its supporting analyses, especially those for extracting important control dependencies that are obscured in the program execution traces. We also present a new way to symbolically explore the program states for MDFS and extend the capability of MDFS to the handling of multiple faults, in addition to an improvement in finding new correctness properties when updating the analysis

- *F. Li and X. Feng are with State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100049, P.R.China. E-mail: {lifeng2005, fxb}@ict.ac.cn.*
- *Z. Li is with the Department of Computer Science, Purdue University, West Lafayette, IN 47907. E-mail: zhiyuanli@purdue.edu.*
- *W. Huo is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100049, P.R.China. E-mail: huowei@iie.ac.cn.*

Manuscript received 29 Sept. 2015; revised 14 Oct. 2016; accepted 19 Nov. 2016. Date of publication 0 . 2016; date of current version 0 . 2016.
Recommended for acceptance by M. Dwyer
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2632122

domain. New experiments are performed on a new set of test cases that are more up to date.

We organize the rest of the paper as follows. Section 2 provides technical backgrounds. Section 3 introduces the concept of the minimum debugging frontier set (MDFS). Section 4 presents the core MDFS algorithm and uses an example program that involves multiple faults to illustrate the algorithm. After describing how we implement our MDFS-based scheme in a debugging tool (Section 5), we present experimental results (Section 6). We apply our tool to 16 real bugs from real world programs and compare our fault reports with those generated by two state-of-the-art fault-localization methods that are based on state alteration and a mutation-based method. We also demonstrate the benefit of selecting points of state alteration based on MDFS over two alternative ways. We compare our scheme to related work found in literature (Section 7) and make concluding remarks in Section 8.

2 Background

A program *state* at any execution point, *p*, consists of both the path conditions leading to *p* and the values of variables at point *p*. For the purpose of debugging, one can perform a simple transform such that every branch condition is represented by the truth value of a variable. Hence, for simplicity of discussion, we will consider the program state to be the values of variables.

In this paper, a program failure at *p* is the violation of a program property $P(x_1, x_2, \ldots, x_n)$ that must hold at *p*. Such a property is specified by a predicate that consists of logical connectives (conjunction, disjunction, and negation) over arithmetic (in)equalifies. (For convenience, in the rest of the paper, we consider an equality to be a special case of inequalities.) For example, a divide-by-zero exception caused by integer division x/y is the violation of the property $y \neq 0$. For $P(x_1, x_2, \ldots, x_n)$ to be *false* at *p*, one or more of the variables x_i must hold incorrect values. There are several possibilities for x_i to hold an incorrect value, as listed below.

Definition 2.1 (Incorrect assignments). *Without loss of generality, we assume a variable x obtains its value from an assignment statement* $x := F(y_1, y_2, \ldots, y_m)$*, where* $\{y_i\}$ *is a set of variables and F is an operator over* $\{y_i\}$*. One of the two possibilities for x to hold an incorrect value is for F to be incorrectly composed by the programmer. This includes the special case in which an operand is missing from the* $\{y_i\}$ *set by mistake. Such an assignment is clearly a fault. The second possibility is for the correctly composed F to produce an incorrect result, due to the value of one or more operands* y_i *being incorrect. This assignment by itself is not a fault, but it may propagate a fault towards the failure site.*

A special case of incorrect assignments is an incorrectly written branch condition, e.g., a *while* loop condition $i<100$ being written as $i \leq 100$ by mistake. We call this special case an incorrect branch composition. Taking a wrong *if* branch is also viewed as having an incorrect branch composition, i.e., the branch condition should have been negated.

At run time, any use of a variable at any execution point *p* must have a unique reaching definition. An operand x may obtain an incorrect value either because its reaching definition is an incorrect assignment or because its reaching definition is an unintended one.

Definition 2.2 (Causes for unintended reaching definitions). *A variable use* u *may have an unintended reaching definition* d *for reasons listed below.*

Case 1: The definition *d* may reach *u* at execution point *p* by mistake, due to the erroneous omission of the intended definition (i.e., the intended assignment statement) in the program. We call this a fault of missing assignments, and we name the statement corresponding to *u* as the location of such a fault in this paper. A special case of missing assignments is a missing condition, e.g., a missing check for a *null* pointer. For this special case, we name the first statement that should have been guarded by the missing condition as the location of the fault in this paper.

Case 2: Suppose the intended definition, *d′*, for *u* does exist in the program, *u* may still have an unintended reaching definition *d*. There are three possibilities: (i) The assignment statement for *d* is not supposed to be in the program, which we consider to be a fault of incorrect assignment in Definition 2.1; (ii) The assignment statements for *d* and *d′* are misordered in the program by mistake. We view *d* as an incorrect assignment and name it to be the location of the fault; (iii) The assignment statements for *d* and *d′* are both correctly written but an incorrect branch taken previously causes *d′* to be skipped or *d* to be executed after *d′*.

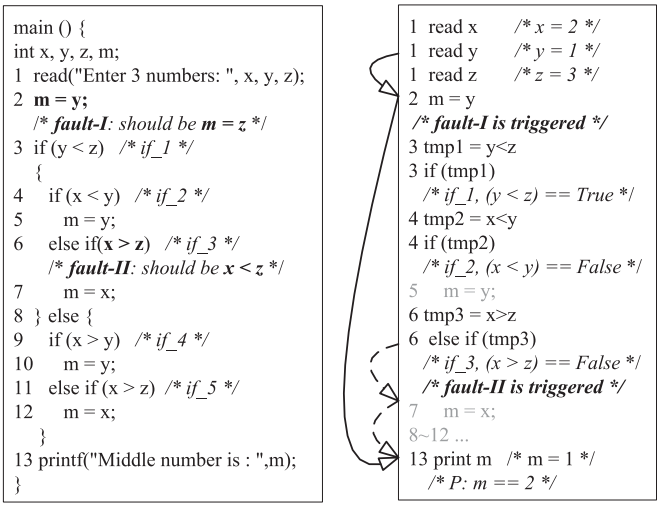
In Case (iii) above, the incorrect branch decision may be due to the branch statement being written incorrectly, which is clearly a fault. A correctly written branch condition may also result in an incorrect branch decision at run time, due to some operand used in the evaluation of the branch condition holding an incorrect value. The branch condition by itself is not a fault. Instead, we view it as a step to propagate the fault towards the failure site.

Definition 2.3 (Direct root causes for a failure). *If the violation of correctness property* $P(x_1, x_2, \ldots, x_n)$ *at execution point* p *is due, at least in part, to an incorrect assignment* $x_i := F(y_1, y_2, \ldots, y_m)$ *that reaches the use of* x_i *in P, we say that assignment is a root cause for the failure, i.e., the violation of P, at* p*. Similarly, if the violation of P is due to an unintended definition reaching the use of* x_i *in P as the direct result of a missing assignment or misordered assignments, we say such a fault is a root cause for the failure. Finally, if the violation of P is due to an unintended definition reaching the use of* x_i *in P as the direct result of an incorrect branch composition, we say the incorrectly written branch condition is a root cause for the failure. Since all root causes listed above directly impact P without any propagation, we call them direct root causes for the failure.*

Definition 2.4 (Transitive causes for a failure). *Suppose the violation of P is due, at least in part, to variable* x_i *in P having an incorrect value, but no direct root causes (according to Definition 2.3) are responsible for* x_i *being incorrect. We have one of the following transitive causes for the failure:*

Case 1: A correctly written assignment statement $x_i := F(y_1, y_2, \ldots, y_m)$ makes the correct reaching definition for the use of x_i in P but at least one of its operands, say y_j, has an incorrect value.

Case 2: An unintended definition reaches the use of x_i in P due to a *correctly* written branch condition evaluating to a wrong Boolean value, as the result of having at least one of



(a) a program P with multiple faults

(b) A failed execution trace of P with Input I = {x←2, y←1, z←3}

Fig. 1. An example with multiple faults.

its operands holding an incorrect value. This causes a wrong branch decision and hence a wrong reaching definition as described in Definition 2.2.

Either case listed above is called a transitive cause for the failure.

If we take the incorrect value of x or the incorrect branch condition in Definition 2.4 as the violation of a certain property, P, mentioned in Definition 2.1 or Definition 2.2, then we can apply Definition 2.3 to find the direct root causes or Definition 2.4 for the transitive causes for such violation. A root cause found in such a way is called a propagated root cause for the violation of P, i.e., the original failure. Applying Definitions 2.3 and 2.4 recursively, we will eventually reach all root causes (including the direct and the propagated ones) for P.

The example program in Fig. 1(a) take three input values (x, y, and z) and its intended output in line 13 should be the middle value. However, an incorrect value is printed when the program takes ($x = 2, y = 1, z = 3$) as its input. By our definitions given above, the cause for this execution failure can be traced back to two faults in the program. One is the incorrect value assignment in line 2 and the other is the incorrect branch composition in line 6. If the branch condition in line 6 were written correctly, then the incorrect value assignment in line 2 may not have reached the print statement in line 13. However, under the given input, both faults are triggered, and the missing events (illustrated by the dotted lines) cause an unintended reaching definition (Definition 2.2), illustrated by the solid arrow in Fig. 1(b)). According to Definition 2.3, the incorrect assignment in line 2 and the wrong branch composition in line 6 are the direct root causes of the failure.

Where no confusion results, in this paper, the term "*root cause*" may refer to either a statement in the program or an execution instance of that statement during the failed run. To make the latter explicit, we use the term *root cause in the failed run*. The program state associated with a root cause is called an initial faulty state of the failed run. The program state associated with a transitive cause is called an intermediate faulty state. The example given above has two root causes for the failure and such a failure is called a multi-fault failure.

The technique presented in this paper is aimed at reporting suspected root causes to the programmer in the event of an execution failure. Additional information based on dynamic dependence chains, as discussed in Section 3, is provided to the programmer for searching around the suspected root causes that do not turn out to be the actual ones.

3 Minimum Debugging Frontier Sets

In this section, we formally define the *minimum debugging frontier sets* (MDFS) that underlies our fault-localization approach. The MDFS is extracted from a dynamic dependence graph, which is also formally defined below.

