PAPER 10

***Abstract*—We propose a novel fine-grained causal inference technique. Given two executions and some observed differences between them, the technique reasons about the causes of such differences. The technique does so by state replacement, i.e. replacing part of the program state at an earlier point to observe whether the target differences can be induced. It makes a number of key advances: it features a novel execution model that avoids undesirable entangling of the replaced state and the original state; it properly handles differences of omission by symmetrically analyzing both executions; it also leverages a recently developed slicing technique to limit the scope of causality testing while ensuring that no relevant state causes can be missed. The application of the technique on automated debugging shows that it substantially improves the precision and efficiency of causal inference compared to state of the art techniques.**

I. Introduction

Explaining *why* something happened is a subtle task; philosophers have debated the notion of causation for centuries [1]–[3]. One common thread among the myriad approaches is that they involve comparing a world in which that something happened to others in which it did not. Many software engineering techniques take similar approaches in explaining software behavior. For example, in probabilistic fault localization, a set of failing runs is contrasted with a set of passing runs [4], [5] to provide probabilistic insights into the cause of the failures. Compared to techniques that do not rely on comparison to explain software behavior, such as program slicing [6], these techniques are more precise as they use comparison to trim unnecessary information.

One classic fine-grained comparative technique for identifying causes when one execution (e.g., a buggy execution) differs from another (e.g. a similar correct execution) is Zeller's delta debugging approach [7]. It is capable of reasoning about causality at the granularity of individual instructions and variables, generating much more informative and precise failure explanations compared to other techniques [8]. The technique involves replacing part of the state in the correct execution with that from the buggy execution and determining whether such replacement induces the failure in the modified execution. However, due to the complexity of program state (e.g. inter-connected data structures in the heap, pointers, and external resources), it faces many problems in practice. In particular, entangling the states from both executions allows them affect each other in undesirable and unexpected ways, leading to poor failure explanations. More discussion of the limitations of the technique can be found in Section II.

In this paper, we propose a novel fine-grained causal inference technique. Given two executions and some observed differences between them, the technique can precisely reason about the causes of such differences. While the technique reasons about causality through state replacement, it makes three key advances. It features a novel execution model that avoids undesirable entangling of the replaced state and the original state such that the precision of causal inference can be substantially improved. It is capable of handling execution omission errors by analyzing both executions symmetrically. It also leverages an existing slicing technique called dual slicing [9] to limit the scope of causality testing while ensuring no relevant state differences can be missed. As a result, the efficiency is substantially improved.

Our main contributions are highlighted as follows.

- We first thoroughly discuss the limitations of the state of the art fine-grained causal inference technique that has been used for many years. We especially study the problems in state replacement.
- We propose a novel causal inference model that is symmetric and comparative. We declare the goals of the model, which reflect the user's intention when reasoning about software behavior by comparison.
- We propose a novel realization of the model. It leverages dual slicing to ensure relevance of the causes and limit the scope of causality testing. While it makes use of state replacement to determine causality, a novel execution model and its approximation are developed to avoid the undesirable entangling of the state from both executions.
- We implement and evaluate a prototype. We apply the causal inference engine toward failure explanation for 15 real world bugs, including all the reported bugs for `tar`, `make`, and `grep` in a one year period. Comparison against the causal inference engine from the most recent improved delta debugging [8] and dual slicing techniques shows that our technique has substantially improved the efficiency and effectiveness of failure explanation.

II. Causal State Minimization in Delta Debugging

Delta debugging is a classic debugging technique that can minimize failure inducing inputs [10] or the faulty internal program state essential to reproducing a failure [7], [11]. The original work first contrasts a buggy execution with a similar correct execution to determine state differences [7], [11]. It then performs *Causal State Minimization* (CSM) to determine

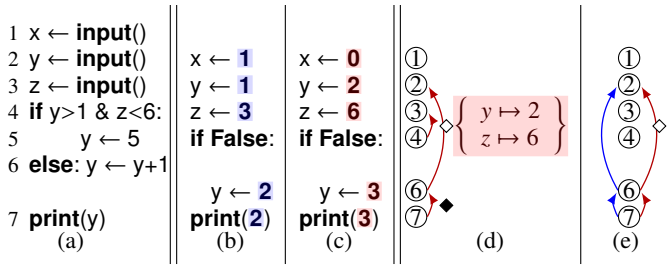


Fig. 1: (a) A program. (b-c) executions with differing input. (d) CSM. (e) dual slice. Symbols ◇ and ◆ denote the cause point and effect point, respectively. The set in (d) represents the causal state set.

the minimal subset of state differences essential to reproducing the failure. CSM involves performing the correct execution up to a point of interest preceding the failure, called *the cause point*, replacing a subset of program state with state from the buggy execution, and continuing this patched execution to determine whether the failure can be induced. If so, the subset is called a *causal state set* or *cause set*. The technique makes use of a generalized binary search to enumerate and test different subsets until it identifies the minimal cause set. Sumner et al. recently combined delta debugging with more precise execution alignment techniques [8], [12] to improve its robustness, precision, and efficiency. By applying CSM inductively, a causal chain or *summary* of a failure can be computed, comprising a sequence of the minimal causal state sets computed for a sequence of execution points leading from the root cause to the failure [8], [13].

