2008

Automatic Failure Inducing Chain Computation through Aligned Execution Comparison

William N. Sumner

Xiangyu Zhang
Purdue University, xyzhang@cs.purdue.edu

Report Number:
08-023

Automatic Failure Inducing Chain Computation through Aligned Execution Comparison

William Sumner
Xiangyu Zhang

CSD TR #08-023
September 2008

Automatic Failure Inducing Chain Computation through Aligned Execution Comparison

William N. Sumner Xiangyu Zhang
Department of Computer Science, Purdue University
West Lafayette, Indiana 47907
{wsumner,xyzhang}@cs.purdue.edu

Abstract

In this paper, we propose an automated debugging technique that explains a failure by computing its causal path leading from the root cause to the failure. Given a failing execution, the technique first searches for a dynamic patch. Fine-grained execution comparison between the failing run and the patched run is performed to isolate the causal path. We introduce a formal system, wherein the corrected version of a faulty program is assumed so that the concept of ideal failure inducing chain *(FIC) can be defined by comparing the failing run and the run on the corrected program using the same input. Properties of such chains are studied. A product of the formal system is a metric that serves in the objective evaluation of the proposed technique. We identify a key enabling technique called execution indexing, whose goal is to establish a mapping between equivalent points in two different executions so that comparison can be meaningfully performed. We show that a control structure based indexing scheme, when integrated into the formal system, demonstrates very nice properties that can be exploited to develop an effective and efficient debugging algorithm. The evaluation shows that the metric lives up to its promise, computing desired FICs, and the proposed approach is able to compute high quality FICs. The results of our technique significantly supercede the state of the art.*

1 Introduction

Software development is mainly a human activity and humans inevitably make mistakes, resulting in bugs (faults) in the software. Software bugs are the main source of various computer-related problems. Companies lose money as software crashes. End users constantly suffer from malware due to software vulnerabilities. Handling the large volume of bug reports can easily drain a developer's energy. Identifying and removing bugs from software is a long-standing open challenge. Recently, significant progress has been made in model checking and static code analyses that are capable of capturing bugs at compile time. However, these techniques face problems such as false positives and scalability. Furthermore, they are mostly designed for specific kinds of faults and thus lack of generality. As a result, software bugs often survive the compile time sanitization and lead to runtime failures. *The process of understanding and fixing the fault that causes a failure is called debugging* in this paper. The common practice of debugging is that upon observing a failure, such as a segment fault or a wrong output, the programmer sets breakpoints, restarts the execution, changes values at breakpoints, and inspects the result of the perturbation. Such a practice is tedious and sometimes painful for developers.

Long-lasting efforts have been devoted to the automation of debugging. Fault localization focuses on automatically computing the faulty statement through techniques such as statistical analysis [1, 2, 3, 4] that aggregates information from multiple runs. The outcome of most of these techniques is a ranked list of statements that are candidates for the faulty statement. The onus is on the programmer to understand the failure and decide which listed candidate is faulty. However, due to the lack of information about how a fault leads to a failure, such an inspection is very labor-intensive. Furthermore, fault localization hardly handles cases in which the root cause lies in the high level requirements instead of the code.

Dynamic slicing [5, 6] tries to explain a failure by capturing all the statement executions that a failure is data/control dependent on. However, it is often the case that the failure not only depends on the faulty states induced by the fault but also a large number of benign states, resulting in very fat slices that contain a large volume of useless information. Previously, various techniques have been proposed to make slices thinner [7, 8]. However, their success is still limited due to the following factors. First, most slicing-based techniques focus on a single run – the faulty run – and have not taken advantage of other runs. Deciding if a statement execution is faulty based on only the failing run is very challenging. Existing proposals of considering multiple runs, such as dicing [9], hardly work for realistic programs as they do not handle cases in which the execution of the faulty statement does not necessarily lead to a failure. Second, to prune slices, these techniques often assume symptoms other than an observable failure, such as correct outputs in a failing run [8]. These symptoms may not be generally available.

Recently, Zeller et al. proposed the idea of isolating the state transitions that are critical for a failure by *comparing the failing execution with a similar but correct execution* [10, 11]. In the real world, developers have a "correct" oracle execution in mind. During debugging, they keep comparing the state in the failing execution to the oracle to identify faulty variables. The technique tries to mimic such a procedure by comparing the state in the faulty run with that in a similar but correct run, which serves as the oracle. Ideally, this approach can address many of the issues of fault localization and slicing. For example, it aims to produce a chain of state transitions that explain the induction of the failure, which is much more informative than a ranked list and not limited to code errors. It excludes benign transitions by looking at a reference execution, resulting in thinner chains, when compared with slicing.

Unfortunately, the state of the art in [10, 11] has not fully exploited the potential of the idea for the following reasons. *First*, while the idea is intuitive, the process is not formalized. Important properties that are critical for effectiveness and efficacy have not been studied. A metric is missing for objectively deciding the quality of the reported transition chains. *Second*, state comparisons are supposed to be performed at corresponding points in the two respective executions. Due to the differences between the two executions, construction of such correspondence is challenging. In [10, 11], it was carried out in an ad-hoc way such that points that are not semantically compatible may be selected for comparison. As a result, the computed chain is often hardly relevant to the failure. *Third*, our experience shows that the reported state transitions that are supposed to explain failure causality often fail to do so because they are caused by the inherent semantic differences between the two different executions.

In this paper, we propose an automated debugging technique that computes the causal path of a failure, which is a subsequence of the failing run that explains the failure. Given a failing run, the technique searches for a dynamic patch. If a patch can be found, which is true for most cases, the technique aligns the failing run and the patched run by establishing a mapping between instruction instances in the two runs. The states of aligned pairs are compared to identify faulty variables and causality testing is performed to identify subsets of faulty variables that are essential. The sequence of essential faulty states explains the failure. The contributions are highlighted as follows.

- We formally define the concept of *failure inducing chain* (FIC), which is a subsequence of the failing execution that gives rise to faulty states resulting in the failure. An ideal FIC is the subsequence generated by comparing the failing execution and the execution of the *corrected* program with the same input. Such ideal FICs serve as the ultimate goal and thus the metric for FIC computation. Interesting properties are identified for ideal FICs.

- We find that one of the most important enabling techniques for FIC computation is *execution indexing* (EI), whose goal is to identify aligned points in the two executions so that state comparison is meaningful. Comparisons performed at misaligned points introduce substantial noise because the semantics at those points are inherently different. We propose to use an indexing scheme based on program control structure, named *structural execution indexing* (SEI), in FIC computation. We prove that with SEI the formal model manifests certain properties that are key to effectiveness and efficiency.

- We propose and implement the idea of first patching a failing execution and using the patched execution as the reference to isolate the failure's causal path.

- We propose a demand-driven algorithm, which exploits FIC properties and delivers efficiency.

- We implement a prototype. We evaluate the effectiveness of both the metric and the proposed technique. The results show that the metric lives up to its promise by producing desired FICs. The proposed technique delivers high quality FICs. When compared with the state of the art in [10, 11], our technique computes much higher quality chains, largely due to the introduction of structural indexing and using a patched execution instead of a different execution as the reference.

2 A Formal Model

In debugging, understanding the causal path from a fault to a failure is the most critical task. We call such a path the *failure inducing chain* (FIC).

Despite their importance, FICs have not been formally defined and systematically studied. Approximations exist [12, 11, 10]. Unfortunately, the essence of these techniques is subtle and objective evaluation can hardly be achieved due to the absence of formalization. In this section, a formal model is introduced, in which the concept of an ideal FIC is proposed, assuming the presence of the corrected version of a faulty program. While this is not a realistic assumption for debugging, ideal FICs do serve as a metric to evaluate our solution, which is called an *approximate* solution. Essential properties of ideal FICs will be studied. In Section 4, the approximate solution is presented as a practical relaxation of the formal model.

As an FIC is essentially part of the failing execution, its formulation requires that we first define program execution. This paper assumes sequential programs.