3.1 The Dynamic Dependence Graph

Although the concept of an execution trace is useful for introducing the definition of root causes and transitive causes in this paper, our technique does not explicitly generate an execution trace for diagnosis. Instead, after a failure is observed, we execute an instrumented version of the program that reproduces the failure and, at the same time, constructs a dynamic dependence graph defined over the executed statement instances. (In this paper, we handle only failures that are reproducible.) Each statement instance has a sequence number in the order of execution. The *dynamic dependence graph* (abbr. DDG) for the failed run contains nodes that represent statement instances from the start of the execution to the failure site p. A dependence edge (s_i, s_j) indicates a dependence of statement instance s_j on s_i. Such an edge may be a flow dependence, a direct control dependence, or a *summary control dependence*, all to be introduced below. Traditional DDGs do not include summary control dependences. However, for failures due to incorrectly skipping certain statement instances during execution, the root causes may not appear in the DDG unless summary control dependences are included. Experimental evidence will be presented later (in Section 6.3). An array $Map[s_i]$ maps statement instance s_i to the program statement s in the (static) program dependence graph (PDG) [7].

The search for the root causes is conducted by following chains of dependencies between statement instances that take place during the execution, eventually leading to the violation of property P. By Definitions 2.1-2.4, each link in such chains of dependencies corresponds to a transitive cause. Establishing flow dependencies at run time is done by recording, for each variable, the most recent instance of the assignment statement s that writes to the variable. Until the same variable is modified again, any statement instance that reads this variable has a flow dependence on this instance of s.

We also establish two kinds of control dependencies between statement instances at run time, namely *direct control dependencies* (which are also known as dynamic control dependencies [8]) and *summary control dependencies*. A statement instance s in the execution trace has a direct control dependence on b if and only if b is the most recent branch statement instance such that a different branch decision would have caused s *not* to be executed. (Where no confusion results, we may simply say that s has a control

dependence on b, dropping the "direct" attribute.) The following algorithm describes the actions taken when executing the next statement instance s in order to detect direct control dependencies.

Algorithm 1. (Run-time detection of direct control dependencies)

Suppose the statement instance executed next is s.

If s is an instance of a branch statement B, insert s to a stack, Q, of live branch statement instances. Otherwise, let s be an instance of statement S and suppose the top of Q is an instance, b, of a branch statement B'.

If S is control dependent on B' in the PDG, then draw a control dependence edge from b to s. If not, then b is no longer a live branch statement instance. (In other words, the execution has exited the branch.) We remove b from the stack and continue searching and updating Q until either the correct branch statement instance is found or Q becomes empty. In the latter case, s is not control dependent on any statement instance.

A summary control dependence captures the effect of the branch decision on reaching definitions even if such an effect cannot be represented *at run time* as a composition of direct control dependencies and flow dependencies. To make a formal definition for summary control dependencies at run time, we first define summary control dependencies in terms of the PDG.

Definition 3.1. *A statement* S *is said to have a* summary control dependence *on a branch statement* B *if and only if 1)* S *is not control dependent on* B*; 2) there exists a path from* B *to* S *that contains an assignment statement* T *whose value definition reaches a use of the same value by* S*; and 3) there exist a branch decision for* B *that can cause* T *not to be executed.*

Note that, in the definition above, T does not need to be directly control dependent on B. A chain of control dependencies connecting B to T suffices. The reason we explicitly establishes a dependence between B and S statically is because, at run time, certain statement instances, an instance of T for example, may not be executed due to the branch decision, making it infeasible to infer the dependence of the instance of S on the instance of B. Further note that we may not always be able to exactly verify condition 2) at compile time, due to potential complications such as pointer aliasing. The default is to conservatively assume that condition 2) is met. From the summary control dependencies computed based on the PDG, we can establish run-time summary control dependencies between statement instances, which subsume potential control dependencies proposed in prior work [9]. Note that, under the given program input, one may not always be able to verify condition 2) in Definition 3.1 between a pair of statement instances at run time either, because the said path from B to S may not be actually taken, hence giving no opportunity to check the potential reaching definition.

In order to establish summary control dependencies based on PDG, we first transform the program into a static single assignment (SSA) form [10] such that each use has a unique reaching definition. If a use has multiple reaching definitions in the original form, the SSA form will have a new definition created, denoted by a *phi* node, that makes a single selection among those reaching definitions. Such *phi* nodes are placed at the dominance frontiers of those original reaching definitions, which are essentially the merging points of paths originating from certain branches. The following algorithm computes summary control dependencies.

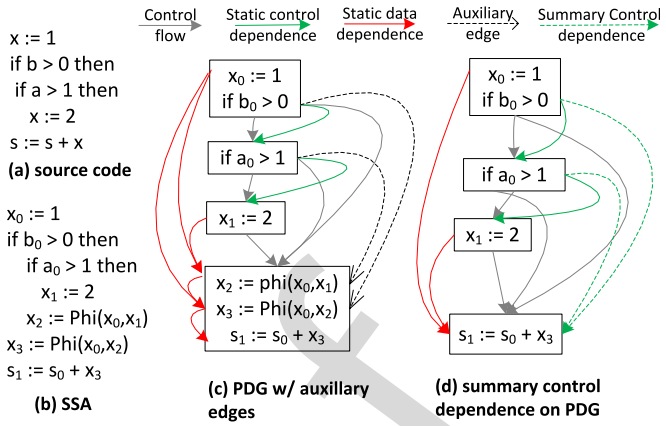


Fig. 2. An example of establishing summary control depedence.

Algorithm 2. (Computing Summary Control Dependencies)

STEP 1 (*Establishing relationship between branch statements and phi nodes*) Suppose a *phi* node is created to select among a number of reaching definitions. Let $\hat{C}$ be a set of branch statements such that each member in $\hat{C}$ can reach at least one of the reaching definitions through a path consisting of only (direct) control dependence edges. We add an auxiliary edge from each member B to the *phi* node in PDG.

STEP 2 (*Finding summary control dependencies*) Suppose a flow-dependence path exists in PDG from a *phi* node to a statement S and there is an auxiliary edge from a branch statement B to the *phi* node. We add a (static) summary control dependence edge from B to S in PDG.

Fig. 2(c) and 2(d) show how we extract summary control dependences based on the SSA form (Fig. 2(b)) of the given program (Fig. 2(a)). Each dotted arrow in Fig. 2(c) represents an auxiliary edge added from a branch statement to its corresponding *phi* node. The dotted arrows in Fig. 2(d) represent summary control dependences. For brevity, statements from the same basic block, i.e., statements shown in the same rectangle node in Fig. 2, share the same control dependence edges. The algorithm can be extended to compute inter-procedural summary control dependences in a system dependence graph (SDG) [7]. An SDG is a collection of PDGs, one for each procedure. Interprocedural summary control dependences are calculated in the same way except that flow dependences involving each call site are calculated beforehand based on the def/use summary collected for each procedure.

Due to the fact that a summary control dependence involves a flow dependence in the PDG, any inaccuracy of flow dependence information, due to the limitation of static program analysis, can cause inaccuracy of summary control dependence information. Therefore, the summary control dependencies may be computed overly conservative, which can potentially increase the cost of fault localization performed on the DDG.

The following algorithm describes the actions taken when executing the next statement instance s in order to

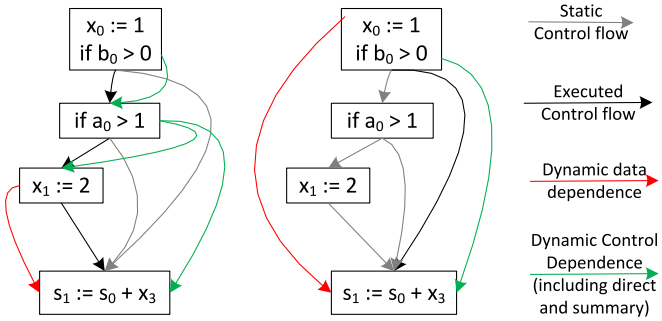


Fig. 3. An example of dynamic summary control depedence.

detect summary control dependencies. The summary control dependencies of each statement instance are detected after analyzing its data and direct control dependences. Note that at the time s is executed, the instance of the branch statement on which s has a summary control dependence may not be live any more. Hence, we cannot search the stack Q for such instances of branch statements. Instead, we maintain the sequence number of each executed instance of branch statement B, in the reversed order, in a list q^B. The actions taken by Algorithm 3 on s are performed after the actions taken by Algorithm 3.

Algorithm 3. (Run time detection of summary control dependencies)

Suppose s is the statement instance (of S) to be executed next. For each branch statement B' in the PDG such that S has a summary control dependence on B', we draw a summary control dependence edge from each instance b' in $q^{B'}$ to s. If s is an instance of a branch statement B, append the sequence number of s to the head of q^B.

Note that, at run time, the branch condition may be such that the assignment statement T in condition 2) of Definition 3.1 does get executed before s (i.e., s is flow dependent on s', an instance of T, and s' is directly control dependent on b, an instance of B'). In this case, the path in condition 2) exists at run time. Hence, there exists a chain of dependences from members of $q^{B'}$ to s that have already been established. We remove the summary control dependence edge from $q^{B'}$ to s, since any root cause reachable, in the reversed order, by following this edge is also reachable by following the existing dependence chain. Also note that a summary control dependence in the PDG can potentially result in a large number of summary control dependencies in the DDG, because we do not know a priori which branch instance b' may be incorrect. The presence of many summary control dependencies in the DDG may increase the cost of fault localization. In our experiments (c.f. Section 6), we compute intra-procedural summary control dependencies only, in order to save time and space.

Fig. 3 shows different dynamic dependence edges added for the code snippet shown in Fig. 2(a) when different control flows are executed.

Given any node, n, in DDG, all nodes and edges that are backward reachable from n constitute a subgraph of the DDG induced by n. Since such a subgraph is always acyclic, we denote the set of its roots by V_{begin} and the set of its sinks by V_{end}, respectively. The subgraph is then denoted by $DDG < V_{begin}, V_{end} >$. Once V_{begin} and V_{end} are given, all nodes V and all edges E in the subgraph are determined. The DDG has a unique sink that is p, where failure F is observed. We introduce a unique *ENTRY* node to be the unique root of DDG.