Example. Consider the simple program presented in Fig. 1. This program reads three integers, re-defines one of them, and then prints it. In the execution of (b), the user inputs 1, 1, 3 and the program prints 2. In contrast, in the execution (c), the user inputs 0, 2, 6 and the program prints 3. Suppose that execution (c) is buggy. Given the buggy output 3 on line 7, called the *effect point*, we apply CSM to determine what state on line 4, called the *cause point*, actually caused the buggy output. The cause and effect points are respectively marked in the figure as empty and filled diamonds in (d).

Note, here the term "*buggy*" is a generalized notion as there is not a faulty statement per se. *Any* behavioral difference between the executions may be considered buggy and we are interested in what caused these differences. The discussion and the technique are universally applicable for cases where true faults cause the behavioral differences.

CSM repeatedly replays execution (b) up to line 4. Each time, it then replaces a subset of state with state from execution (c) to identify a subset sufficient to produce y↦3 within execution (b). For instance, replacing (on line 4) the variable/value mappings y↦1 and z↦3 in execution (b) with y↦2 and z↦6 from execution (c) yields y↦3 on line 7. Thus, the process identifies that the values of y and z are buggy on line 4 in execution (c), leading to the buggy output. Fig. 1d presents the causal state set on line 4 along with relevant program dependences for comprehension. The computation continues in order to determine whether a smaller causal set can be identified. If not, the identified minimal set will be reported.

If we desire a summary of the failure, the current cause point becomes the new effect point and the identified causal state set becomes the new target buggy state. The algorithm then continues to compute the causal state set for a preceding new cause point, until no such sets can be computed [8].

Limitations. Delta debugging [7], [11] and its recent improvements [8], [13] all use CSM. While prior research demonstrated the effectiveness of these techniques, we find that inherent limitations of CSM often lead to low quality failure summaries. Next, we discuss these limitations in detail and motivate the need for a new causal inference engine.

a) Confounding caused by Partial State Replacement: The first problem with CSM is that *replacing only a subset of the state in an execution can induce new behavior that was not present in **either** of the original executions*. We call this problem the *confounding of partial state replacement*. The introduced new behavior can affect the validity of a causality test. Particularly, a causal chain may terminate prematurely because key buggy state is excluded due to confounding, or it may contain additional state that does not pertain to the failure. In the worst case, the entire chain may not even be relevant for explaining the failure. From our experiments, 11 of the 15 real bugs suffered from this problem.

For example, consider the program presented in Fig. 1. Previously, we showed that CSM can determine that {y↦2,z↦6} is the causal state set on line 4. Suppose CSM tries to further consider a smaller subset {y↦2}. When it replaces the value of y in execution (b) with that from (c), the condition of the **if** statement becomes **True**, and y is redefined by line 5, rendering the target state y↦3 uninducible. Because of that condition, CSM finds that replacing the values of both y and z is necessary. Note, however, that z is unrelated to the original behavioral difference. The only contribution of z in both executions is its use on line 4, which had the value **False** in *both* executions. Ideally, only the definition of y↦2 should be blamed for the failure.

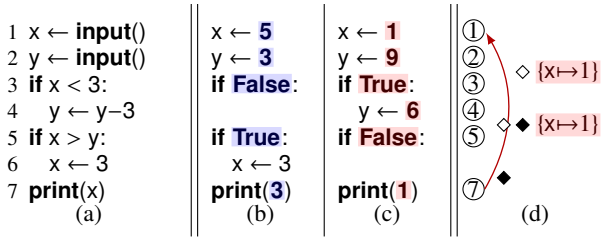


Fig. 2: Missing causes by execution omission. (a) program. (b-c) executions with differing input. (d) CSM result.

b) Execution Omission: The second problem is that *CSM may miss important causal state in the presence of execution omission errors [14]*, where the buggy target state is produced because statements were *not executed* due to the bug. In such cases, the computed failure summaries are usually incomplete. The root cause of the problem is that *CSM is **asymmetric***, meaning the buggy and correct executions have asymmetric roles in the process: CSM reasoning is based on modifying state only in the correct execution; its final results only include information from the buggy execution.