Definition 1. *A program execution is a sequence of tuples in the form of* $< i,\ d = op(o1, o2, ..., on),\ v >$, *in which* i *is a unique identifier for the executed instruction,* d *is the destination,* $o1$, $o2$, ..., on *are the* n *source operands, and* v *is the value of the definition.*

A program execution is a sequence of executed instructions. Destination and source operands are variables. The identifier i uniquely represents an instruction execution instance. A simple form of i is s_j with s being the source code location and j being the execution instance, e.g., 4_2 means the second execution instance of statement 4. Intuitively, the FIC of a failing execution is a subsequence of the execution that explains the causality between faulty states from the root cause to the failure. Fig. 1 (a) shows a faulty program that has a faulty statement `x=1`, whereas the corrected version has `x=0`. The failure is manifested as the output being *True* at statement 7. Consider the failing execution E in Fig. 1 (b). The subsequence of 1_1, 3_1, 4_1 and 7_1 constitutes an FIC. Informally, an FIC should have the following properties: *(1) statement executions in an FIC propagate faulty state;* and *(2) they are essential to the manifestation of the failure, i.e., perturbing these executions may fail to produce the failure.* Note that not all variables that are directly or transitively computed from the faulty statement execution contain a faulty value. In Fig. 1 (a), if statement 2 changes to `y=(x>=0)`, y contains the correct value despite it uses a faulty variable x. Therefore, a definition based on def-use relations of variables is insufficient; it must be based on faulty state. The second property is also essential since not all faulty state necessarily contributes to the failure. For instance, inserting a statement `z=x<<1` in between statements 6 and 7 in Fig. 1 (a) introduces a faulty value into variable z. However, z has nothing to do with the failure and thus should not be part of the FIC. The aforementioned concept and properties are intuitive but vaguely defined. For example, whether a state is faulty is often decided by the programmer. A computable definition demands the introduction of a second execution that serves as a reference. Next, we introduce the formal definition of program state and faulty program state.

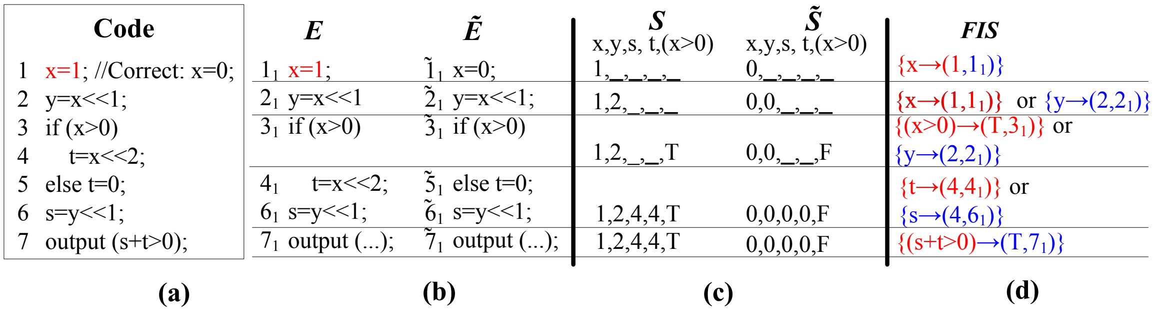
Definition 2. *The program state at an execution point* i*, denoted as* $\mathcal{S}(i)$*, is a function with the signature of* $Var \longrightarrow Val \times Def$*, which represents a mapping from variables to their values and the definition points of the values.*

A simpler definition of program state that maps variables to only their values is not adequate for our purposes. Our technique relies on comparing states in two executions, one being faulty and the other being benign. It is possible that a variable is defined by a statement in one execution to a value and by a completely different statement in the second execution to the same value. In such a case, the variable is decided to be benign if only values are compared, which is not desirable. In practice, a variable might be reset to zero by different statements in different executions, giving rise to the aforementioned case. Examples of program state are shown in Fig. 1 (c)[1]. The variables are presented in the first row and their values at execution steps are listed. Symbol '_' represents *undefined* and 'T' and 'F' denote *True* and *False*, respectively. In this paper, predicates are formulated as variables, such as the predicate $(x > 0)$ in Fig. 1 (c), in order to identify faulty control flow. To precisely determine faulty state, the corrected version of the faulty program is assumed[2]. Note that this corrected version does not exist in practice during debugging, which makes the definition to be an ideal one. Nonetheless, we need such an ideal definition to evaluate the effectiveness of the proposed technique. And for evaluation purposes, it is often the case that both the faulty program and its patched version are available.

Intuitively, a state in the failing execution that differs from the benign execution is faulty. Unfortunately, effective comparison of two executions is a non-trivial challenge. If the comparison is not carefully designed, a state in the beginning of one execution could be subject

[1] For readability, definition points are omitted.

[2] In this paper, we assume a program has only one fault.



```
Code
1  x=1; //Correct: x=0;
2  y=x<<1;
3  if (x>0)
4      t=x<<2;
5  else t=0;
6  s=y<<1;
7  output (s+t>0);
```

(a)

E	*Ẽ*	*S* x,y,s, t,(x>0)	*S̃* x,y,s, t,(x>0)	*FIS*
1_1 x=1;	$\tilde{1}_1$ x=0;	1,_,_,_,_	0,_,_,_,_	{x→(1,1_1)}
2_1 y=x<<1	$\tilde{2}_1$ y=x<<1;	1,2,_,_,_	0,0,_,_,_	{x→(1,1_1)} or {y→(2,2_1)}
3_1 if (x>0)	$\tilde{3}_1$ if (x>0)	1,2,_,_,T	0,0,_,_,F	{(x>0)→(T,3_1)} or {y→(2,2_1)}
4_1 t=x<<2; 6_1 s=y<<1;	$\tilde{5}_1$ else t=0; $\tilde{6}_1$ s=y<<1;	1,2,4,4,T	0,0,0,0,F	{t→(4,4_1)} or {s→(4,6_1)}
7_1 output (...);	$\tilde{7}_1$ output (...);	1,2,4,4,T	0,0,0,0,F	{(s+t>0)→(T,7_1)}

(b) (c) (d)

Figure 1. An example to illustrate our formal model.

to comparison with a state at the end of the other execution. In other words, the correspondence between steps in the two executions needs to be established. This challenge is formulated as *execution indexing* (EI) in our prior work [13].

Definition 3. *An indexing function regarding two executions E and $\widetilde{E}$ of the same program, denoted as $idx : E \longrightarrow \widetilde{E}$, produces a mapping between the two executions. Two execution points in the two respective executions are aligned iff the indexing function maps one to the other. An indexing function is valid if it produces unique indices for different points in E.*

A statement execution $i \in E$ may have $idx(i) = \perp$ if it does not have the aligned execution point in $\widetilde{E}$ due to the difference between the two executions. One may notice that the definition of indexing function is with respect to the same program but we have two programs, the faulty program and its corrected version, in our formalization. To accommodate this case, we assume the faulty statement is a mutant of the correct statement so that they are treated as the same statement by the indexing function. Practical relaxations to accommodate missing statements or additional statements that significantly distinguish the faulty program from the corrected version will be discussed in Section 4.

The horizontal lines in Fig. 1 illustrate the indexing function for the two executions. More particularly, the line below 2_1 and $\widetilde{2}_1$ indicates $idx(2_1) = \widetilde{2}_1$. The lack of a line below 4_1 means $idx(4_1) = \perp$.

Facilitated by execution indexing, faulty state can be clearly defined.

Definition 4. *At an execution point $i \in E$, its state $\mathcal{S}(i)$ is faulty iff (1). $idx(i) \neq \perp$ and (2). there exists a var s.t. $\mathcal{S}(i)(var) \neq \widetilde{\mathcal{S}}(idx(i))(var)$ with $\widetilde{\mathcal{S}}$ representing the states in the ideal execution.*

The variable *var* is called a faulty variable. The definition point of the value of *var* is called a faulty definition. Note that the equivalence comparison in condition (2) requires both the values to be identical and their definition points to align with each other. Consider the example in Fig. 1. State at 1_1 is faulty with x being the faulty variable and 1_1 being the faulty definition.

As mentioned earlier, not all the variables (even faulty variables) at a particular execution point contribute to the failure. In order to identify relevant faulty variables, the *failure inducing set* (FIS) is defined regarding a faulty state.

Definition 5. *Let $\mathcal{S}(i)$ be a faulty state and j be i's closest successor with $idx(j) \neq \perp$. A failure inducing set at i, denoted as $FIS(i)$, is a minimal subset of $\mathcal{S}(i) - \widetilde{\mathcal{S}}(idx(i))$ that has to be retained in order to induce $FIS(j)$, or induce the failure if j happens after the failure, when the rest of $\mathcal{S}(i)$ can be overwritten by the correct state $\widetilde{\mathcal{S}}(idx(i))$.*

Intuitively, an FIS is a minimal faulty state subset that induces the next FIS. We say a state subset is induced if the variable mappings defined in the subset happen. Fig. 1 (d) shows examples of FISs. Execution E is faulty and $\widetilde{E}$ is the corresponding ideal execution. $FIS(7_1) = \{(s + t > 0) \rightarrow (T, 7_1)\}$ because the faulty value of the predicate $(s + t > 0)$ has to be retained to expose the failure while the remaining variables can be overwritten with the correct values. Furthermore, there are two FISs at 6_1 as shown in different colors in column (d). Copying the correct values, i.e., values from $\widetilde{\mathcal{S}}$, to the complement set of either FIS does not mask the failure. For example, copying the correct values from $\widetilde{\mathcal{S}}(6_1)$ to all variables except s results in a state '0,0,4,0,F', which still causes 7_1 to print the wrong value. This justifies $\{s \rightarrow (4, 6_1)\}$ being a valid $FIS(6_1)$. These two FISs are also minimal.