Claim 3.1. DDG contains all root causes for the observed failure, F.

Proof. Dynamic control dependencies computed by Algorithms 1-3 involve all branch statement instances that may have had an effect on the truth value of the property P. Dynamic flow dependencies in DDG involve all assignment statement instances that may have had an effect on the truth value of P. □

3.2 MDFS

Let v be a node in $DDG < V_{begin}, V_{end} >$, we call a subset of the nodes, C, in this subgraph a cut for v if, by removing C, v becomes unreachable from the nodes in V_{begin}.

Definition 3.2 (Minimal Debugging Frontier Set). *Given a node* v *in* $DDG < V_{begin}, V_{end} >$, *a minimum debugging frontier set* mdfs *for* v *is a cut that, among all possible cuts for* v, *has the minimum number of nodes.*

Note that, throughout this paper, the capitalized abbreviation MDFS refers to the concept and the method, but the lower case abbreviation *mdfs* denotes such a set used in the algorithm. Even though in practice the correct program states are usually unknown at most of the execution points, it is useful to analyze how the root causes could be located, suppose the correct program states were available. We have the following two claims that apply to both single fault and multi-fault failures.

Claim 3.2. If the program states are correct both at V_{begin} and at all statement instances in an *mdfs* for v, then a root cause for the failure observed at v must exist in subgraph $DDG < mdfs, v >$.

Proof. Suppose the claim were false, which would mean that all root causes must be in $DDG < V_{begin}, mdfs >$. Consider any one of these root causes. By definition, it must be located at the head of a dependence chain that leads to v such that the program state at every link in this chain is faulty. This chain must be a part of a chain from a node in V_{begin} to v. Hence, by Definition 3.2, it must contain a member in *mdfs*. This contradicts the assumption that the program states at all nodes in *mdfs* are correct. □

The claim above gives a sufficient condition for an initial faulty state to exist between the current *mdfs* and the failure point.

Claim 3.3. If any of the statement instances in an *mdfs* for v produces an incorrect value, then a root cause for the failure observed at v must exist in subgraph $DDG < V_{begin}, mdfs >$.

Proof. Suppose no root cause of the failure observed at v exists in any dependence chain from V_{begin} to any node in the *mdfs* for v. This means that no faulty program state exists at any statement instance in any dependence chain

leading to the *mdfs*. All values produced by members of *mdfs* then must be correct, which is a contradiction. □

The claim above gives a sufficient condition for an initial faulty state to exist prior to the nodes in *mdfs* in the trace.

We must emphasize again that, at most statement instances, no correctness criterion is actually available for verification. The significance of the claims above is to guide the development of heuristics for finding suspects of initial faulty states.

Algorithm 4. The Core Algorithm

Input: failure site p and a correctness property P at p.
Output: A fault report consisting of a root cause and a set of transitive causes.
// *Preprocessing*
Rerun the faulty program with the same input to construct the DDG;
Extract the initial analysis domain $D = DDG(V_{begin}, p)$;
repeat
  // *Step 1: Extract mdfs*
  Generate an *mdfs* for D;
  // *Step 2: Identify Polluted Entities*
  Repeat
    Symbolically alter a selected set of values produced by the statement instances in *mdfs*;
    Rerun the program from scratch, using sparse symbolic exploration to determine whether *the altered instances* are polluted entities; *(See* Section 4.2 *for details)*
  until all combination of the statement instances in *mdfs* are checked;
  //*Step 3: Re-extract an analysis domain*
  Update D according to CLAIM 3.2 and CLAIM 3.3;
  Simplify D based on the polluted entities, if any, found in this iteration, excluding those entities directly dependent on program inputs;
  Attach a new *correctness property* to the end point of D if available;
until no more *mdfs* is found in D;
Generate a fault report based on the set of polluted entities;

In Section 4, we present a scheme that iteratively extracts an *mdfs* and assesses which of the two conditions listed above is satisfied. In either case, the analysis domain will be narrowed, eventually converging to a root cause candidate.

4 An MDFS-Based Scheme

In this section, we present a scheme for fault-localization based on the MDFS concept. We first present a core algorithm that locates a single root cause responsible for a single observed failure. We then discuss the handling of the cases in which multiple root causes simultaneously trigger a single failure. In the event of multiple failures being observed, we can use the algorithm to locate the root causes of one failure first. After the programmer makes the correction, the algorithm can be applied again, suppose the other failures persist under the same input, to the next failure repeatedly, until no failures are observed.

4.1 The Core Algorithm

Our core algorithm (Algorithm 4) begins by constructing the initial analysis domain that consists of the DDG for a failed run. Next, it takes three major steps: 1) extracting a set of statement instances, on the basis of MDFS, to examine the program states; 2) identifying *polluted entities, namely* those statement instances that can be altered to make the failure disappear; and 3) narrowing the analysis domain with the conclusion of state exploration. These three steps are iterated until the analysis domain can no longer be narrowed further. Details of the core algorithm are provided below.

Preprocessing. Without loss of generality, we assume that the failure manifests itself either by generating an output that deviates from the expected output or by violating an assertion. In either case, we have a faulty program state at the failure site, p, and a set of variables that, collectively, are responsible. Viewing these variables as being used by p, we extract the subgraph DDG $< V_{begin}, p >$ from the DDG constructed during the execution, or the re-run, until the occurrence of the failure. This subgraph is the initial analysis domain for our scheme. For clarity, the following steps are first discussed for the initial analysis domain. After this, we will discuss what changes may be made in the later iterations.

Step 1: Extracting mdfs. In this step, we determine a minimum debugging frontier set *mdfs* for DDG $< V_{begin}, V_{end} >$, where $V_{end} = p$ in the first iteration. The problem of the MDFS calculation on a DDG subgraph can be transformed into the minimum cut problem [11] on the dual of the DDG subgraph. The minimum cut set of the dual is then converted back to a set of nodes in the original DDG. The statement instances represented by these nodes form the *mdfs*. If the *mdfs* is not unique, the one with the shortest average dependence chain from each of its elements to V_{end} is chosen.

Step 2: Identifying polluted entities. To identify polluted entities, we apply state alteration and exploration on the statement instances in the *mdfs* to see if changing any of the values generated by them can make the failure disappear, i.e., whether a correctness property, P, specified by the programmer at p, can be satisfied by the alteration. Since a fault may be propagated through multiple chains in the DDG, reaching multiple statement instances in the *mdfs*, we subject combinations of members in the *mdfs* to simultaneous alterations after altering the single members separately, unless each single member has already been identified as a polluted entity. When to stop exploring new combinations will be discussed later below.

Generally, altering an intermediate program state symbolically will in general cause multiple possible paths that deviate from the original failing trace, with more statement instances having multiple reaching definitions and, hence, producing more symbolic values (subject to branch-condition predicates) as a result. In our previous work [12], while we tried to execute as many statement instances using concrete values as possible, we used the bounded model checker CBMC [13] to perform symbolic execution wherever concrete values are unavailable. We used the MiniSAT2 tool [14] to solve the constraints generated by CBMC.

Under CBMC, however, the MDFS-based scheme must explore all possible paths (subject to a limit set by the user on the degree of loop unrolling) in the current analysis domain in order to generate the set of constraints to be solved by MiniSAT2. This can be highly time-consuming. To reduce the number of explored paths, one can take advantage of the fact that the direction in which the analysis

domain is narrowed depends solely on whether polluted entities exist in the current *mdfs*. When we apply an alteration trial to a selected combination (or a singleton, which is viewed as a special combination), as soon as the combination is found to be polluted by conducting symbolic execution through a single path, no other paths need to be explored for this combination any more. This observation leads to our new exploration strategy that explores one path at a time. (The cost comparison between our old and new strategies will be given in Section 6.1, and the method to find distinct paths is discussed in Section 4.2.)

Under the new strategy, in each alteration trial, let s denote the first statement instance (in the order of the original trace) among a specific combination to be altered, we rerun the program from scratch using concrete values until reaching s. From this point on, we explore various execution paths that can lead to p. On each path, we execute the program in a dual mode, i.e., using a set of concrete values to execute the statements in the path such that the intended branch selections can be followed. On the other hand, unlike in our previous work [12], we do not explicitly set up preconditions as predicates that alter the values assigned to any of the statement instances in the given combination. Instead, before starting symbolic execution on the currently chosen path, we introduce a new symbol for each assignment statement instance, if any, in that combination, to represent the assigned value. If a pointer or any field of a structure whose value is an address is marked symbolic, then all statements which may load value from that address must also be marked symbolic. The ensuing symbolic execution propatates such new symbols, statement by statement, towards p, such that a set of path constraints, C, is collected from the encountered branches and, at p, the correctness property P is also transformed into a set of predicates. Both C and P are now predicates over the new symbols.

If $P \wedge C$ is satisfiable, then we face two possibilities. It could mean that the program state of the combination at the *mdfs* was correct before altering, hence the root cause of the failure exists between the *mdfs* and the failure, but the altering (before the root cause was triggered) makes the failure disappear. The second possibility is that the altering indeed has fixed faulty program states at the *mdfs*, implying the root cause of the failure to exist prior to the *mdfs*. From our empirical experience (c.f. Section 6.2), the first scenario, i.e., "two wrongs make it right", is much less likely than the second possibility for programs in practice. Hence, unless additional information (c.f. Section 6.2) is available, our algorithm assumes the latter scenario and marks this combination as a *polluted combination*. In contrast, if during the exploration trial, the execution never reaches p or the set of path constraints generated at p is never compatible with P, then the combination is assumed as correct.

Since we do not need to always explore all paths, we call this mode of state exploration *sparse symbolic exploration*. As will be clear later, in the interest of finding new correctness properties that can be used for the new analysis domain, we want to find as many polluted combinations as we can. The details of formation of the new properties are given below. After finding all polluted combinations, members of *mdfs* that belong to any polluted combinations are marked as *polluted entities*.