Fig. 2 presents an example. The correct execution in (b) follows the **False** branch of line 3, then the **True** branch of line 5, and prints 3, whereas the "buggy" execution in (c) follows the **True** branch of line 3, then the **False** branch of line 5, and prints 1. Suppose that initially the effect point is line 7 and the cause point is line 5. CSM determines that replacing the value of x is sufficient to induce the buggy target state in (b), so it identifies x↦1 as the only buggy state at the cause point. However, the buggy output x↦1 on line 7 in (c) is due to the undesirable omission of line 6, which is partially determined by the buggy state of y↦6. Missing y↦6 in the cause set leads to an incomplete summary of the failure.

Suppose the computation continues backward with a new effect point on line 5 and new cause point on line 3. CSM determines that replacing the value of x on line 3 is sufficient to induce the buggy target state x↦1 on line 5. Fig. 2d shows the result of this analysis. This implies x↦1 is the sole root cause of the bug. However, replacing the value of x on line 3 in (b) cannot induce the final failure although it can induce x↦1 on line 5, because line 3 evaluates to **True** in the patched execution. Hence, line 4 produces y↦0 and leads to to x↦3.

In our experiments, 5 of the 15 real bugs face this problem.

c) Efficiency: CSM may demand a large number of reexecutions. The number of state differences can be as large as the size of the allocated memory [8]. The number of possible subsets that need to be tested for causality is potentially combinatorial in terms of the full set. To combat this, existing approaches use *delta debugging* [7] to perform a generalized binary search over the subsets. However, the number of reexecutions can still be quadratic in the size of all used memory. Even the most recent implementation of CSM [8] may take a few hours to reason about a failure while the original execution time is just a few milliseconds.

III. Comparative Causality

In this paper, we propose a more effective and precise causal inference model called *comparative causality* (CC). This model focuses on *symmetrically* reasoning about two executions, one buggy and one correct[1], in order to explain why they both differ from eachother. It also enables efficient and practical implementation. In the following, we first define a number of notations and concepts. Then we study the intended properties of the new model. Here we assume we can properly align the control flow and the variables/memory regions of the two executions for fine-grained comparison using existing work [8], [12].

- **Execution point:** We use a superscripted label l^e to denote a point in execution e. Symbol $l^{(e_1,e_2)}$ denotes a point that appears in both executions e_1 and e_2, determined by the given control flow alignment [12]. It is also called an *aligned point*.
- **State difference:** we use $\{x \mapsto (v_1,v_2)\}$ to denote that a variable x has value v_1 in e_1 and value v_2 in e_2, with $v_1 \neq v_2$.

Problem Statement: Given a set of state differences Δ at an aligned execution point $l_{\blacklozenge}^{(e_1,e_2)}$ and a preceding aligned point $l_{\diamond}^{(e_1,e_2)}$, we want to find a set of state differences at $l_{\diamond}^{(e_1,e_2)}$ that is *relevant*, *sufficient*, and *minimal* for inducing Δ.

[1]How to acquire a correct execution given only the buggy execution can be found in a survey [15].

The preceding execution point is the *cause* point and the latter one the *effect* point. We demand aligned points because state comparison is not meaningful at non-aligned points. An inducing state difference in the cause point is called a *cause*; a state difference in Δ is called an *effect*.

A. Property One: Relevance

The causes identified by CC must be *relevant* to the target effects. Intuitively, a difference d is relevant to a later difference d_s if d_s is (transitively) produced from d through a sequence of differences. It represents the notion that "buggy state must be derived from preceding buggy state (except at the root cause)".

Consider the example presented in Fig. 1. The state difference $\{z \mapsto (3,6)\}$ on line 3 is not relevant to $\{y \mapsto (2,3)\}$ on line 7 even though there is a dynamic dependence path from line 3 to line 7, because the difference of z is neutralized on line 4, which yields **False** in both runs. In contrast, The difference $\{y \mapsto (1,2)\}$ on line 2 is relevant to $\{y \mapsto (2,3)\}$ on line 7.

The formal definition is as follows:

Definition 1 (Relevance): A state difference $\delta_\diamond$ at $l_{\diamond}^{(e_1,e_2)}$ is *relevant* to a target state difference $\delta_\blacklozenge$ at a later effect point $l_{\blacklozenge}^{(e_1,e_2)}$ if either of the following conditions is satisfied.

1) There exists a dynamic program dependence path from $\delta_\blacklozenge$ to $\delta_\diamond$ in e_1 (e_2) where all the statement computations along the path yield different results from the other execution e_2 (e_1).
2) There exists a state difference δ_x in an aligned point in between $l_{\diamond}^{(e_1,e_2)}$ and $l_{\blacklozenge}^{(e_1,e_2)}$ such that $\delta_\diamond$ is relevant to δ_x and δ_x is relevant to $\delta_\blacklozenge$.