It is worth noting that Definition 5 dictates local causality, i.e., causality between consecutive aligned points in the failing execution. In comparison, Zeller's cause effect chains demand global causality, i.e., his causal state transition at a step needs to induce the final failure. We found that global causality is undesirable as the resulting chains may be problematic. Consider the example in Fig. 1 (d), if global causality is enforced, $FIS(3_1)$ and $FIS(6_1)$ may be computed as $\{y \rightarrow (2, 2_1)\}$ and $\{t \rightarrow (4, 4_1)\}$, respectively, because both lead to the final failure. However, $FIS(3_1)$ does not result in $FIS(6_1)$ so that the chain does not explain the flow of faulty state.

Property 1. *The FIS for a given execution point may not be unique.*

The property has been demonstrated by Fig. 1 (d). The different FISs at each execution step are displayed in different colors. Multiple FISs at an execution step represent the multiple independent ways of leading to the failure.

Definition 6. *Given a sequence of FISs for the failing execution, the sequence of definitions that are captured in these FISs constitute an FIC.*

In Fig. 1, the two FIS sequences are rendered in different colors. $FIS(1_1)$ has two colors because it belongs to both sequences. The same is true for $FIS(7_1)$. The definition points in the FISs in the red chain constitute the FIC $1_1 \rightarrow 3_1 \rightarrow 4_1 \rightarrow 7_1$. The blue chain leads to another FIC $1_1 \rightarrow 2_1 \rightarrow 6_1 \rightarrow 7_1$. Both explain the failure. Note that although comparisons occur only at aligned points, the resulting FIC may contain execution points that are not aligned, e.g., 4_1 is present in one of the FICs although it does not align with any point in the benign run.

A metric to evaluate an approximate FIC computation proposal is to compare computed FICs to the corresponding ideal chains. Extra care has to be taken because of the non-uniqueness of FICs. One expensive solution is to compute all ideal FICs. Our solution is to always select the first FIS returned by the deterministic FIS computation procedure at each step.

3 Indexing in FIC Computation

The computation of an FIC hinges on the indexing function. A naïve indexing scheme such as mapping the ith instance of statement s, denoted as s_i, in one execution to the ith instance of the same statement in another execution fails to provide meaningful alignment in many cases [13].

```
1  if (P1)       | 1  if (P1)
2      F();      | 2      F();
3  if (P2)       | 3  s1;
4      F();      | 4  s2;
     (a)         |      (b)
```

Figure 2. Cases that fail naïve EI schemes.

Consider the example in Fig. 2(a). Assume in one run, method `F()` is called at line 2, and in the other run, it is called at line 4. It is often undesirable to create the mapping between the two calls despite that they are both the first instance. For a similar reason, a calling context based indexing approach fails to work [13].

Another simple indexing method based on a wall-clock does not work well either. Consider the code in Fig. 2 (b). Assume a very heavy method `F()` is called in one run but not in the other run. A time based indexing will very likely map `s1` and `s2` in the second run to some points inside the method call of `F()`. As a result, FISs computed by comparing these points are often not useful.

Although EI is in general an undecidable problem, in our prior work [13], we proposed an indexing function based on program control structure that provides good approximation. We called it *structural execution indexing* (SEI). The intuition of SEI is that *two points in two respective executions align if and only if they are instances of the same statement and their immediate enclosing control constructs align, which in turn requires the transitive enclosing constructs align up to the highest level – the two method bodies of the main function.* More formally, all possible executions of a program can be described by a language called an execution description language (EDL) based on structure. An execution is a string of the language.

Code	1 s_1;
	2 s_2;
	3 s_3;
	4 s_4;
		1 if (...)
		2 s_1;
		3 else
		4 s_2;
			1 while (...) {
			2 s_1;
			3 }
			4 s_2;
				1 void A() {
				2 B();
				3 }
				4 void B() {
				5 s_1;
				6 }

EDL	$S \longrightarrow \bar{1}\ \bar{2}\ \bar{3}\ \bar{4}$	$S \longrightarrow \bar{1}\ R1$ $R1 \longrightarrow \bar{2} \mid \bar{4}$	$S \longrightarrow \bar{1}\ R1\ \bar{4}$ $R1 \longrightarrow \bar{2}\ \bar{1}\ R1 \mid \epsilon$	$S \longrightarrow \bar{2}\ RB$ $RB \longrightarrow \bar{5}$
Str.	1 2 3 4	1 2 1 4	1 2 1 4 1 2 1 2 1 4	2 5

Table 1. EDLs for simple constructs.

Table 1 presents the EDLs for a list of basic programming language constructs. The second column shows sequential code without nesting, whose execution is described by a grammar rule that lists all the statements. Note that a terminal symbol s is denoted as $\bar{s}$ in the EDL grammar rules for readability. In the third column, the `if-else` construct introduces a level of nesting and thus the EDL has two rules, one expressing the top level structure that contains statement 1 and the intermediate symbol $R1$ representing the substructure led by 1. The two alternative rules of $R1$ denote the substructure of the construct. In the fourth column, the self recursion in the second grammar rule for the `while` loop expresses the indefinite iterations of the loop. Intuitively, an EDL describes all possible executions and the alphabet of the EDL contains all the statement ids in the program. Grammar construction is based on program control structure. In particular, statement instances that are dynamically control dependent on the same predicate instance or function call site should be parsed by the same grammar rule. Consider the rules for the `while` construct in the fourth column of Table 1. Statements 1 and 4 have the same dependence and they are listed on the right hand side of the first rule; the body of rule $R1$ lists the statements that are dependent on statement 1. Note that statement 1 is control dependent on itself as the execution of a loop iteration is decided by its previous iteration.

The EDL for the program in Fig. 1 is shown as follows.

$$\begin{aligned} S &\longrightarrow \bar{1}\ \bar{2}\ \bar{3}\ R3\ \bar{6}\ \bar{7} \\ R3 &\longrightarrow \bar{4} \mid \bar{5} \end{aligned}$$

The formal definition and more complicated examples of EDLs can be found in [13].

At runtime, an execution can be parsed by an EDL parser to a derivation tree. The structural indexing function that maps one point in E to its aligned point in $\widetilde{E}$ can be defined based on the derivation tree.

Definition 7. (Structural Indexing) *Given an execution E and a point i in E, let p be the path leading from the root in the derivation tree of E to the leaf node representing i, $idx(i)$ is the point in $\widetilde{E}$ that shares the same path p in the derivation tree of $\widetilde{E}$.*

Informally, the path from the root of the derivation tree to a leaf node represents the dynamic nesting structure of the statement instance. Therefore, SEI associates points in two executions that have the same nesting structure and execute the same statement. SEI was shown to be a valid indexing function in [13] according to Definition 3.

The derivation trees for the two executions in Fig. 1 (b) are shown in Fig. 3. It is easy to tell that all statements in the left tree except 4 share the same path with some statement in the right tree, which explains the horizontal lines in Fig 1. Real executions with loops, recursion, and non-structural control flow give rise to complicated derivation trees. Handling these cases can be found in [13]. A cost-effective implementation was also presented with the average overhead of 42%.

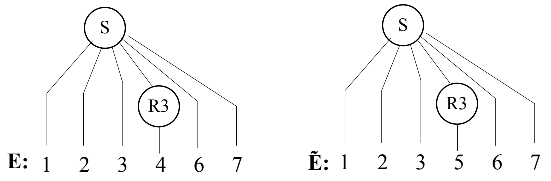


Figure 3. Structural indexing for executions in Fig. 1

One might notice that the constructive definitions in Section 2 regarding FIS and FIC are independent of the underlying indexing function as long as it is valid. In other words, even with a bad indexing function that significantly misaligns the two executions, FISs and thus the FIC can still be computed. However, the computed FIC will instead comprise the inherent semantic differences at these misaligned steps rather than the relevant faulty states. In other words, the quality of the FIC is heavily decided by the quality of the indexing function. We find that FIC computation possesses nice properties if the execution indexing function satisfies certain conditions.