Step 3: Re-extracting analysis domain. If no polluted entity is found in step 2, then by Claim 3.2, the root cause is assumed to exist after the *mdfs*. The analysis domain is narrowed to $D_B = DDG < mdfs, V_{end} >$, i.e., the bottom part, for the next iteration. We retain the property P for compatibility check at p for the next iteration, in which D_B becomes the new DDG $< V_{begin}, V_{end} >$.

In contrast, if any polluted entities are found in the current *mdfs*, then according to Claim 3.3, the analysis domain is narrowed by removing the DDG subgraph between the *mdfs* and p. The remaining subgraph $D_T = DDG < V_{begin}, mdfs >$, is further simplified by including only those nodes and edges that are backward reachable from the polluted entities, excluding those that are directly dependent on the program inputs (under the assumption that the inputs are always correct and hence are not a part of any analysis domain). The resultant subgraph becomes the new analysis domain, still denoted by $DDG < V_{begin}, V_{end} >$, for the next iteration. We then check to see whether we can move compatibility check further towards the program entry in the interest of more efficient symbolic exploration in the next iteration. If all polluted entities are due to branch decisions, then we can replace P at p by a new correctness property, P', to be checked at the last executed statement instance in V_{end}. To form P', for each polluted combination found above, we negate the Boolean value of each member and take the conjunction. This will be a condition to satisfy P. Take the subjunction of all such conjunctions obtained from different polluted combinations, we obtain P', which implies P. We rename P' to P for the next iteration to perform compatibility check for each explored path. (Each atomic proposition in the new P will be evaluated when the corresponding statement instance gets executed. However, for convenience, we will regard the last executed statement instance, denoted by p, in V_{end} as the point for compatibility check.) Our core algorithm iterates Steps 1-3 until the analysis domain cannot be narrowed further.

One can see that if we find all polluted combinations in Step 2, then *P' constructed in Step 3 will be equivalent to P.* On the other hand, there is no need to explore a combination if it contains a subset that is already found to be polluted, because the conjunction formed for the subset is weaker but still implies P. So it is natural for our core algorithm to test the singletons in the mdfs first and then test combinations of two (if necessary), and so on, essentially climbing up the lattice formed by the power set.

The formation of a new correctness property presented above is rather different from that used in our previous work [12], where a new property is always formed regardless of the kind of polluted entities. For each polluted entity (whether a branch decision or a value assigned), an atomic proposition is created to accept any value but the original one. The conjunction of these atomic propositions is then used as the correctness property for the new analysis domain. Although easier to implement, this conjunction does not generally imply P. One can use a constraint solver to find certain sufficient conditions to substitute for P. However, in general it will not be equivalent to P. Hence, we currently do not replace P by a new property in such cases.

The fault report produced by the core algorithm includes all polluted entities found during the process. However,

only those polluted entities found during the last iteration are marked as the suspected root causes, and their corresponding source code snippets are reported as the suspected error source. For failures caused by missing statement faults, the statements that use wrong operands due to such faults are listed in the error report, even though those statements themselves may be correct. Other polluted entities are included in the report to show how the fault is propagated. These are a subset of the transitive causes, but not all, because we have skipped many intermediate nodes to speedup convergence. If desired, such omitted intermediate nodes can be easily added to the report. Note that instances of input statements are never included in the fault report as we assume they are correct, but the first usage of the input value will be marked as the suspected root causes.

4.2 Details of Sparse Symbolic Exploration

Algorithm 5 presents implementation details of our sparse symbolic exploration. The algorithm is applied to a given combination of statement instances in the current *mdfs*, with a set E containing the new symbols introduced for the assignment statement instances. Note that alterations of branch statements in the given combination will result in paths different from that in the failing trace. Our algorithm uses the altered branch decisions for these statement instances as part of the initial concrete values to drive the sparse symbolic exploration, but it uses the original concrete values in the failing trace for symbols in E (line 1). Branch decisions henceforth will be made depending on the concrete values in the branch conditions. Obviously, such "concolic execution" [15] may or may not reach p, and a path reaching p may or may not lead to satisfiable $P \wedge C$ as the result of the execution. If $P \wedge C$ is satisfied, we say E is *polluted*, which is equivalent to saying that the given combination is a polluted combination, or that the combination is polluted (line 2–6).

Algorithm 5. Algorithm of Sparse Symbolic Exploration

Input — E: A set of symbols for currently altered values assigned by statement instances; P: a set of constraints specifying certain expected properties at program point p.
Constants — tBOUND: an upper bound on time spent on exploring a given path.
Key Variables — BN: # of branch instances encountered in a path; pBOUND: an upper bound on the length of paths to explore, default set to 10*BN; C: a set of path constraints; pc: a constraint in C; VS: a set of concrete values found by constraint solver to activate an execution path; DQ: a queue that stores paths to continue exploration later, represented by the branch constraints for each path and the index of the first branch to start next exploration.
Output: *True* if E is polluted, *False* otherwise.

```
1  Rerun the program from the beginning, letting each symbol
   in E use its concrete value obtained in the failing trace and
   executing all statements that are dependent on E
   symbolically;
2  If the given combination contains no assighment statements,
   let C be True. Otherwise, C = the set of path constraints col-
   lected by the symbolic execution at encountered branches
   until reaching p, or, in case p is unreachable, until execution
   terminates. In the latter case, skip the following if statement
   and jump to L1;
3  if E is empty (i.e., the combination contains only branches)
4    (P is a tautology)? return True: return False;
5  else if C is compataible with P
6    return True;
   endif
7  L1: BN = # of branch instances executed in the current run;
8  if Explore(C, 1) return True;
9  while DQ is not empty do
10   Pop the first element (C', id) from DQ;
11   if Explore(C', id)            return True;
   endwhile
12 return False;

   Explore (C, id) {/* id is the index of a path constraint in C */
13 pcn = # of the path constraints in C;
14 for i = id to pcn do
15   pc = ith path constraint in C;
16   VS = a set of concrete values for E that satifies C[pc/pc];
17   if VS is empty
18     Let C_i include the first i-1 constraints in C;
19     VS = a set of concrete values for E that satisfies C_i ∧ pc;
     endif
20   if VS is not empty
21     C_0 = C[ pc/ pc];
22     Rerun the program from scratch, letting E use the val-
     ues in VS and executing statements depending on E
     symbolically.
23     Kill the execution if its time cost exceeds tBOUND and
     continue to next iteration of the for loop;
24     Let C' be the set of path constraints collected until
     reaching p, or, in case p is unreachable, until execution ter-
     minates. In the latter case, if C_0 is compatible with C', add
     <C', i+1> to DQ and continue to next iteration of the for
     loop; otherwise, skip the following if statement and jump
     to L2;
25     if P is a tautology or C' is compatible with P
26       return True;
27     L2: bn = # of branch instances executed in the current
       rerun;
28     if bn > pBOUND continue;
29     if Explore(C', i+1) return True;
     endif
   endfor
30 return False;}
```

If E is not found to be polluted after exploring a chosen path, we use a constraint solver to find a set of concrete values for E to satisfy $C[pc/pc]$, which is C with exactly one of the constraints, pc, negated (line 16). If such concrete values do not exist, we then try to find a set of concrete values to simultaneously satisfy the set of constraints in C that are already fixed (i.e., the alternatives are already explored without success) and the constraint pc, as listed in line 18–19. Once the set of concrete values is found, we rerun the program from scratch, letting E use the values in the set and executing each statement instance depending on E symbolically (line 22). This rerun is monitored by an external script and will be killed automatically if its time cost exceeds a predefined bound (line 23). In our experiments, we set the bound to 10 seconds.

If the execution reaches p within the time limit and the set of constraints C'' generated at p is compatible with P, the algorithm termintes after marking E as a *polluted*

combination; otherwise, it will continue path exploration (line 24–26).

Note that, in a recently published program repair approach known as Angelix [16], concolic execution is used in conjunction with constraint solution to automatically repair expressions that are suspected to be erroneous. Like Algorithm 5, Angelix introduces new symbols to represent the values of instances of such expressions during concolic execution. The most important distinction, however, is that Angelix exhaustively explores all paths in order to allow the repairs to be synthesized through constraint solving. In contrast, the statements whose instances (in E) are subjected to our algorithm are usually not faulty themselves. Instead, they may have obtained faulty values through propagation. With the MDFS approach, we simply need to determine whether the state is polluted and, therefore, we can short-circuit the path exploration as soon as the state is found to be faulty.

Another important novelty of Algorithm 5 is the way it selects the next path to explore and performs concolic execution. Notice that only those paths that reach p are useful for finding polluted entities. Therefore, if our algorithm finds it unlikely to obtain new paths that can reach p by performing constraint negation on a certain path, such a path is stored in a deferred queue. The algorithm then switch to exploring other paths that are more likely to reach p. The deferred queue will not be checked or explored unless all other paths have been explored without E being marked as polluted. Due to these important short cuts, we call the approach in Algorithm 5 sparse symbolic exploration.

In Algorithm 5, if the current path fails to mark E as polluted, several steps are taken to select the next set of paths that are most likely to reach p. If the current path reaches p, C′ will be explored first. Otherwise, the algorithm makes choice by testing the compatibility of C and C_0 (line 24). If the last executed path (i.e., the one C is collected from) reaches p and the current path C' does not, then the compatibility of C' and C_0 implies that the latter path does not encounter any of the branch instance in C after negating pc (Otherwise, there would exist a subpath in C' that satisfies a subsequence of path constraints in C, making the path reach p. which is a contradiction.) In this case, there exists a chain of control dependence edges (in the static PDG) from the branch statement associated with pc to p. Therefore, any path generated from C' by making a different branch decision after pc would either fail to reach p or must take mutiple iterations of the enclosing loops (if any) before reaching p. We put C' in a queue (DQ) to defer its further exploration. We store the branch constraints of C' as well as the index of the first branch constraint that follows the negated pc. After this, we continue by checking to see whether by negating the remaining path constraints in C we can find a path to reach p.