Condition (1) expresses the requirement that a difference cannot be neutralized within an execution in order to be relevant. Note that it is symmetric to both executions as relevance can be determined by a dependence path in *either* execution. It allows us to precisely capture relevance in the presence of execution omission. Consider the example in Fig. 2, state difference $\{y \mapsto (3,6)\}$ on line 5 is relevant to $\{x \mapsto (3,1)\}$ on line 7, due to the dependence path $y@5 \leftarrow \texttt{True}@5 \leftarrow 6 \leftarrow 7$ in (Fig. 2b). Observe that there is no dependence between $y@5$ and $x@7$ in the failing execution (Fig. 2c) due to the omission of line 6. The intuition is that omission is an asymmetric concept regarding one execution. An omitted statement regarding one execution implies that it appears in the opposing execution. With our symmetric definition, omissions are conceptually precluded.

Condition (2) expresses that relevance can be transitive, even across the two executions.

B. Property Two: Sufficiency

The identified set of causes must sufficiently induce the target effect of *each* of the two executions within its opposing execution. This inducement acts as a new causality test and witnesses the causal relationship between the identified causes and the target state.

The property is *symmetric* as it requires the set of effects in either execution to be induced by the causes. It means that

if for all the variables in the cause set, we copy their values from execution e_1 to e_2, we can induce the target effect of e_1 at the effect point in e_2, *and vice versa*.

Consider the example in Fig. 2. State differences {y↦(3,6), x↦(5,1)} on line 5 form a sufficient set regarding the effect {x↦(3,1)} on line 7. In contrast, the state difference {x↦(5,1)} itself is not sufficient because although replacing x's value 5 with 1 in (b) can induce the effect {x↦1} on line 7, replacing x's value 1 with 5 in (c) cannot induce the effect {x↦3}. This symmetry ensures that we capture relevance due to execution omission.

More formally,

Definition 2 (Sufficiency): A cause set $\Delta_\diamond$ at $l_\diamond^{(e_1,e_2)}$ is *sufficient* for a given target effect set $\Delta_\blacklozenge$ at a later effect point $l_\blacklozenge^{(e_1,e_2)}$ if and only if, in the absence of confounding, copying the state of e_2 in $\Delta_\diamond$ to e_1 at the cause point induces the effect of e_2 in $\Delta_\blacklozenge$ in execution e_1 at the effect point, and vice versa.

One key condition is that reexecution should be confounding-free. Unfortunately, normal program execution cannot guarantee this. The remainder of this subsection focuses on discussing confounding.

1) What is confounding: Determining sufficiency involves replacing part of the state in one execution with values from the opposing execution. However, the continuation of the modified execution has the state from both original executions entangled, affecting each other and inducing undesirable and unexpected results in causal inference.

Recall in Fig. 1, we saw that partially changing the state of execution (b) with the single desired cause variable y yielded output different than in either execution (b) or (c). In addition, we found that including z as a cause along with y would yield the target state, although z is not relevant to the output. Both of these are unexpected results that we call *confounding from partial state replacement*. These confounding effects do not just have the ability to *include* arbitrary state within the set of identified causes, they can *exclude* arbitrary state, as well. Examples are omitted due to the space limitations.

At a high level, these unexpected results occur because partial state replacement *created new behaviors that did not exist in either of the original executions*.

Definition 3 (Confounding): Given executions e_1 and e_2 as well as a patched execution e_p constructed from them, a causality test using e_p is *confounded* if either of the following conditions are satisfied:

1) An execution point in e_p is not present in e_1 or e_2.
2) A data dependence in e_p is not exercised in e_1 or e_2

Condition (1) corresponds to *control flow confounding* and (2) to *data flow confounding*, which means confounding can occur without exhibiting any new control flow.

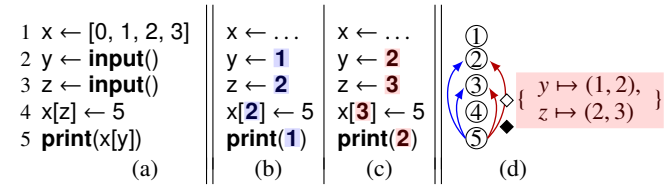


Fig. 3: Data flow confounding example. (a) program. (b-c) executions with differing input. (d) confounded explanation.

Consider the example in Fig. 3. This time, the target state is {x[y]↦(1,2)} with cause and effect points at lines 4 and 5 respectively. Observe that in each execution, the read from and written to elements of x are different. Thus, the only identified cause for the different output should be the differing values of y, which provides the index read from the list. However, when only the value of y is replaced on line 4 in (b), the patched execution reads the new value written to the list on line 4. Thus, the target state is not induced. Observe that in this case, a new data dependence from line 5 to line 4 is exercised.