Definition 8. *An indexing function is order-preserving, iff for two points i and j in E, with $idx(i) \neq \perp$, $idx(j) \neq \perp$ and i happening before j, then $idx(i)$ must happen before $idx(j)$.*

Property 2. *If the indexing function is order preserving, given any two execution points i and j in the failing execution, i happens before j, if $FIS(i) \equiv FIS(j)$, for any point k in between i and j with $idx(k) \neq \perp$, $FIS(k) \equiv FIS(i)$.*

The property states the stability of FISs. It says if two steps in the failing execution have identical FISs, all the intermediate steps have the same FIS. It requires the indexing function to be order preserving. Consider a counter-example in Fig. 4. Aligned points in the two executions are connected. Assume at point h in E that precedes i, a variable x is defined. Since the EI function is not order preserving, it may happen that the equivalent assignment to x occurs after $idx(j)$ in $\widetilde{E}$. Since x has different values at i and $idx(i)$, x is a faulty variable at i. The same is true at j. Assume x belongs to both $FIS(i)$ and $FIS(j)$. Let k be a point in between i and j, whose alignment $idx(k)$ happens after the assignment to x. Variable x has the same value at k and $idx(k)$ and the definition points are aligned too, precluding the presence of x in $FIS(k)$. In other words, $FIS(k)$ cannot be identical to $FIS(i)/FIS(j)$.

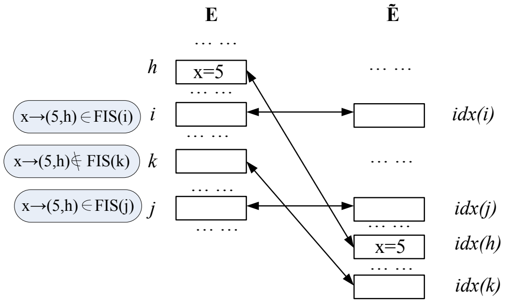


Figure 4. A counter-example for Property 2, assuming $FIS(i) \equiv FIS(j)$ and the EI function is not order preserving. Steps are represented by rectangles. Aligned steps are connected by double-arrows. Shaded round boxes list the assumptions.

Proof. Recall that variable definition points are part of an FIS. $FIS(i) \equiv FIS(j)$ implies all the definitions in $FIS(i)$ occur before i and are not killed in between i and j, otherwise, they cannot be present in $FIS(j)$. At any point k in between i and j with $idx(k) \neq \perp$, the same set of variables as those in $FIS(i)$ are able to induce $FIS(j)$ as they are not re-defined. Furthermore, this set of variables must be faulty at k too. Otherwise, there must be a variable $x \in FIS(i)$ that becomes benign at k. That is to say, in the ideal execution, x is re-defined at an point in between $idx(i)$ and $idx(k)$ to the same value and the re-definition point is aligned to the definition of x in the faulty run, which must happen before i. This is contradictory to the condition that the indexing function is order preserving. Finally, it is easy to see that $FIS(i)$ is also a minimal set at k to induce $FIS(j)$. Therefore, $FIS(k) \equiv FIS(i) \equiv FIS(j)$. □

This property will be shown in Section 4 to be very important for the efficacy of FIC computation.

Theorem 1. *Structural execution indexing is order preserving.*

Proof. Let $idx : E \rightarrow \widetilde{E}$ establish correspondence between E and $\widetilde{E}$ using SEI. Let i, j be points in E such that i precedes j and $idx(i) \neq \perp$, $idx(i) \neq \perp$. Under SEI, both i and j are paths in the derivation tree. All structural indices share a common root representing the entry point of the program, so i and j have at least one node in common. Consider the *last* common node of i and j in the derivation tree. Because the individual production rules of SEI express temporal order within a structural region, the paths of i and j must separate at this node such that the continuing path of i precedes the continuing path of j in the derivation tree. However, these same paths identify $idx(i)$ and $idx(j)$ in $\widetilde{E}$, so the continuing path of $idx(i)$ precedes the continuing path of $idx(j)$. Thus, again because the production rules capture local temporal order, $idx(i)$ must precede $idx(j)$. □

Not all valid indexing functions are order-preserving. The indexing scheme that relies on the id of a statement and its instance count is not order-preserving.

Theorem 2. *Assume FIS computation terminates at i, meaning $FIS(i) \equiv \phi$ and $FIS(j) \neq \phi$ with i and j being consecutive aligned points and i preceding j. If the state at i is benign, $FIS(j)$ must be a singleton that contains an instance of the faulty statement r, which is herein called the root cause.*

Recall the definition of FIS implies backward computation. The theorem says that if FIS computation terminates at a benign state in the failing execution, it captures the root cause.

Proof. Let's first prove $FIS(j)$ is a singleton. Assume $FIS(j)$ contains two faulty variables x and y that are defined at k and l, respectively. Since i has a benign state, k and l must occur after i and before/at j. That is to say, there must be at least one execution point in between i and j. Let's assume it is k. The point k must be aligned because the state at i is benign so that the next executed statement must be the same for both E and $\widetilde{E}$. That contradicts the condition that i is the immediate aligned predecessor of j. As there must not be any other execution points in between i and j and $FIS(j)$ has a faulty variable, this variable must be produced at j from benign values. Therefore, j has to be an instance of the faulty statement. □

Our experience shows that FIS computation mostly terminates at a benign state if the fault is a single statement fault[3] so that the root cause is caught. The concept of root cause is not as simple as one might think. Multiple instances of a faulty statement may contribute to a failure. For instance in the following example (a), both instances of the faulty statement contribute to the final faulty output. In such a case, the first instance is considered as the root cause and captured in $FIS(2_1)$.

```
1  for (i=0 to 1; i++)
2     sum+=1;//should be sum+=2
3  output(sum);
```

(a)

```
1  for (i=0 to 1; i++)
2     A[i]=0;//should be A[i]=1
3  output(A[1]);
```

(b)

[3] More complex faults will be discussed in Section 4.

It is wrong to assume the first execution instance of the faulty statement must be the root cause. Consider the example in (b). The failure is that value 0 is printed while value 1 is expected. We can see that the faulty statement gets executed twice. The second instance is the root cause instead of the first one. According to Definition 5, FIS computation terminates at the second instance of statement 2, which means even though 2_2's aligned predecessor 1_2 does not have a benign state, i.e., `A[0]` has a faulty value at 1_2, $FIS(2_2)$ still captures the root cause. That is to say, the technique often goes beyond what is stated in Theorem 2. Unfortunately, it is in general not provable that the FIS computation implied in Definition 5 always captures the root cause. An example can be constructed to show that FIS computation can terminate at a faulty state and the last computed FIS does not capture the root cause. In practice, we have not encountered such a case.

In general, one can observe that FIC computation facilitated by SEI produces a subsequence that starts with the root cause, ends with the failure, produces faulty values, and has causality between individual steps. Such a chain is exactly what a programmer expects in order to understand the failure.

Comparison with A Slicing-based FIC Definition. In our prior work [12], we informally defined an FIC as the dependence chop in between the root cause and the failure, i.e., the set of executed instructions that are dependent on the root cause and also depended on by the failure. A few limitations of the slicing-based definition motivate us for the new formalization. First, instruction executions that are dependent on the root cause and also depended on by the failure are often benign, meaning they do not produce faulty state and thus do not contribute to understanding the failure. A simple example is presented as follows

```
1  x=1; //should be x=0
2  y=x+1;
3  if (x>=0)
4      output(y);
```

The root cause is at 1 and the failure is at 4, since both 2 and 3 are dependent on 1 and depended on by 4. The chop contains all the statement executions. However, as we can see, the chain 1 → 2 → 4 is the one that explains the failure. Second, a slicing-based FIC is often much fatter than an FIC defined in this work as it contains all possible chains from the root cause to the failure including the proposed chain. Our initial experience shows that studying one chain is usually enough to understand the failure. Moreover, as a programmer, being presented with a smaller chain is often preferable. However, this needs to be confirmed by a larger scale study.

Third, traditional slicing techniques based on data/control dependence tracking are often not sufficient to disclose a failure induction path due to the existence of *implicit dependence* [12], which is neither data nor control dependence and cannot be computed by tracing one execution. The computation of implicit dependence is often more expensive than data/control dependencies.

Fourth, a slicing-based FIC is computed by tracking instruction level dependencies instead of comparing with the ideal execution. As a result, in cases such as a method is called in the faulty run but not in the ideal run, the slicing-based FIC traces each relevant exercised dependence in the method call, whereas the comparison-based FIC identifies and presents relevant state differences at the method call boundary. The latter case is often more preferable because knowing that the method call as a whole is faulty is in many cases enough without knowing all the instruction level details in the call.