If neither C nor C' reaches p, then the compatilities between C' and C_0 implies that C' contains either all or none of the branch instances in C after the node associated with the negated pc. In the former case, any path generated from C' by making a different branch decision after pc can also be generated by making a different descison for the corresponding branch instance in C, and the algorithm will perform further exploration on C first. In the latter case, the branch instance that pc corresponds to is not a direct dominator of p. The algorithm will perform further explorations on C' if it finds no path to reach p and make P compitible by negating the remaining path constraints in C.

Another bound, pBOUND, is imposed on the number of branch instances executed in the unexplored paths (line 27–28). In our experiment, we set the default value of pBOUND to be 10 times of the branch number executed by the original failing execution. If these limits are exceeded, then the algorithm may miss certain feasible paths and cause the core algorithm to miss certain polluted combinations. If this happens, then the new correctness properties formed in Step 3 of the core algorithm will be sufficient conditions for the original correctness property P, but not guaranteed to be equivalent.

4.3 Extension to Multiple Faults

Suppose a program contains multiple faults, i.e., multiple faulty statements, and they together cause a failure during the execution. In the dynamic dependence graph, we may find multiple initial faulty program states, each at the head of a dependence chain that leads to the failing operation. We discuss how to apply the core algorithm in this situation.

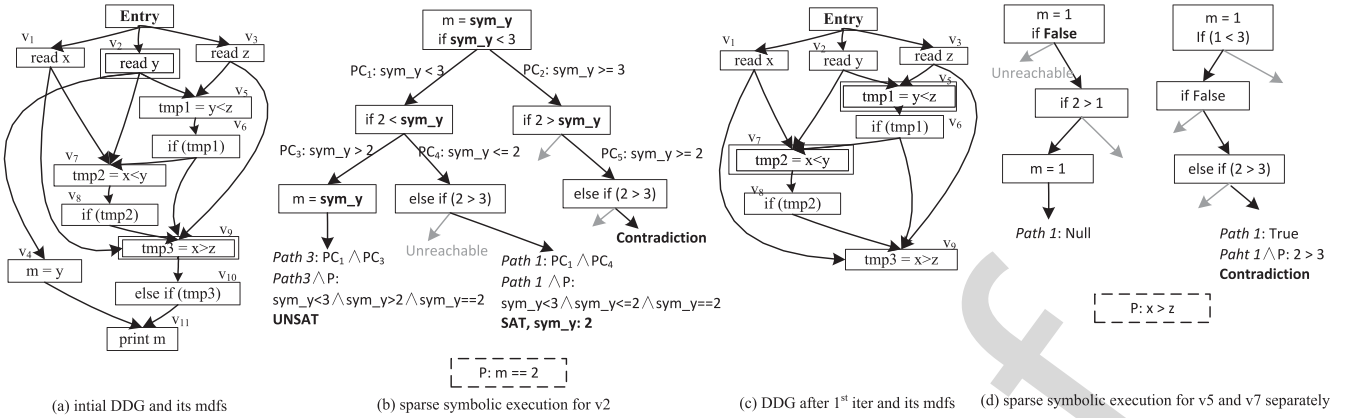
When we perform state exploration on the statement instances in the *mdfs*, we apply Algorithm 5 to find out whether they contain any polluted entities. If not, then the program states at the *mdfs* are considered to be correct, and by Claim 3.2, our method still determines that all root causes are located between the *mdfs* and the failure site. If, however, polluted entities are found, then two possibilities exist. It could mean that the program state at the *mdfs* was correct before altering, hence all initial faulty states exist between the *mdfs* and the failure site, but the altering makes the failure disappear. The second possibility is that the program state at the *mdfs* was faulty but can be corrected by altering of the value produced by a certain statement instance. Hence *at least one* root cause of the failure exists prior to the *mdfs*. Our algorithm still accepts the latter possibility.

Immediately before the algorithm terminates, we may have one of the two following situations:

1) All initial faulty states are located on the same side of the *mdfs* in the final iteration;
2) Some of the initial faulty states are on one side and the others are on the opposite side of the *mdfs*.

In the first situation, we would have found all the root causes for the failure. In the second situation, however, we would miss those root causes located on the wrongside. To mitigate this weakness, the programmer can rerun the program under the same input after correcting the root causes found by our technique. If the missed faulty statements cause a failure again, the MDFS-based technique can be reapplied. Further, in order to find latent faulty statements, it is best for the user of our tool to conduct extensive tests using various inputs. This would improve the chance that the missed root causes may be found when debugging other failures.

Let us revisit the example in Fig. 1, which has two faults (in statement 2 and statement 6, respectively). Under input $(\mathrm{x}, \mathrm{y}, \mathrm{z}) = (2, 1, 3)$, both faults are triggered, causing the erroneous value, 1, to be printed by statement 13, when the correct output is supposed to be 2.



(a) intial DDG and its mdfs (b) sparse symbolic execution for v2 (c) DDG after 1st iter and its mdfs (d) sparse symbolic execution for v5 and v7 separately

Fig. 4. An example of locating multiple faults.

In the dynamic dependence graph of the failed run, the statement instance that produces the erroneous output is represented by node v_{11}. Our method calculates an *mdfs* for $DDG < ENTRY, v_{11} >$, which contains two nodes, v_2 and v_9, boxed in double-lined rectangles in Fig. 4(a). Altering the state at v_2, the input value of y is replaced with a fresh symbol *sym_y*. Sparse symbolic exploration is then performed on *sym_y* to see if any alternative value exists for *sym_y* that makes the execution reaches v_{11} and obtains a state at v_{11} that satifies the correctness property "$m == 2$". As shown in Fig. 4(b), the exploration first lets *sym_y* take its original value and then symbolically executes all statements that are dependent on *sym_y*, resulting in a set of path constraints at v_{11} that consist of $PC_1\,(sym_y < 3)$ and $PC_4\,(sym_y <= 2)$. The correctness property "$m == 2$" placed at v_{11}, after symbolic substitution, becomes "$sym_y == 2$". Using a constraint solver, we find that this set of path constraints is compatible with the correctness property "$sym_y = 2$", which makes v_2 a polluted entity. (Recall that we assume the input files to be always correct, but the *read* statement may constitute a fault.) We do not need to explore other paths originating at v_2. (Note that, had we continued to explore other paths, the path constraints would not have been compatible with the correctness property, as shown in Fig. 4(b)). State exploration is next applied to v_9, showing it to also be a polluted entity. Since both v_2 and v_9 have been considered as polluted, no exploration needs to be performed on their combination. By Algorithm 4.1, the analysis domain is first narrowed to $DDG < ENTRY, v_2, v_9 >$. However, since v_2 depends on user inputs alone, it is already at the head of a dependence chain leading to the failure site. Therefore, our method reports v_4, i.e., the first usage of *y* after the current *mdfs*, as a root cause. The analysis domain is thus simplified to $DDG < ENTRY, v_9 >$ (Fig. 4(c)) with a new correctness property "$x > z$" generated at v_9. The search continues towards the entry of the trace by finding an *mdfs* for $DDG < ENTRY, v_9 >$. This *mdfs* contains v_5 and v_7. As shown in Fig. 4(d), neither v_5 nor v_7 alone is found to be polluted. Next, state alteration is attempted on the combination of v_5 and v_7, but negating the truth value of v_5 makes v_7 unreachable. So this combination does not turn out to be polluted. Since no polluted entity is found in the second iteration, the current *mdfs* becomes V_{begin} for the next analysis domain, for which we find no further cuts. The algorithm terminates after reporting statements 2 and 6 as root causes of the failure observed at statement 13.

5 Implementation

We implement our method in the framework of Valgrind [17], which is a suite of tools for debugging and profiling based on dynamic binary instrumentation. We write a new plug-in for Valgrind to construct DDGs and to find *mdfs*. We add a new module to the Diablo tool [18] to find static direct control dependence offline, and the results are mapped to Valgrind by instruction addresses. As Diablo has limitations in flow analysis, we add the static summary control dependence analysis[1] to the Open64 compiler [19]. The analysis results are mapped to Valgrind's VexIR based on the line number of source code before the program execution is traced. When constructing DDGs, interprocedural summary control dependencies are omitted to save time and space. Experimental result in Section 6 will show that computing intra-procedural summary control dependencies is sufficient for locating faults for the test cases used in our experiments. For finding *mdfs*, we use the classical Edmonds-Karp algorithm to generate minimum cuts, with time complexity of $O(|V| \times |E|^2)$. Although slower than the push-relabel method, which runs in $O(|V|^3)$, and other newly discovered algorithms, this algorithm is advantageous in handling sparse graphs such as those analyzed by our tool. The number of edges and the number of vertices in our examined DDGs (or sub-DDGs) are often of the same order of magnitude. Statement instances in the *mdfs* are mapped to their corresponding source code, using the debugging information generated by the GCC Compiler, to support the MDFS-based state alteration and exploration. We turned off GCC optimizations in order to minimize the differences between the source code and the binary. Algorithm 5 in our sparse symbolic exploration is implemented on top of Crest-z3 [20], in which Crest [21], a concolic test generation tool for C, instruments the target program to perform symbolic execution concurrently with the concrete execution, with our own path selection order. The generated symbolic constraints are translated into an SMT format before solved by using z3 [22] to generate input that drives

1. Both direct control dependence and summary control dependence are detected in our previous work [12] without explicit discussion on implementation.

TABLE 1
Faults Used for Studies

BugID	Program	Patch Date	Fault Desc.	Mutli?
1	grep-2.6	2010-03-25	incorrect goto	No
2	grep-2.13	2012-07-08	weak guard	No
3	grep-2.18	2014-04-21	strong guard	No
4	grep-2.19	2014-05-29	missing assignments	Yes
5	gdb-6.8	2008-12-19	missing function calls	Yes
6	gdb-6.8	2008-12-31	design error	Yes
7	gdb-6.8.50	2009-09-22	missing branch	No
8	gdb-7.0	2010-01-25	mutation operator	No
9	sed-4.2	2009-05-11	missing guard	No
10	sed-4.2.2	2014-09-06	wrong assignement	No
11	tar-1.22	2009-07-30	missing function call	No
12	tar-1.22	2009-07-30	weak guard	No
13	tar-1.22	2009-10-07	design error	Yes
14	tar-1.23	2010-03-17	design error	No
15	tar-1.23	2010-06-28	call at wrong place	Yes
16	tar-1.26	2013-08-04	design error	Yes

the test execution along chosen paths. We add several utilities to Crest to provide functionalities needed by Algorithm 5, including support for constraint generations for dynamic memory operations and simple pointer arithmetic expressions (e.g., malloc, free, *(pointer + constant), pointer - pointer) when a pointer is marked symbolic [23].