In later sections, we will examine new execution models that can avoid/mitigate confounding.

We argue that the two properties, together with the minimality requirement, are essential for understanding execution differences. They precisely express the programmer's intentions.

IV. Realizing Comparative Causal Inference

In this section, we discuss the realization of CC. Given a target effect set and a cause point, we leverage a technique called dual slicing to compute a set of candidate causes and only apply causality testing on the candidate set. Dual slicing is a symmetric slicing technique that works on two executions. It first determines control flow and value differences in the two executions through trace comparison and then performs slicing on these differences (in and across both executions). The benefits of using dual slicing are twofold. First, it ensures relevance of the candidates. Second, it is more efficient because causality testing only needs to enumerate subsets of the candidates instead of the full set of state differences as in CSM [7], [8].

After acquiring the dual slice, we then *symmetrically* minimize the causes included in the slice to a minimal subset sufficient for inducing the target state within *both* executions. During the minimization process, one key step is to perform causality testing by state replacement. In order to avoid confounding, we devise an execution model that harnesses a patched execution in such a way that it respects the control flow and dependences in the two original executions while allowing flexibility for reasoning about the effects of state replacement.

A. Background: Dual Slicing

Dual slicing was first introduced to study concurrency bugs [9] and software vulnerabilities [16].

Algorithm 1 presents the basic dual slicing algorithm. Although it is not part of this paper's contributions, we present a simplified version of the algorithm for completeness.

Given a slicing criterion, an execution point that exhibits a state difference, the algorithm returns its dual slice, a set of dynamic dependences from both executions denoting the causality of the difference. Lines 1-7 describe the process of slicing in execution e_1. It first ensures that the current criterion $l_\blacklozenge$ is present in e_1 (line 1). Here, $l_\blacklozenge^{(\perp,e_2)}$ denotes that $l_\blacklozenge$ is not present in e_1. Lines 2-4 traverse each dynamic data dependence edge of the criterion in e_1 with x, the variable involved, denoted as $l_\blacklozenge^{(e_1,e_2)} \xrightarrow[e_1]{x} l_\diamond^{(e_1,e_2')}$. We use variable e_2' to represent that

Algorithm 1 Dual Slicing

dualSlice($l_{\blacklozenge}^{(e_1,e_2)}$)

Input: $l_{\blacklozenge}^{(e_1,e_2)}$ - the slicing criterion
Output: $\mathcal{D}$ - the dual slice, a set of deps in either execution

```
1: if e1 ≠ ⊥ then
2:     for each data dep dd ← {l◆^(e1,e2) —x,e1→ l◇^(e1,e2')} do
3:         if e2' ≡ ⊥ or x has different values on l◇ then
4:             D ← D ∪ dd ∪ dualSlice(l◇^(e1,e2'))
5:     control dep cd ← {l◆^(e1,e2) ⟹e1 l◇^(e1,e2')}
6:     if e2' ≡ ⊥ or l◇ has different branch outcomes then
7:         D ← D ∪ cd ∪ dualSlice(l◇^(e1,e2'))
8: if e2 ≠ ⊥ then
9:     /* operations symmetric to when e1 ≠ ⊥ */
10: return D
```

$l_\diamond$ may or may not be in the second execution, disregarding the value of e_2. On line 3, if $l_\diamond$ is exclusively in e_1 (i.e, $e_2' \equiv \perp$), meaning that it is a control flow difference, or even though it is not exclusive, the variable x has different values in the two executions, the data dependence is added to the slice (line 4). The dual slice of $l_\diamond$ is recursively computed and added to the slice too (line 4). This suggests that when $l_\diamond$ is present in both executions and produces the same value, it is not added because it cannot induce the criterion. In lines 5-7, the algorithm traverses the control dependence edge in e_1, denoted as "$\underset{e_1}{\Longrightarrow}$". Similarly, if the guarding predicate is exclusive or has different branch outcomes, the edge gets added and the dual slice of the predicate is recursively computed. Lines 8-9 are symmetric to lines 1-7, describing the process of slicing in execution e_2.

```
1  t ← input()
2  x ← input()
3  y ← input()
4  z ← input()
5  if x + y + z > 3:
6      z ← −10
7  if x + y + z > 0:
8      z ← 5
9  if z < 0 and y > 0:
10     z ← t
11 else: print(z)
```
(a)

```
t ← 0
x ← 1
y ← 0
z ← 4
if True:
    z ← −10
if False:

if False:

    print(-10)
```
(b)