4 An Approximate Solution

The formal system presented earlier enables the computation of FICs. However, in reality, relaxations have to be made in order to develop a practical debugging algorithm.

4.1 Constructing A Reference Execution

The definition of FIC assumes the availability of the corrected program. While such an assumption is useful for evaluation, it is not realistic for debugging. Therefore, we need to construct a reference execution from the failing one. A simple idea is to select an execution similar to the failing execution that produces correct output. In Zeller's work [10], delta debugging [14] was used to systematically explore the input space to find an input that is close to the failure inducing input but is able to drive the faulty program to produce correct output. Unfortunately, it demands a test oracle that decides if an execution produces the desired output. More importantly, even if such an input can be found, the semantic difference determined by the two different inputs inevitably confuses FIS computation because a faulty (benign) variable regarding to the ideal execution may no longer be a faulty (benign) variable regarding to the selected execution.

We proposes to construct a reference execution from the failing execution by patching it. A failing run is patched if it produces the desired output for the failure inducing input. Note that patching an execution is completely different from patching a faulty program. In our prior work [15], we have found that a failure can often be patched by switching the branch outcome of a predicate instance. It has been shown that 9 out of the 12 real bugs collected in [15] can be patched by predicate switching. In this work, we have also conducted a larger scale experiment on all failing test cases for SIR [16] programs. Results are shown in Table 2. On average, 80% of the failing executions had critical predicates that could patch the executions for use as oracles.

Program	Execs w/o criticals	Total Executions	% w/o criticals
tcas	126	1536	8.20%
replace	982	3267	30.05%
print_tokens	113	484	23.35%
print_tokens2	560	2064	27.13%
schedule	130	799	16.27%
schedule2	61	316	19.30%

Table 2. Switched predicate presence.

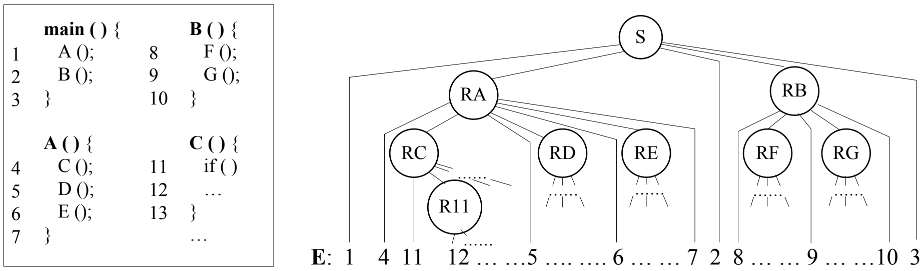
There are cases in which more than one predicate can be found such that switching any one of them produces the correct output. Our criterion is to select the first such predicate as it is often the closest one to the root cause. Predicate switching patches a failing run in two possible ways. In the first case, the predicate instance to be switched happens before the root cause such that predicate switching prevents the faulty statement from being executed. In this case, an FIC computed by comparing a failing run with its patched version often captures the root cause. In the second case, the predicate to be switched happens after the root cause, as a result of some faulty state. Switching the predicate re-directs the control flow back to the desired path. In such a case, the FIC often does not capture the root cause but rather part of the ideal FIC. However, we argue that capturing part of an ideal FIC is equally desirable as capturing the root cause. It was presented in a recent white paper [17] by *national research council* that the majority of software errors are requirement errors. Therefore, there usually does not exist a statement or a set of statements that constitute a root cause.

4.2 State Differencing and Minimizing

Recall that an FIS is the minimal faulty state subset that induces its successive FIS. The minimality requirement implies that an exponential number of possible subsets have to be tried. In [10], a state comparison algorithm was proposed. In the algorithm, the state of an execution at a particular point is represented by a memory graph [18] that reflects the variables, values, and indirect memory structures applicable to the execution at that point. As such, each node in the graph corresponds to a single scalar value or memory structure, and each edge in the graph reflects a memory reference relationship. By establishing edge and node correspondence between two memory graphs for an execution point i in E and its peer $idx(i)$ in $\widetilde{E}$ respectively, it is possible to express state difference as the graph difference. Standard delta debugging [14] is then applied to the graph difference to isolate the minimum set that produces the failure once applied. However, the algorithm does not extend into computing the FIS at an execution point, rather it only seeks to find the minimal state at a point which will induce execution failure in the end. As discussed with Definition 5, local causality is more restrictive. As such, only the very last state in the entire FIC is derived to minimally induce the failure itself. In our FIS computation algorithm, this relevant state is computed first, and all prior states within the FIC can then be derived "backwards" from their successors. Constructively, delta debugging is then applied such that each state minimally induces not merely the failure, but the precise failure inducing state of its successor in the FIC.

4.3 A Demand-Driven Algorithm

Recall that FIC computation with SEI support has the stability property (Property 2), which says that if the FISs computed at any two aligned points are the same, there is no need to compute FISs in the middle of these two points as they will be identical. Based on this property, a hierarchical algorithm can be designed to compute FISs in a demand-driven fashion. The idea is to carry out state comparison and causality testing top-down and in reverse order along the indexing tree of the failing execution until the right granularity is reached. The top-down traversal descends into and examines only as much of the indexing tree as is necessary to find the appropriate FISs, and the FISs are computed bottom up along this reverse order traversal via the leaves representing indices at each level. Thus, each FIS is computed to induce the successive one except the last FIS, which induces the observed failure.



```
   main ( ) {          B ( ) {
1    A ();        8      F ();
2    B ();        9      G ();
3  }              10   }

   A ( ) {             C ( ) {
4    C ();        11     if ( )
5    D ();        12       ...
6    E ();        13   }
7  }                   ...
```

Figure 5. An example for the algorithm.

Consider the example in Fig. 5. The code is on the left; an execution and its index tree are shown on the right. The index tree reflects the nested function calls. Assume all the execution points presented in the trace are aligned with some points in the other run, and suppose the failure is first observable immediately after point 3. Because point 3 is the closest point to the failure in both executions, the final FIS, which induces the failure directly, must be calculated there. Then, instead of performing state comparison at all these points, the hierarchical algorithm first performs comparison at points with the indexing depth of 1, i.e., points 1, 2 and 3, corresponding to the highest level of the execution. If the FISs at two consecutive steps are not identical, assuming $FIS(2) \neq FIS(3)$, the algorithm moves one step down and computes FISs for the points in between 2 and 3 with the depth of 2. They are 8, 9, and 10. The process repeats until the adjacent FISs are all identical or the finest granularity is reached. Observe, that if $FIS(9) \neq FIS(10)$, then $FIS(8)$ must be computed with respect to inducing $FIS(9)$, and upon returning one level in the traversal, $FIS(2)$ must be computed to properly induce $FIS(8)$. The algorithm requires a lesser number of state comparisons and causality tests compared to a linear algorithm that computes FISs backwards step by step. Since the computation of ideal FICs and that of approximate FICs only differ by using either an ideal execution or a patched execution as the reference, the same algorithm can be used to compute both ideal and approximate chains.

Pseudo-code of the algorithm is shown in Algorithm 1. Using the failing execution and the generated passing execution, CalculateFIC() generates the complete FIC for the failing execution. Observe that the FIC computation first descends to the temporally latest index common to both the failing and passing executions, as computed by FindInitialFailure() and stored in `currentNode` on line 1. The first FIS at this point induces the original failure, as stored in `target` on line 2. Then, in lines 4-7, for each par-

Algorithm 1 Hierarchical FIC computation.