6 Evaluation

We collect a pool of 16 real bugs of different types that have been reported in the last decade from several sources, including Sed, Grep, Tar and Gdb. Characteristics of these bugs are listed in Table 1. For convenience, we identify each bug with a number (under Column "*BugID*"). Column "*Program*" shows the name and the version of the program that each bug belongs to. Columns "*Fault Desc.*" and "*Patch Date*" list the bug type as decribed in the program's bug-update mailing list and the date when a patch for the bug is provided by the developer, respectively. Column "*Multi?*" indicates if the bug involves multiple faults. In our experiment, we used the inputs from the bug-update mailing list to reproduce the failed run.

TABLE 2
Fault Reports for MDFS Methods

Bug ID	#State Alter	Fault Report			Time (sec)
		Where	Dist	#Stmt	
1	3	dfasearch.c @ 301	1	2	34.27
2	12	dfasearch.c @ 413	1	5	79.61
3	11	dfa.c @ 678-687	2	13	86.08
4	12	dfa.c @ 3979	0, 1	3	88.01
5	15	valops.c @ 1921	1	2	149.71
6	10	infrun.c @ 3538	1	2	105.87
7	42	eval.c @ 1516	0	1	378.43
8	2	c-valprint.c @ 189	0	1	70.94
9	8	compile.c @ 505	0	1	35.28
10	4	execute.c @ 1570	0	1	31.18
11	12	create.c @ 1437	1	4	101.10
12	10	create.c @ 1380	0	1	72.62
13	33	create.c @ 1675-1687	0	1	268.16
14	7	unlink.c @ 104, 105	2	4	142.96
15	8	list.c @ 94	1	2	189.45
16	14	buffer.c @ 725,743	0	3	162.88

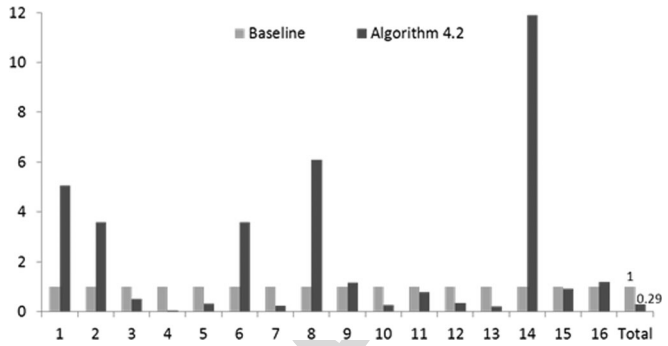


Fig. 5. A comparison between our previous work (Baseline = 1) and sparse symbolic exploration under Algorithm 5.

6.1 Overall result

Table 2 describes the fault report generated by our method and the time spent on fault localization for each test case. Column "*#StateAlter*" shows the number of states automatically altered by our algorithm in order to find the root cause for each failure. Column "*Where*" identifies the reported location of the root cause in the source code. If the fault is not at the reported location, our tool orders the statements based on their dynamic dependence distance from the root cause candidate, so that those statements that are closest to the reported location can be inspected first. Column "*Dist*" shows the distance, measured by the length of dynamic dependencies, between the suspected root cause and the true faulty statements. The number of statements needed to be examined by the programmer before finding the fault is recorded in Column "*#Stmt*", which shows that, for 10 out of the 16 benchmarks we studied, no more than two statements need to be examined before finding the real faults. Column "*Time*" lists the time spent on generating a fault report for each test case, including the time spent on tracing, DDG construction, MDFS generation and sparse symbolic exploration. It also includes the time spent by Crest on instrumenting the source file and generating the object file for sparse symbolic exploration. Multiple source files for the same project are merged into a single source file using CIL [24] before the instrumentation. For the four test cases from Gdb, source files whose functionalities have nothing to do with the tests are excluded from the merged source file in order to save time of instrumentation. Seven of the test cases (BugID 5∼8, 14∼16) take advantage of the checkpointing scheme introduced in our previous work [12] during DDG construction. The construction would otherwise have taken more than an hour.

Fig. 5 compares time for state exploration under Algorithm 5 against the baseline, which is the time (normalized to 1) consumed by our previous method [12]. The time for the baseline is spent on running CBMC and calling MiniSAT2 to solve the constraints generated by CBMC. It, however, does not include the time spent on manually extracting and modifying the source code of the current analysis domain for CBMC. (Since CBMC is not designed for concolic execution, we needed to modify the program by hand to make CBMC perform concrete execution on statement instances that have concrete values for all of their operands. Symbolic execution is performed on remaining statement instances.) The time consumed by Algorithm 5 is spent on instrumentation of the entire code (since we re-run

from the beginning in each iteration) and on path exploration (including the time to call z3). It, however, does not include the time spent on merging the whole project into a single source file. For the test cases that were used in both this and the previous experiments, the time consumed by the old method is re-collected using the most recent versions of CBMC and MiniSAT2.

Fig. 5 shows that Algorithm 5 outperforms the old state exploration method in 10 out of the 16 test cases. For BugID 2 and 16, the old method fails to locate the root causes because CBMC generates constraints that are too complicated for it to solve. For the remaining four test cases, instrumentation takes on average 71.99 percent of the total time for state exploration effort under Algorithm 5, because the analysis domain soon becomes small, which, however, still takes several iterations to converge. Therefore, the time to reinstrument the whole program overshadows the time for path exploration. Nonetheless, when cumulated over all test cases, the new method takes only 29% of the time taken by the old one.

Six of the test cases in our benchmark have multiple faults that simultaneously cause the individual failures. These can be divided into three categories:

1) (BugID 15, 16) The failure is triggered by multiple faulty statements that reside in different functions or different basic blocks in the same function. (This category includes cases of misplaced statements. It also includes the special case in which fixing one fault triggers another.)
2) (BugID 4) The failure is triggered by multiple faulty statements that reside in the same basic block. Each of the faulty statements directly affects a different value assignment;
3) (BugID 5, 6, 13) The failure is triggered by multiple faulty statements in the same basic block and all these faulty statements directly affect the same value assignment.

As shown in Table 2 (column "*Dist*"), our method is able to locate transitive causes that are one dependence step away from the real faults for three of the test cases (BugID 5, 6, 15) and locate one of the real faults and a transitive cause that is one dependence step away from the other for another test case (BugID 4). In the remaining two cases (BugID 13, 16), all root causes are found. For BugID 13, all of the statements in the final analysis domain are reported by our method as root causes, among them the actual faulty statements. The program input for BugID 16 consists of two *tar* commands. Fixing the fault that is triggered in the second *tar* command consequently triggers another fault, making the first command fail. Both faults must be fixed in order to remove the failure.

6.2 Discussions

Accuracy of Suspected States. In Section 4, our algorithm determines how to narrow the analysis domain according to the result of sparse symbolic exeploration. Due to the lack of information on the correct program state at statement instances belonging to the current *mdfs*, our method uses heuristic criteria motivated by Claims 3.2 and 3.3 to determine the search direction. If a statement instance is found to be a polluted entity by our sparse symbolic exeploration, we assume its state to be incorrect. If one or more polluted entities are found in the current *mdfs*, the search continues towards the entry of the analysis domain. We now examine how often our tool judges incorrectly about the state in the experiments.

TABLE 3
The Frequency to Misjudge a Suspected State

BugID	#State	#Misjudge-I	#Misjudge-II
1	3	-	-
2	12	-	1
3	11	1	-
4	12	-	-
5	15	1	-
6	10	1	-
7	42	-	-
8	2	-	-
9	8	1	-
10	4	-	-
11	12	-	-
12	10	-	4
13	33	3	-
14	7	-	-
15	8	-	-
16	14	-	-
Rate	-	3.45%	2.46%

Table 3 lists the frequency of our method to misjudge a state for the individual test cases from Table 1. Row "*#State*" lists the number of states to be checked by our method. To collect the frequency data, we compare two versions of each program, one with the failure corrected and one without. For each state subject to alteration by our MDFS-based scheme, we find the correct state information from the successful run of the same program with the failure fixed. We then use the correct state to determine whether the decision made by our method is correct, i.e., whether our method misjudges a correct state to be incorrect or vice versa. Note that, if a state to be checked by our method does not exist in the fixed run, then such a state is regarded as an incorrect state. Overall, for the 16 test cases used in our experiments, of all state instances that are subject to state alteration, the average rate of our method mislabeling a correct state to be incorrect is 3.45 percent ("*Misjudge-I*" in Table 3), and the average rate of our method mislabeling an incorrect state to be correct is 2.46 percent ("*Misjudge-II*" in Table 3).

There may be two reasons for mislabeling a correct state as incorrect: (1) A certain statement instance that does not belong to any propagation chain from an initial faulty state to the failure happens to have an alternate value that makes the constraints generated at a certain program point compatible with the correctness criteria given at the same point, causing the statement instance to be identified as a polluted entity. This scenario occurred with three test cases (BugID 3, 9, 13) in our experiment; (2) For faults of missing statements, an unintended reaching definition between two otherwise correct statements may cause a statement instance to be mislabeled as a polluted entity. This has occurred to two test cases (BugID 5, 6). Note that the mislabeling in both scenarios can cause a wrong search direction only if it happens to all of the statement instances in the current *mdfs*.

There are also two reasons for mislabeling an incorrect state to be correct: (1) The state alteration causes the

program execution to take a different path that does not reach the failure site any more. This has happened to BugID 12; (2) The specification of the correctness property given in the failure site is incomplete. This has happened to BugID 2 in our experiment. This test case takes two incorrect output actions. The first one is to produce an output at a wrong program point but incidentally has the same value as the expected output. Hence, unless one inspects the execution trace, the incorrect behavior is hidden. Our MDFS method identifies polluted entities based on the specification given at the program point where the second incorrect output action takes place. This caused a mislabeling during the second last iteration, making the analysis domain converge to the wrong direction and miss the real fault.