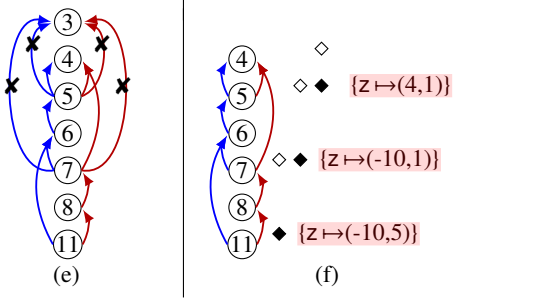
```
t ← 1
x ← 1
y ← 1
z ← 1
if False:

if True:
    z ← 5
if False:

    print(5)
```
(c)

```
y ← input()
z ← input()
if 1+y+z >3:
    z ← −10
if 1+y+z >0:
    z ← 5

print(z)
```
(d)



(e) (f)

Fig. 4: (a) program. (b-c) two runs. (d) program from the dual slice. (e) dual slice. (f) CC explanation.

Example. Consider the program in Fig. 4a. The dual slice of the two executions, (b) and (c), is presented in Fig. 4e (including the crossed-out dependences). Part of the computation is represented as follows. We use *dS()* as a shorthand for *dualSlice()*. The superscripts of execution points are elided for brevity when explicit from the context. The box in a step denotes that the next step is to execute the recursive call inside.

$$dS(11^{(b,c)}) = \{11 \xrightarrow[b]{z} 6\} \cup \boxed{dS(6^{(b,\perp)})} \cup \{11 \xrightarrow[c]{z} 8\} \cup dS(8^{(\perp,c)}) \quad [1]$$
$$= \{11 \xrightarrow[b]{z} 6,\ 6 \Longrightarrow_{b} 5\} \cup \boxed{dS(5^{(b,c)})} \cup \{11 \xrightarrow[c]{z} 8\}... \quad [2]$$
$$= \{11 \xrightarrow[b]{z} 6,\ 6 \Longrightarrow_{b} 5, 5 \xrightarrow[b]{z} 4, 5 \xrightarrow[c]{z} 4, ...\}... \quad [3]$$

Observe at step [1], the control dependence to line 9 is not involved as it has the same branch outcome in the two runs. Also, observe that dual slicing line 6 of execution (b) in step [1] entails slicing line 5 in both executions (step [2]). Line 1 is not included, even though it denotes a difference, as it is not reachable from the criterion.

The dual slice captures the behavioral differences of the two executions related to the criterion.

B. Dual Slices are Relevant, but Confounding-prone and Redundant

Dual slices are represented in terms of dependences, whereas causal inference is conducted on program state. Hence, we first introduce a projection from a dual slice to the corresponding set of state differences at a given execution point so that we can discuss the properties of dual slicing in our context. These properties are unique to the proposed technique and have not been studied before.

Given a dual slice and a cause point $l^{(e_1,e_2)}$, which is an aligned point, we define the cut of the dual slice with respect to the point as follows.

$$C(\mathcal{D}, l^{(e_1,e_2)}) = \{x \mapsto (v_1, v_2) \mid l_{\blacklozenge}^{e_t} \xrightarrow[e_t]{x} l_{\diamond}^{e_t} \in \mathcal{D}, \text{ with } l_\diamond \prec_{e_t} l \prec_{e_t} l_\blacklozenge \text{ or } l_\blacklozenge \equiv l, \text{ and } x \mapsto (v_1, v_2) \text{ on } l \text{ with } v_1 \neq v_2\}$$

It denotes the set of state differences involved in the dual slice on the given cause point. It essentially denotes the set of variables when we cut the dual slice on the cause point. Symbol $l_a \prec_{e_t} l_b$ denotes l_a precedes l_b in execution e_t.

Consider the dual slice in Fig. 4e. The cut on line 7 is the following. $C(\mathcal{D}, 7^{(b,c)})$ = {z↦(-10,1), y↦(0,1)}. Note that {t↦(0,1)} is not in the cut.

Theorem 1: All the causes in a cut $C(\mathcal{D}, l^{(e_1,e_2)})$ are relevant to the slicing criterion. All relevant causes on $l^{(e_1,e_2)}$) are included in its cut.

The proof is omitted due to space limitations. The property suggests that dual slices cover all the causes the programmer needs to inspect.

Unfortunately, a dual slice cut may not sufficiently induce the slicing criterion given the confounding-prone regular execution model. That is, replacing the state of all causes in a cut may not induce the failure. Let us revisit the example in Fig. 1. The dual slice is shown in Fig. 1e. Its cut on line 4 has only y. However, from the discussion in Section II, we know that replacing y↦1 with y↦2 in execution (b) does not lead to the target effect due to the confounding from z.