Primitives:

- FindFailingChild()- Finds the last child of this node before the failure common to the failing and passing runs.
- CommonChildTerminals()- Finds the leaf children of this node common to the failing and passing runs.
- FIS()- Finds the FIS at the execution index represented by the given node, inducing either failure or the successive FIS.

```
CalculateFIC()
 1 currentNode ← FindInitialFailure()
 2 target ← FIS(currentNode) inducing failure
 3 defs ← {target}
 4 while target ≠ ∅ do
 5     currentNode ← Parent(currentNode)
 6     (sets,target) ← SetsInRegion(currentNode,target)
 7     defs ← defs ∪(∪_{fis∈sets})
 8 return TemporalSort(defs)

FindInitialFailure()
 1 currentNode ← executionRoot
 2 loop
 3     nextNode ← FindFailingChild(currentNode)
 4     if nextNode = ∅ then
 5         return currentNode
 6     currentNode ← nextNode

SetsInRegion(node, target)
 1 kids ← CommonChildTerminals(node)
 2 if kids = ∅ then
 3     return ({},target)
 4 (last,first) ← (LastIndex(kids),FirstIndex(kids))
 5 sets ← ∅
 6 if kids[last] precedes a shared nonterminal x then
 7     sets ← SetsInRegion(x, target)
 8 target ← FISTo(target, kids[last])
 9 sets ← sets ∪ target
10 stack ← [first]
11 while stack is not empty do
12     seek ← pop the last element of stack
13     mid ← midpoint of (seek,last) truncated
14     if seek = mid then
15         if ∃ internal node x between seek and mid then
16             (subsets,target) = SetsInRegion(x, target)
17             sets ← sets ∪ subsets
18         end ← mid
19         target ← FIS(kids[mid]) inducing the old target
20         sets← sets ∪ target
21     else if FIS(kids[mid]) = FIS(kids[last]) inducing target then
22         last ← mid
23         push seek on stack
24     else if FIS(kids[mid]) = FIS(kids[seek]) inducing target then
25         push mid on stack
26     else
27         push seek on stack
28         push mid on stack
29 return (sets,target)
```

ent node, stored in currentNode, back along the path from the failure up to the root of the indexing tree, we perform the previously mentioned demand driven approach with SetsInRegion() to collect the FISs relevant within each subtree of the indexing tree rooted at currentNode that has not yet been visited. Note that the bottom-up traversal in lines 4-7 is dictated by the fact that FISs are computed backwards, i.e., an FIS can not be properly computed if its successor is not properly computed. That makes the algorithm slightly more complicated than what was illustrated in the example. The computation for a subtree yields the FISs collected over that subtree along with the new target FIS that must be induced. The relevant definitions from the collected FISs are aggregated into defs on line 7. Once all FISs have been aggregated, when the next target to induce is empty, the definitions they contain are sorted by their temporal position within the failing execution and returned on line 8. This sequence of definitions comprises the complete FIC.

FindInitialFailure() uses a direct top-down search of the indexing tree starting at executionRoot, the common entry point of both programs. It uses FindFailingChild() in line 3 to search all of the children of an internal node, or the internal node following a leaf node, common to the failing and passing runs in order to find the last common leaf child before the failure becomes observable. Lines 4-5 determine that when there is no such child, the current node of the search is returned as the last common index before the failure.

The top-down search for most of the indexing tree is performed by the recursive procedure SetsInRegion(). This generates the FISs for the portion of the indexing tree rooted at node such that the last analyzed index of the tree induces the FIS target. On line 1, we extract the ordered list of all the leaf nodes common to both executions and store them in kids. If there are no such leaves, then there are no shared indices between the executions at this subtree, so we return after no work on line 3. Note that in our grammar, a subtree is led by a leaf, which is usually a predicate. If the last child is an internal node, we must recurse and receive a new target FIS in lines 6-7. Lines 10-28 perform incremental, reverse order binary searches of the children, with the invariant that the last known and aggregated FIS is stored in target. The variable stack maintains a stack that holds approximate points in the list of children where changes in FIS may occur. The last element on the stack is the index of the next child that should be compared in the binary search against the target FIS, which pertains to the last child that was traversed. As elements are added to and popped off stack, the changes in FISs are refined by the search in lines 21-28. When two neighboring child nodes have different FISs, the procedure descends and aggregates any new FISs in lines 14-20, traversing the last node on the stack and receiving a new target FIS. Finally, the aggregated FISs for the subtree at node and the FIS applicable to the first in-order element of the subtree are returned at line 29.

4.4 Indexing in the Presence of Various Types of Faults

One of the major contributions of our work is to propose the concept of ideal FIC that serves as an objective evaluation metric. The ideal FIC computation requires two versions of the program: the faulty version and the patched version. In the formal system mentioned earlier, we assume the faulty statement is a mutant of the correct statement so that the indexing function can virtually treat them as the same statement. In practice, we encounter many cases in which faulty programs are much more complex transformations of their corrected versions. In some cases, a new method has to be inserted to the faulty program to fix a bug. To overcome this issue, we use diff to identify the static common parts of the two programs. Only the common parts are subject to execution indexing. Under such a relaxation, the indexing function is still order-preserving. Detailed discussion of this issue is beyond this paper.

5 Evaluation

A prototype was developed to analyze C programs using a combination of source to source transformation via CIL[19], Python, and the publicly available Python and GDB infrastructure developed in [11]. SEI and predicate switching are implemented using CIL. The demand-driven algorithm is implemented using Python. We used the programs in SIR [16] for our evaluation. For each program, SIR provides the correct version of the program, a set of mutants with faults injected, and a test suite. For the first four programs, we used all the available mutants, which are around 10. For programs replace and tcas, the first ten mutants were selected as there are too many mutants for in depth study. For each mutant, we selected the first four failing runs. Note that an injected fault may fail in multiple unit tests. Table 4 presents the results. As both the faulty program and its corrected version are available, we are able to compute the ideal FICs. In contrast, approximate FICs are those computed by comparing failing executions and their patched versions. For comparison, we also downloaded the system presented in [11] and applied it to the same set of failing runs. Recall that their system does not have indexing support and uses a similar execution as the reference. The results are presented in columns labeled with DD. In the table, FIC length refers to the number of elements in the FIC sequence. Coverage refers to the percentage of an ideal chain that is captured by the approximate/DD chain. Relevance refers to the percentage of an approximate/DD chain that is contained in the ideal chain. Edit distance is the standard Levenshtein distance metric between the ideal and approximate/DD chains, which represents the number of changes required to turn one chain into the other. To present a better view, each group of tests (failing runs) are divided into two sub-groups, *best* and *worst*, by the coverage values. Averages are computed for each subgroup. The following observations can be made from the results.

Observation One. The ideal FICs capture the root causes as their starting points, except for code omission errors. They explain causality and thus serve perfectly as a metric. Let's use one example to illustrate this observation. The `replace` program in the Siemens suite substitutes a pattern in an input string with an alternative user provided pattern. In the tenth mutant, the injected fault is that `esc()` returns `'\0'` instead of an escape pattern. This is propagated as seen in the ideal FIC in Fig. 6 until it is assigned to a position in a substitution string, truncating it and causing several characters to be omitted from the output. Observe that the root cause is perfectly captured at the start of the ideal FIC, and each step causes or collaborates with successive steps to create the failure.

Observation Two. The approximate FICs are consistently much better than those computed by the algorithm in [11] in terms of coverage and relevance. The first main reason is the introduction of execution indexing. Without indexing, execution comparison does not have a clear meaning, resulting in low quality chains. The low coverage and relevance of the DD algorithm are not contradictory to the results reported in [11], in which the observation was that the computed chains contain points *close* to root causes along dependence edges. Our metric is more stringent, as we have the ideal FICs. Furthermore, we studied four failing test cases for each mutant whereas they studied one in [11]. The second main reason is that a different execution is used as the reference in [11] so that the inherent semantic difference of the two executions significantly pollutes the resulting chains. We did another experiment, in which we used the same algorithm from [11] but took the patched execution as the reference rather than a different execution. The resulting chains have better coverage and relevance than the DD results in Table 4, but they are still not comparable to our chains. These results are labeled DD$_p$ in Table 4.

Code Snippet

```
esc(s, i):
1   result = '\0'; //was '@'
2   return result
addstr(c, outset, j, maxset):
3   outset[*j] = c;
makesub(arg, from, delim, sub):
4   escjunk = esc(arg, &i);
5   junk = addstr(escjunk, sub, &j, 100);
putsub(lin, s1, s2, sub):
6   while (sub[i] != '\0') {
7         fputc(lin[j],stdout);
main(argc, argv):
8   makeres = makesub(arg, 0, '\0', sub);*
9   putsub(lin, i, m, sub);*
```

Ideal FIC:

At 1, result is given '\0'
At 4, escjunk is given '\0'
At 5, arg c is given '\0'
At 3, sub/outset[59] is given '\0'
At 6, (sub[i] != '\0') is false
Thus the output ... differs from ...

Figure 6. An ideal FIC for replace

Observation Three. The proposed technique is able to compute high quality chains. For the *best* half of test cases, the average coverage and relevance range from 84%-100% and 68%-100%, respectively. The column *Roots* shows the number of cases in which the root causes are captured by our technique. The results imply that chains computed with our technique contain substantial and highly relevant information that (partially) explains the failures. An example can be found in our later case study. Two cases are dominant, one is that the approximate chains, i.e., chains computed by the proposed technique, are subsequences of ideal chains. It corresponds to the predicate to be switched happening after the root cause. There are also cases that the approximate chains are super-sequences of ideal chains. It corresponds to the predicate happening before the root cause. For some of the benchmarks, e.g. `tcas`, the technique was less effective for the *worst* half. The main reason is that the predicate selection strategy, i.e., selecting the first predicate patch, is less effective. We plan to further study these cases.