Impact of Summary Control Dependences. Detecting summary control dependences for DDG construction has made impact on 9 out of 16 test cases in our expriment. The following two positive effects are found:

1) (BugID 2, 3, 8, 9, 16) Detecting summary control dependences exposes the root cause statement in the initial analysis domain; and
2) (BugID 4, 5, 7, 12, 16) Statement instances added to the initial analysis domain by detecting summary control dependences play a critical role in obtaining the correct analysis domain, since these instances become members of *mdfs* and they turn out to be the only polluted entities in a certain iteration.

On the other hand, inclusion of summary control dependences can potentially cause mislabeling of a state as erroneous, especially if these dependencies do not turn out to be "real" due to the conservative nature of static analysis (BugID 3). Table 4 presents details for specific test cases.

Note that, we only detect intra-procedural summary control dependences when constructing DDGs. Though it is sufficient for locating the root cause by detecting intra-procedural summary control dependences in our experiments, considering inter-procedural summary control dependence may give us more clues about how the fault propagates. For BugID 16, the fault triggered by the second *tar* command prevents some statements from being executed, which results in the erroneous value assigned to *record_start->buffer*. This fact is obscured without inter-procedural summary control dependence, because *record_start* is a global variable. However, detecting inter-procedural summary control dependence could result in an extremely large and inaccurate DDG. Considering the trade-off between accuracy and efficiency, our tool only considers intra-procedural summary control dependences when constructing DDGs.

Usage of Additional Information. In the second step of Algorithm 4, users can provide additional information to refine the result of state exploration. Additional output that is known to be correct may be used to serve as additional correctness criteria to guide analysis-domain narrowing, as discussed in our previous work [12]. User annotations may also be helpful. Let us revisit the example in Fig. 1(a). Suppose statements 2 and 6 are corrected as indicated in the comment lines. Instead, suppose statements 7 and 10 are exchanged by mistake. The execution under the given input would still generate the incorrect result of "$m = 1$". This time, the execution trace contains instances of statements 1-4, 6-7, and 13. Suppose the *mdfs* consists of the instances of statements 2 and 6. Our algorithm will find the latter to be polluted, because switching its *if* condition to *false* would cause statement 10 to be executed, which actually causes the correct output (i.e., $m = 2$) to be printed. Without additional information, our method would incorrectly continue to search towards the program entry, missing the actual faults. However, had the user marked the *if* condition to be *true* under the given input, our search would have continued correctly towards the failure site.

TABLE 4
Impact of Detecting Summary Control Dependences

BugID	Details
2	The detection of summary control dependences makes the root cause to appear in the initial analysis domain, although our method fails to identify it as a polluted entity.
3	The root cause appears in the initial analysis domain due to summary control dependences. In 2nd iteration, a statement instance added to the analysis domain due to summary control dependences is misjudged as a polluted entity. (However, the mislabeling makes no effect on the converging direction as there are other real erroneous states found in the same iteration.)
4	The only polluted entities found in 3rd itertaion are added to the analysis domain due to *summary control depenences*.
5	The *if* condition directly affected by the missing code is added to the initial analysis domain due to summary control dependences, and it is then recognized to be the only polluted entity in the last iteration.
7	Similar to BugID 5.
8	The root cause appears in the initial analysis domain due to summary congrol dependences, and it is successfully identified to be the only polluted entiy in the last iteration.
9	Similar to BugID 8.
12	Similar to BugID 4.
16	Both the root cause and the polluted entities found in 2nd iteration appear in the initial analysis domain due to summary control dependences.

6.3 Comparisons

Comparisons with Other State-based Techniques. Among previous state-based approaches, CT [1], [25] uses delta debugging [26] to study the differences between successful runs and failing runs in order to identify a chain of variables and values that causes the observed failure. This method is time-consuming because the state comparison and alteration must be applied to a large number of trace points along the failing trace. To mitigate the time complexity, the *predicate switching* technique [2] constrains state changes to predicate instances only, and the *value replacement* method [3] constrains the sets of alternate concrete values in order to make the search space more acceptable.

To compare our technique with predicate switching and value replacement, we implement both techniques based on

TABLE 5
Fault Reports Generated with One Hour Time Limit

Dist	MDFS	Predicate Switching	Value Replacement
0	7	3	-
1	7	3	-
(1,3)	2	1	-
Miss	-	9	All

the descriptions in open literature [2], [3], as the source codes for these techniques are not publically available. Minor changes are made to speed up the experiments. For the sake of fairness, both these two reference techniques and our MDFS method are implemented within the framework of Valgrind and they all share the same preprocessing step.

The predicate switching method in reference [2] starts its search from the failure site and stops when the first "*useful critical predicate*" is found. If switching a particular predicate instance can make the failure disappear, then it is called a critical predicate. It is the tool user's responsibility to decide whether the predicate is useful for locating faults. We simply find all critical predicates within a pre-determined time limit. Among these, the one that is closest to the fault is chosen to be the first useful critical predicate. Note that, if switching a predicate instance can cause an infinite loop, then that instance is excluded.

For the value replacement method, we implement the improved version described in [3]. For each failing execution trace, this improved version starts its search for *interesting value mapping pairs* from the beginning of the failing trace. (If replacing the value assigned by a statement instance can make the failure disappear, then the original value and the replacing value form an *interesting value mapping pair*, abbr. IVMP.) For each statement instance that appears in the analysis domain, we try four alternate values, namely the lowest alternate value possible, the highest alternate value possible, and the two alternate values closest to the original from each side. In the absence of a training set, the alternate values for each statement instance are collected from the other instances of the same statement in the same failed run. Among all IVMPs found for a certain failing execution trace, the one that is closest to the fault is reported to be the root cause.

Table 5 compares the fault reports generated by those three methods mentioned above within the time frame of one hour. This time frame includes neither the time spent on tracing nor that spent on generating alternate sets of values for each statement instance. Row "*Miss*" records how often a method fails to identify any root cause or even a transitive cause. Without a carefully prepared training set, the improved value replacement method fails in all test cases. Predicate switching misses the root cause for 9 of the test cases within the one hour time frame. However, it is more efficient than our method for failures that are due to incorrect branch composition (BugID 12).

Results from the test cases highlight two main advantages of our new method. First, our iterative algorithm converges deterministically and the analysis domain shrinks quite rapidly. Furthermore, we believe that by using both dependence information and symbolic analysis to identify polluted entities, our method is built on a more solid theoretical foundation than the heuristics used in the other two methods discussed above. Note that like our method, predicate switching and value replacement require the desired state at the failure site, such that the search space for the faults can be defined by all statement instances that may affect that program state.

TABLE 6
Fault Reports for Mutation-Based Methods

Bug ID	#Mutant	#Passed	#Stmt	Top-5
1	275071	686	2	-
2	300641	4	-	Unrelated
3	292440	32	-	Unrelated
4	286511	1272	155	Rank5: tran. cause
9	79197	-	-	-
10	135584	-	-	-
11		4	1	-
12	328956	98	9	-
13		-	-	-
14	353735	-	-	-
15		96	-	Rank2: tran. cause
16	399445	11	2	-

Comparison with mutation-based approaches. Mutation-based methods [27], [28], [29] can be applied to any combinations of failed and passed test cases. These methods do not require execution trace or user-provided specifications on the failure site. They assign a suspicious score for each statement based on mutation analysis such that a higher score is assigned if the statement is deemed more likely to be faulty. The user is to inspect statements in the order of the suspicious scores. To compare our method with mutation-based methods, we use ProteumIM 2.0 [30], which is an upgraded version of the mutant generator used in [29], to generate mutants for each of the buggy program in Table 1. As in [29], for each mutation point in a statement, we make the tool generate exactly one mutant for each supported mutation operator. Scripts are written to collect mutants that make the failed test case pass. We also modify the tool to evaluate a suspicious score for each statement in the faulty program using the formula prescribed in [29].

For all 16 test cases in Table 1, we feed the merged source file generated by CIL to the mutant generator. By manually removing or modifying certain syntax (such as removing the *inline* keyword and renaming structures, fields and variables), we are able to make 12 of the test cases acceptable by the tool. The running time of all these test cases through the tool takes over a week, and the result is listed in Table 6.

Columns "*#Mutant*" and "*Passed*" show the total number of mutants generated for each buggy program and the number of mutants that make the failed test case eventually pass, respectively. Following the inspection order based on the suspicous scores, column "*#Stmt*" counts the number of statements that must be examined before encoutering the real fault. If the real fault is not included in the list of suspects, one can check column "*Top-5*" to see in which ways the top 5 suspicious statements in each test case are related to the real fault.

Table 6 shows that, for 3 out of 12 test cases we have studied (BugID 1, 11, 16), using the report given by the mutation-based method, the user can locate the root cause by manually examining fewer than two statements, which

TABLE 7
Comparison with Other Divide-and-Conquer Strategies

BugID	Single-Step	Bisection	MDFS
1	6	17	3*
2	20	10*	12
3	23	12	11*
4	50	8*	12
5	25	37	15*
6	1*	39	10
7	2*	30	42
8	16	63	2*
9	13	79	8*
10	6	7	4*
11	36	24	12*
12	4*	21	10
13	1*	35	33
14	24	3*	7
15	22	18	8*
16	25	18	14*
Time	1.43	1.63	1

is a more accurate result than the fault report generated by the MDFS method. However, for seven of the remaining test cases, the mutation-based method fails to identify any root cause. Although it is able to place a transitive cause in the top 5 suspicious statements for two of the test cases, one of the reported transitive causes is dozens of dependence steps away from the real faulty statement and the other is seven dependence steps away from the real fault. On the other hand, should additional mutations are generated by the mutation tool at each mutation point, the mutation-based method may be able to identify additional faults.