A cut may also not be minimal. It may contain causes that are not essential for inducing the target effect. In Fig. 4e, the cut on line 7 is {z↦(-10,1), y↦(0,1)}, but the minimal sufficient set is just {z↦(-10,1)}.

These limitations motivate us to realize the proposed CC by performing confounding-free minimization on dual slices.

C. The Basic Algorithm

In this subsection, we introduce the basic minimization algorithm, assuming a confounding-free execution model. We will discuss the execution model in the next subsection.

Algorithm 2 Minimizing Causes

inferCauses($\mathcal{D}$, $l_{\diamond}^{(e_1,e_2)}$, $l_{\blacklozenge}^{(e_1,e_2)}$, $\Delta_{\blacklozenge}$)

Input:	$\mathcal{D}$ - the dual slice	$l_{\diamond}$ - the cause point	
	$l_{\blacklozenge}$ - the effect point	$\Delta_{\blacklozenge}$ - the target state	
Output:	causes of target at $l_{\diamond}$		

1: $\Delta \leftarrow C(\mathcal{D}, l_{\diamond})$
2: $\Delta_{min} \leftarrow \Delta$
3: **for each** $s \subset \Delta$ by delta debugging **do**
4: **if** $|s| < |\Delta_{min}|$
$\wedge\ \mathcal{E}^{e_1}[s\downarrow_{e_2} / s\downarrow_{e_1}]_{l_{\blacklozenge}}^{l_{\diamond}} \rightsquigarrow \Delta_{\blacklozenge}\downarrow_{e_2}$
$\wedge\ \mathcal{E}^{e_2}[s\downarrow_{e_1} / s\downarrow_{e_2}]_{l_{\blacklozenge}}^{l_{\diamond}} \rightsquigarrow \Delta_{\blacklozenge}\downarrow_{e_1}$ **then**
5: $\Delta_{min} \leftarrow s$
6: **return** Δ_{min}

The basic algorithm is presented in Algorithm 2. Given a precomputed dual slice, the cause and effect points, and the target state, the algorithm returns a minimal set of causes sufficient to induce the target state. The algorithm starts by computing a dual slice cut at the cause point, which is essentially the set of relevant causes. Lines 3-6 minimize the set to only those sufficient for inducing the observed target state of each execution in the other. We leverage the delta debugging algorithm to enumerate subsets of the relevant causes and test their causality. Symbol $\mathcal{E}^{e_1}[s\downarrow_{e_2} / s\downarrow_{e_1}]_{l_{\blacklozenge}}^{l_{\diamond}}$ means executing e_1 up to the cause point $l_{\diamond}$, replacing its variable/value mappings in s with those from e_2, and continuing the execution up to the effect point $l_{\blacklozenge}$. Symbol $s\downarrow_{e_1}$ denotes the projection of state differences s on execution e_1. If the variables in the target state have the values from e_2, we say that the target state of e_2 was induced, written $\rightsquigarrow \Delta_{\blacklozenge}\downarrow_{e_2}$.

Observe that in contrast to existing CSM approaches [7], [13], our minimization algorithm performs *two symmetric* causality checks. This is necessary to include causes via omission.

D. Confounding Free Execution Model

Recall that confounding occurs when new control flow or data dependences not in either original execution occur in a patched execution. By Theorem 1, we know that all the relevant causes are included by the dual slice. This suggests we only need to perform causality testing *within* the dual slice.

Conceptually, the essence of our new execution model is to construct a program containing only the behavior of the dual slice and all reexecutions for causality testing occur on the constructed program. Statement executions not in the dual slice should be prevented in order to minimize confounding.

Illustrative Example. Consider the example in Fig. 4. Assume we start by using the target state {z↦(-10,5)} at line 11. Assume the cause point is line 7 and we apply Algorithm 2 to minimize the causes at this point. The cut of the dual slice (Fig. 4e) involves variables y and z. When we consider variable z with a regular execution model, we reexecute (c) up to the cause point and replace the value of z with -10. It induces the false branch outcome on line 7 but the true branch outcome on line 9, which is different than execution (b). Hence, {z↦(-10,1)} is not considered a valid cause set.

With our new execution model, conceptually, we construct a program representing the dual slice, as in Fig. 4d, in which lines 1, 2, 9, and 10 are precluded as they are not in the slice. Also, line 11 is no longer guarded by any predicate. Operands that are in the slice and have identical values in both executions are concretized (e.g. x on lines 5 and 7).