Runtime. The prototype is expensive. The average runtime over all test cases ranges from 7 - 9814 seconds with the average of 752 seconds. Without the demand-driven algorithm, a naïve linear algorithm that computes FISs backwards at each aligned step is orders of magnitude slower and does not terminate in 10 hours for many cases. Details can be found in Table 3. The main hurdle is that memory graph construction and causality testing, which are the most basic computation units, require expensive communication with GDB. We are working on replacing the GDB based component with an instrumentation based component. More specifically, the new implementation will be GDB free and memory graphs will be built incrementally on the fly by instrumentation rather than being constructed from scratch by querying GDB at each FIS computation.

Program	Min	Max
print_tokens	379	435
print_tokens2	74	6134
replace	8	9814
schedule	7	899
schedule2	19	4564
tcas	27	138

Table 3. Execution times in seconds.

5.1 Case Studies

grep. Version 2.5.1 of the `grep` utility has a flaw which causes `grep` to fail to find the strings that match the provided patterns if both -F and -w options are specified [15]. These options require that pattern be a string and a word respectively, but those cases are not mutually exclusive and should not preclude matching. This is a design bug.

The switched predicate disables the -w option, so the oracle execution behaves as if only the -F option were passed in. The resulting FIC and relevant code is reproduced in Fig. 7. Observe that the starting point relates to the switched predicate. First, the FIC reveals that the -w option is required for the error and that it requires word patterns by setting the `match_words` flag. The third FIC element says this flag is checked in `Fexecute`, which then returns -1 to `match_offset`, signalling that the the buffer is done scanning. Finally, at statement 5, `match_offset` is checked and the program ends.

From this case, we can see there is not a single statement to be blamed for the failure. Understanding the causal path of the failure becomes much more important. In `grep` 2.5.3, the bug is fixed by placing a multi-line patch in statement 2. The fact that the FIC provides an explanation and points the programmer to the places right before and after statement 2 clearly demonstrates the power of our technique.

Code Snippet

```
Fexecute(buf, size, match_size, exact):
1   if (match_words) {
2        /*verify the match is a word*/
3        return -1;
grepbuf (char const *beg, char const *lim):
4   match_offset = Fexecute(p, lim-p, &match_size, 0);
5   while (match_offset != -1) {
6        /*Print results & Keep scanning*/
main(argc, argv):
7   while (opt != -1) { ...
8        case ’F’: setmatcher (”fgrep”);
9        case ’w’: match_words = 1;
10  n += grepbuf (beg, lim);*
```

Approximate FIC

At 7, opt is given 1
At 9, match_words is given 1
At 1, (match_words) is true
At 3, Fexecute returns -1
At 4, match_offset is given -1
At 5, (match_offset != -1) is false
Thus the output ... differs from ...

Figure 7. Approximate FIC for grep 2.5.1

gzip. Version 1.3.9 of `gzip` has a design flaw wherein some files are not compressed when they should be. By default, `gzip` may refuse to compress files for various reasons. For example, it will reject files with certain filename extensions, such as `.gz`, in order to avoid recompressing files that have already been compressed. In general, this behavior may be overridden by using the -f option to force compression, however, in version 1.3.9, the forcing option erroneously still does not allow the compression of already compressed files.

Using our infrastructure, the switched predicate causes an empty string to be copied into the buffer used for detecting filename extensions of already compressed files. The resulting FIC and relevant code is reproduced in Fig. 8. The first several steps show the causal behavior for detecting filename extensions that denote already compressed files: first that the filename must be examined, then that it is converted to lowercase and compared against expected extensions. All of this finally yields that an extension must be detected by `get_suffix` at line 15 and, finally, that this ultimately causes the predicate on line 17 to irretrievably lead to failure within `make_ofname`. Because this is a design flaw, there is not simply one faulty predicate, but the code *examining* file extensions is shown to behave properly by the FIC, so the only appropriate place to fix the bug is in line 17. At the time of writing, a fix for the flaw is present only in the CVS repository for `gzip` source code. Indeed, it changes line 17 to additionally check if the force (-f) option is used.

Benchmark		# Tests	FIC Length Ideal /	Approx	Coverage	Relevance	Edit Distance	Roots	DD Length	DD Coverage	DD Relevance	DD_p Length	DD_p Coverage	DD_p Relevance
schedule	best	14	2.5	3.5	100.00%	78.21%	1.0	14	2.44	43.45%	46.30%	2.00	50.00%	50.00%
	worst	14	13.71	8.71	45.29%	48.59%	12.5	1	3.44	20.17%	30.56%	3.14	31.24%	33.33%
schedule2	best	14	4.64	6.36	98.21%	79.86%	0.38	13	3.50	25.00%	29.17%	4.60	23.94%	34.33%
	worst	14	11.21	7.21	30.73%	60.73%	9.0	0	4.67	15.53%	22.22%	3.80	19.82%	32.00%
print_tokens	best	12	4.42	4.42	100%	100%	0	12	2.00	46.30%	66.60%	2.00	80.00%	41.74%
	worst	12	4	10	97.22%	93.66%	6.17	11	6.40	22.28%	50.0%	2.00	41.73%	100.0%
print_tokens2	best	16	12.56	15.94	98.68%	72.59%	4.63	14	3.53	20.01%	36.56%	3.69	33.58%	91.28%
	worst	16	12.19	13.44	62.80%	69.34%	8.5	0	4.26	6.77%	26.22%	2.71	11.21%	62.86%
replace	best	19	12.0	15.95	93.36%	68.56%	5.74	9	5.36	15.45%	61.54%	2.05	23.21%	100.00%
	worst	19	102.26	8.73	33.43%	70.45%	95.52	0	5.21	14.98%	27.47%	2.17	12.08%	81.67%
tcas	best	20	6.8	7.0	84.64%	80.0%	1.6	12	3.00	20.00%	66.67%	2.00	29.76%	100.00%
	worst	20	11.4	7.0	27.35%	42.86%	8.4	0	3.00	20.00%	66.67%	2.00	18.23%	100.00%

Table 4. Evaluation.

Code Snippet

```
get_suffix(filename):
1  if (strlen(filename) <= MAX_SUFFIX)
2      strcpy(suffix_buffer, filename);
3  else
4      strcpy(suffix_buffer, "");
5  strlwr(suffix_buffer);
6  slen = strlen(suffix_buffer);
7  ...
8      if (slen > strlen(".gz") && /*ends in suffix .gz*/)
9          return ".gz";
10 ...
11 return NULL;

strlwr(s):
12 for (char *t = s; *t; t++)
13     *t = tolowercase(*t);
14 return s;

make_ofname():
15 suff = get_suffix(ofname);
16 ...
17 } else if (suff != NULL) {
18     /* Print out recompressing error and abort */
```

Approximate FIC

At 1, (strlen(filename) <= MAX_SUFFIX) is true
At 2, suffix_buffer is given “IN.gz”
At 5, arg s is given “IN.gz”
At 12, *t is true
At 13, s/suffix_buffer is given “iN.gz”
At 13, s/suffix_buffer is given “in.gz”
. . . (over length of filename)
At 6, slen is given 5
At 8, (slen > strlen(".gz")) is true
At 9, get_suffix returns “.gz”
At 15, suff is given “.gz”
At 17, (suff != NULL) is true
Thus the program aborts and the file was not compressed

Figure 8. Approximate FIC for gzip 1.3.9

6 Related Work

The earliest attempts towards algorithmic and partly automated debugging are [20] by Shapiro and [21] by Fritzson et al. The basic idea is to allow developers to provide a partial specification of the program by answering questions. The debugging algorithm, guided by the specification, gradually isolates the fault. While such a principle is still valid in general, the specific techniques are out of date as they do not allow side-effects or require too many user interactions.

Delta Debugging. The work that is most relevant to ours is the series of works by Zeller [14, 10, 11]. The project in [10] is the first one to propose to compare two similar executions using delta debugging [14, 22] to compute cause effect chains, which are a concept similar to FICs. Later in [11], the technique is further extended to link cause transitions to a faulty statement. Compared to these works, we make significant progress on the following: we introduce a formal model, propose an evaluation metric, identify the key enabling technique – execution indexing, identify a set of important properties, develop an effective algorithm, use a patched execution instead of a different execution to reduce noise caused by inherent semantic differences. As the origin of all these differences, our definition of failure inducing chain differs from Zeller’s definition in a number of aspects: our definition is tied with execution indexing that enables computability and clarity whereas a rigid definition of Zeller’s chain is absent; our definition dictates local causality where Zeller’s definition demands global causality, which loses some important properties; our definition is backward whereas Zeller’s is not directional, and we believe debugging is by its nature a backward problem.