Comparison with Other Divide-and-Conquer Strategies. MDFS is not the only way to partition the analysis domain and make it converge deterministically. We compare the MDFS-based strategy for narrowing the search with two alternatives that select nodes in the DDG for testing polluted entities based on their distances from the end points of the current analysis domain. (For fairness, sparse symbolic exploration is performed in all experiments that are conducted for this comparison.)

The first alternative, called the *single-step strategy*, first tests polluted entities on nodes that are one hop away from the failure site. In each iteration that follows, if the root causes are not yet found, the number of hops away from the failure site is incremented by one for choosing the nodes to test polluted entities. In the best case, the search may be completed in just one iteration, but in the worst case, it may take as many iterations as the maximum length of the dependence chains in the DDG with the failure site as the sink point.

The second alternative, called the *bisection strategy*, always tests polluted entities on nodes that are exactly the midpoints between the entry points and the end points of the current analysis domain, until no further partition is possible.

Table 7 lists the number of suspected states extracted by the MDFS-based strategy and the two alternatives, along with the total time spent under each strategy on state exploration (Row "*Time*", with the MDFS strategy normalized to 1). For each test case, the strategy that examines the smallest set of suspected states is marked by a star on the upper-right corner of its state number. As we can see, the MDFS strategy is the most efficient.

6.4 Limitations

In spite of the accuracy and efficiency demonstrated through the testing results shown above, our MDFS-based scheme has a number of weaknesses.

First of all, since our method is centered on the concept of dynamic dependencies, it does not deal with faults that do not get propagated through dependence chains, e.g., certain failures that are due to incorrect type definitions. Also, for faults due to missing assignment statements, our method locate only the incorrect reaching definitions, but do not necessarily pinpoint the exact location of the missing statements.

Another potential vulnerability for our dependence-centric approach concerns the effect of multiple faulty statements in the same dependence chain, giving rise to the possibilities of two faulty statements canceling the faulty program state. Although this situation has not caused our method to fail in our test cases (which cover various fault types and include cases that have been studied by previous researchers), more extensive experiments are needed to see how serious such a situation may be in practice.

Our method shares a weakness with most of the existing methods that is, if we do not have access to the exact correctness criterion as a reference, our method may suffer from false negatives as well as false positives.

Lastly, our technique targets failures that can be reproduced under the same program input and can be captured by checking outputs or assertion violations. Therefore, it does not handle failures such as infinite loops or those produced by multithread programs.

7 Related Work

Our work is built on several important concepts that have been developed by the software engineering community. Program slicing [31], [32] has long been in use for extracting program entities (e.g., statements and states) that may be responsible for propagating faults to the failure site. Zhang et al. [33] improve the inter-procedural dynamic slicing algorithm in order to handle complex real world programs. They also provide a method to prune dynamic slices [34]. Researchers have also developed variants of slicing technique [35], [36], [37], [38] for narrowing the program segments suspected to be associated with failures. The limitation of these approaches is that they can neither automatically find the causes of failures nor give users guidance on how to find them.

In addition to the state-based fault localization approaches mentioned in Section 6.3, another method [6] has been been known to use dual slicing and a *confounding free execution model* to compare a failing run against a successful run (either under another input or obtained by predicate swiching). This method helps make the automatically generated *failure explanation* more accurate and more concise, although it shares certain shortcomings of predicate switching. Another state-based method, ANGELINA [4], identifies expressions that are candidates for error repair. An expression becomes a candidate if changing its value can fix a failed run while keeping all passing tests unchanged. Our method also discovers value changes that can remove the failure under the given input, but it only needs to

symbolically alter the values written by the statement instances in an MDFS.

We have discussed mutation-based methods in Section 6.3. Fault localization methods based on statistics can also be found in literature [39], [40], [41], [42], [43], [44], [45], [46], [47]. These methods collect execution information from both successful runs and failed runs in order to generate a suspiciousness score for each program entity. They assign the suspicious score based on how frequently a specific program entity appears in the failing and passing runs, respectively, and do not need to know where in the program a failure occurs. These characteristics are in contrast to our method, which is applied directly to the trace of a failed run, without needing information from other runs.

8 Conclusion

In this paper, we present an MDFS-based fault-localization scheme. Algorithms are developed based on a theoretical analysis on how faults may be propagated through chains of dependences during program execution. Experimental results support our main hypotheses that the proposed MDFS-based scheme for iteratively narrowing the analysis domain and the accompanying sparse symbolic exporation method can significantly reduce the time spent on exploring execution paths and on verifying predicates, while still being able to catch the faults. For the test cases collected from real world for this study, our new scheme is shown to locate faults with a high accuracy within reasonable time.

Acknowledgments

We would like to thank the anonymous reviewers whose suggestions have improved the presentation of our work. This work is sponsored in part by National Natural Science Foundation of China (61303053, 61402445, 61402303, 61521092); National High Technology Research and Development Program of China (2015AA011505); and National Science Foundation of United States (1533822).

References

[1] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 342–351.
[2] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 272–281.
[3] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 167–178.
[4] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 121–130.
[5] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," in *Proc. 32nd ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2011, pp. 437–446.
[6] W. N. Sumner and X. Zhang, "Comparative causality: Explaining the differences between executions," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 272–281.
[7] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.
[8] B. Xin and X. Zhang, "Efficient online detection of dynamic control dependence," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 185–195.
[9] B. Korel and J. Laski, "Algorithmic software fault localization," in *Proc. 24th Annu. Hawaii Int. Conf. Syst. Sci.*, 1991, pp. 246–252.
[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.
[11] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein, "Experimental study of minimum cut algorithms," in *Proc. 8th Annu. ACM-SIAM Symp. Discrete Algorithms*, 1997, pp. 324–333.
[12] F. Li, W. Huo, C. Chen, L. Zhong, X. Feng, and Z. Li, "Effective fault localization based on minimum debugging frontier set," in *Proc. IEEE/ACM Int. Symp. Code Generation Optimization*, 2013, pp. 1–10.
[13] E. Clarje, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2004, pp. 168–176.
[14] [Online]. Available: http://minisat.se/MiniSat.html Q1
[15] Koushik Sen, "Concolic testing," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2007, pp. 571–572.
[16] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, 691–701.
[17] [Online]. Available: http://valgrind.org/
[18] [Online]. Available: http://diablo.elis.ugent.be/
[19] [Online]. Available: http://developer.amd.com/tools-and-sdks/cpu-development/x86-open64-compiler-suite/
[20] [Online]. Available: https://github.com/heechul/crest-z3
[21] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 443–446.
[22] [Online]. Available: https://github.com/Z3Prover/z3 Q2
[23] J. Zhang, "Symbolic execution of program paths involving pointer and structure variables," in *Proc. Quality Softw. 4th Int. Conf.*, 2004, pp 87–92.
[24] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. 11th Int. Conf. Compiler Construct.*, 2002, pp. 213–228.
[25] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. 10th ACM SIGSOFT Symp. Found. Softw. Eng.*, 2002, pp. 1–10.
[26] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proc. 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 1999, pp. 253–267.
[27] M. Papadakis and Y. L. Traon, "Using mutants to locate "Unknown" faults," in *Proc. IEEE 5th Int. Conf. Softw. Testing Verification Validation*, 2012, pp. 691–700.
[28] M. Papadakis and Y. L. Traon, "Metallaxis-FL: Mutation-based fault localization," *Softw. Test. Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015.
[29] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: mutating faulty programs for fault localization," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation*, 2014, pp. 153–162.
[30] M. E. Delamaro and J. C. Maldonado, "Proteum/IM 2.0: An integrated mutation testing environment," *Mutation Testing New Century*, vol. 24, pp. 91–101, 2001.
[31] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 439–449.
[32] H. Agrawal and J. R. Horgan. "Dynamic program slicing," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 1990, pp. 246–256.
[33] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 319–329.
[34] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2006, pp. 169–180.
[35] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 1996, pp. 121–134.
[36] T. Gyimóthy, Á. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *Proc. 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 1999, pp. 303–321.
[37] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 189–200.
[38] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 301–310.
[39] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2005, pp. 273–282.
[40] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2005, pp. 15–26.

[41] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *Proc. 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, pp. 286–295.
[42] P. A. Nainar and B. Liblit, "Adaptive bug isolation," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 255–264.
[43] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proc. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 146–156.
[44] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 73–84.
[45] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 141–152.
[46] C. Sun and S. Khoo, "Mining succinct predicated bug signatures," in *Proc. 9th Joint Meet. Found. Softw. Eng.*, 2013, pp. 576–586.
[47] Z. Zuo, S. Khoo, and C. Sun, "Efficient predicated bug signature mining via hierarchical instrumentation," in *Proc. 23th Int. Symp. Softw. Testing Anal.*, 2014, pp. 215–224.

Feng Li received the PhD degree from Institute of Computing Technology, Chinese Academy of Sciences, in 2013. She is currently an assistant professor in the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. She is interested in program analysis and fault localization.

Zhiyuan Li received the BS degree in mathematics from Xiamen University, China, in 1982, and the MS and PhD, degrees in computer science from the University of Illinois, Urbana-Champaign, in 1985 and 1989, respectively. He is professor in computer science, Purdue University, Indiana. Before he joined Purdue University in 1997, he worked as a senior software engineer at Center for Supercomputing Research and Development in University of Illinois, Urbana-Champaign, and as assistant professor in computer science with the University of Minnesota, Minneapolis-St. Paul. Zhiyuan Li's current research primarily focuses on high performance computing and reliable computing.

Wei Huo received the PhD degree from Institute of Computing Technology, Chinese Academy of Sciences, in 2010. He is currently an associate professor in Institute of Information Technology, Chinese Academy of Sciences. His research interests include program analysis and bug detection. He is now working on building a large collaborated program analysis platform.

Xiaobing Feng received the BS degree from Tianjin University, China, in 1992, and the MS from Perking University, China, in 1996, and the PhD in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences, in 1999. He is professor in Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program optimizing, high performance computing and program analysis.

Queries to the Author

Q1. Please provide publication year in Refs. [14], [17], [18], [19], and [20].
Q2. Please provide publication year in Ref. [22].