Fault Localization. Fault localization computes fault candidates by looking at many executions including both passing and failing. Harrold et al. [23] compared the spectra of passing and failing runs and found that failing runs tend to have unusual coverage spectra. Value spectra were further used in [24] to improve the result. Jones et al. [25] ranked each statement according to its ratio of failing tests to correct tests and used this information to assist fault location. Renieris and Reiss [4] focused on the difference between the failing run and a *single* passing run with similar spectra as a means to narrow down the search space for faulty code. Liblit et al. [1] describe a sampling framework and present an approach to guess and eliminate predicates to isolate a deterministic bug. For isolating nondeterministic bugs, they use statistical regression techniques to identify predicates that are highly correlated with the program failure. SOBER [2] is a similar technique that makes use of predicate evaluation distributions instead of predicate occurrences. Daikon [26] is a technique that detects dynamic program invariants that can serve as partial specification for debugging purposes. It is later used in fault localization by observing program invariant violations in [27, 28]. Crisp [29] is a technique that helps developers in regression testing, allowing developers to selectively apply a set of code edits and then observe the correlation between code edits and regression failures. Compared to the proposed work, fault localization techniques are in general lack of or very limited in the capability of explaining failures. They produce a ranked candidate set, usually containing static statements. Reasoning about the candidates and the failure often falls onto the programmer.

Dynamic Slicing. Dynamic slicing was introduced as an aid to debugging [5, 6]. Compared to fault localization, slicing features the capability of capturing causality through program dependencies. Recent works such as [30, 31] have greatly improved the efficacy of dynamic slicing. Experience shows that dynamic slicing tends to produce very fat slices that contain all possible causality chains that lead to the failure, starting from program input. Although various techniques have been proposed to prune dynamic slices [7, 8] or aid in their navigation [32, 33], without using a reference execution to exclude benign chains, inspecting slices still requires non-trivial human effort. Dicing [9] is a technique that aggregates slices from multiple executions. However, the simple set manipulations in dicing undermine causality in slices and make the resulting dices hard to understand. Furthermore, it does not handle cases in which a faulty statement occurs in both the benign and faulty slices. In comparison, the work proposed here does not rely on program dependence, but rather semantic causality, which is more rigid with respect to debugging. The use of a reference execution effectively excludes benign state from computed chains.

Others. Recently, researchers have made significant progress on heap memory failure diagnosis using anomaly detection [34] and statistical analysis [35]. Heap failures can even provably be probabilistically prevented through memory randomization and duplication [36]. Compared to these works, the proposed work is more general. The proposed work is also related to execution selection and generation for debugging. In [37], Wang et al. proposed a path generation technique that generates a correct execution that is similar to the failing execution by switching predicate outcomes in the faulty run. The technique demands an oracle to decide if the execution produces benign output and relies upon constraint solving. Once such an execution is found, the switch points are returned as the bug report. We use patched executions for comparison whereas they used normal executions driven by inputs generated by a constraint solver. It would be interesting to see how to combine their technique with ours. On the other hand, their bug report is lacking causality information and not as helpful as ours.

7 Conclusions

We propose a highly effective technique to automatically compute the failure inducing chain (FIC) of a failure. A formal model of FICs is proposed that suggests FICs should ideally be computed by comparing the failing run with the run of the corrected program with the same input. A technique called execution indexing is proposed to align the two executions for precise comparison. For practical debugging, in which corrected versions are not available, we propose an approximate solution. The idea is to first patch the failing run by switching a dynamic predicate. The patched run is used as a reference to compare with the failing run. Ideal FICs are used as a metric to evaluate the approximate solution. Our experiment shows that ideal FICs precisely represent failure causal paths. The approximate solution is able to deliver chains of high quality.

References

[1] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, pp. 141–154, 2003.

[2] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff, "Sober: statistical model-based bug localization," in *ESEC/FSE-13*, pp. 286–295, 2005.

[3] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, pp. 273–282, 2005.

[4] M. Renieris and S. Reiss, “Fault localization with nearest neighbor queries,” in *ASE*, pp. 30–39, 2003.

[5] B. Korel and J. Laski, “Dynamic program slicing,” *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[6] H. Agrawal and J. R. Horgan, “Dynamic program slicing,” in *PLDI*, 1990.

[7] N. Gupta, H. He, X. Zhang, and R. Gupta, “Locating faulty code using failure-inducing chops,” in *ASE*, pp. 263–272, 2005.

[8] X. Zhang, N. Gupta, and R. Gupta, “Pruning dynamic slices with confidence,” *SIGPLAN Not.*, vol. 41, no. 6, pp. 169–180, 2006.

[9] T. Y. Chen and Y. Y. Cheung, “Dynamic program dicing,” in *ICSM*, pp. 378–385, 1993.

[10] A. Zeller, “Isolating cause-effect chains from computer programs,” in *FSE*, pp. 1–10, 2002.

[11] H. Cleve and A. Zeller, “Locating causes of program failures,” in *ICSE*, pp. 342–351, 2005.

[12] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, “Towards locating execution omission errors,” in *PLDI*, 2007.

[13] B. Xin, N. Sumner, and X. Zhang, “Efficient program execution indexing,” in *PLDI*, 2008.

[14] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[15] X. Zhang, R. Gupta, and N. Gupta, “Locating faults through automated predicate switching,” in *ICSE*, 2006.

[16] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.,” *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[17] D. Jackson, M. Thomas, and L. I. Millett, *Software for Dependable Systems:Sufficient Evidence?* Washington D.C.: Committee on Certifiably Dependable Software Systems, National Research Council, 2007.

[18] T. Zimmermann and A. Zeller, “Visualizing memory graphs,” in *Revised Lectures on Software Visualization, International Seminar*, pp. 191–204, 2002.

[19] G. Necula, S. McPeak, S. Rahul, and W. Weimer, “Cil: Intermediate language and tools for analysis and transformation of c programs,” in *CC ’02*, 2002.

[20] E. Y. Shapiro, *Algorithmic Program DeBugging.* Cambridge, MA, USA: MIT Press, 1983.

[21] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri, “Generalized algorithmic debugging and testing,” in *PLDI*, pp. 317–326, 1991.

[22] G. Misherghi and Z. Su, “HDD: Hierarchical delta debugging,” in *ICSE*, 2006.

[23] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, “An empirical investigation of the relationship between spectra differences and regression faults,” *Software Testing, Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

[24] T.-Y. Huang, P.-C. Chou, C.-H. Tsai, and H.-A. Chen, “Automated fault localization with statistically suspicious program states,” in *LCTES*, pp. 11–20, 2007.

[25] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *ICSE*, pp. 467–477, 2002.

[26] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *TSE*, vol. 27, no. 2, pp. 1–25, 2001.

[27] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *ICSE*, pp. 291–301, 2002.

[28] Y. Brun and M. D. Ernst, “Finding latent code errors via machine learning over program executions,” in *ICSE*, pp. 480–490, 2004.

[29] O. C. Chesley, X. Ren, B. G. Ryder, and F. Tip, “Crisp–a fault localization tool for java programs,” in *ICSE*, pp. 775–779, 2007.

[30] X. Zhang and R. Gupta, “Cost effective dynamic slicing,” in *PLDI*, pp. 94–106, 2004.

[31] T. Wang and A. Roychoudhury, “Using compressed bytecode traces for slicing java programs.,” in *ICSE*, 2004.

[32] T. Wang and A. Roychoudhury, “Hierarchical dynamic slicing,” in *ISSTA*, pp. 228–238, 2007.

[33] A. J. Ko and B. A. Myers, “Debugging reinvented: asking and answering why and why not questions about program behavior,” in *ICSE*, pp. 301–310, 2008.

[34] T. M. Chilimbi and V. Ganapathy, “Heapmd: identifying heap-based bugs using anomaly detection,” in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 219–228, 2006.

[35] G. Novark, E. D. Berger, and B. G. Zorn, “Exterminator: Automatically correcting memory errors with high probability,” in *PLDI*, 2007.

[36] E. D. Berger and B. G. Zorn, “Diehard: probabilistic memory safety for unsafe languages,” in *PLDI*, pp. 158–168, 2006.

[37] T. Wang and A. Roychoudhury, “Automated path generation for software fault localization,” in *ASE*, pp. 347–351, 2005.