Again, let us determine the causality of variable z on line 7. We reexecute (c) up to the cause point using the original program. We replace the value of z with -10, *then continue execution with the program in Fig. 4d*. Since lines 9 and 10 are not in the program, we avoid confounding and can induce the desired target state. Hence {z↦(-10,1)}} is the minimal inducing cause set. Observe that it allows us to prune the relevant but not necessary cause {y↦(0,1)}}. Applying Algorithm 2 transitively, we acquire a more concise failure explanation as shown in Fig. 4f. □

Semantics of the New Execution Model. In the following, we discuss the semantics that allows achieving the effect of executing exclusively within the dual slice without explicitly constructing a new program. During minimization, we first reexecute the original program with normal semantics up to the cause point, and then continue executing the program with *the new semantics* after state replacement, until the effect point.

In the semantics, we assume the runtime availability of the dual slice $\mathcal{D}$ and the traces of the original two executions, denoted by $\mathcal{T}^{e_{1/2}}$. Without losing generality, we assume we are patching e_1 using information from e_2. The value of a variable x at a point l^{e_1} in the original execution e_1 can be queried from the trace by $val(\mathcal{T}^{e_1}, l, x)$. If an execution point l^{e_1} is a conditional statement, $branch(\mathcal{T}^{e_1}, l)$ queries its branch outcome in execution e_1. The semantics is presented in Fig. 5.

Statement executions not in the dual slice are skipped when they are not conditional statements (Rule 1). When executing conditional statements, we cannot simply skip as we need to select a branch to proceed. Rules 2-3 specify the cases for conditional statements.

In Rule 3, if a conditional had different branch outcomes originally or it was present in only one execution, the semantics evaluates the predicate and follows the computed branch. The essence is to allow the flexibility to take either branch based on the predicate evaluation in order to reason about the effect of state replacement when it is in the dual slice.

1) When l is not a conditional with $l \notin \mathcal{D}$, skips l.
2) When l is a conditional and it was in both executions with $branch(\mathcal{T}^{e_1}, l) \equiv branch(\mathcal{T}^{e_2}, l)$, unconditionally continue with the same branch as in the original executions.
3) When l is a conditional and it was in both executions with $branch(\mathcal{T}^{e_1}, l) \neq branch(\mathcal{T}^{e_2}, l)$ or l is in only one execution, evaluate the statement according to Rule 4) and follow the computed branch.
4) When l is not a conditional with $l \in \mathcal{D}$, validate that all the operands involved in some data dependence in $\mathcal{D}$ have the same data dependence as they did in the original executions, otherwise terminate and report confounding; For any operand not in any dependences in $\mathcal{D}$, denoted as x, set its value to $val(\mathcal{T}^{e_x}, l, x)$, and continue.

Fig. 5: Semantics of $\mathcal{E}[]$.

If the statement is not in the dual slice, it does not matter which branch is taken because all non-conditional statements inside the branches must be skipped according to Rule 1. These statements must not be in the dual slice; otherwise, the conditional would have been in the slice according to the dual slicing algorithm.

Rule 4 handles non-conditional statement execution in the dual slice, for all the operands not involved in any dependences in the slice, implying that they must have identical values in the two executions, we concretize them with values from the traces to achieve isolation. For operands involved in some dependence, we ensure no data flow confounding.

This new model will not allow any confounded executions to go through, as can be inferred from the semantic rules.

Theorem 2: A dual slice cut is sufficient within the new execution model.

This theorem ensures that Algorithm 2 must be able to find a minimal sufficient set of causes inducing the target state because in the worst case, the cut is the minimal set. Informally, the theorem holds because reexecution is exclusively within the dual slice and hence replacing all the state in a cut leads to a reexecution equivalent to the part of the dual slice belonging to the opposing execution, and hence the target state.

A Practical Approximation. Unfortunately, the semantics in Fig. 5 demands a prohibitively expensive implementation. It requires collecting traces with dependences and values. The traces and the dual slice have to be accessed during each reexecution. Each statement has to be instrumented to decide if it is in the dual slice (Rule 1) or perform complex control (Rules 2-4). The overhead could easily be many orders of magnitude, not affordable for repeated reexecutions.

In practice, we observe that control flow confounding is the dominant confounding factor and data flow confounding can only affect the execution by causing control flow confounding in most cases. We hence propose a practical approximation that can completely prevent control flow confounding and mitigate data flow confounding. The approximate model ensures a patched execution can only follow dynamic branches taken by at least one of the original executions. Consequently, it enforces a control flow path composed of segments that occurred in either execution. What we do here is essentially constructing guard rails for the execution so that it can never deviate from the dual slice's control flow. Since data dependences heavily depend on control flow, the approximation can also mitigate data flow confounding. The semantics is presented in Fig. 6.

1) Rule 2 from Fig. 5.
2) When l is a conditional and it was in both executions with $branch(\mathcal{T}^{e_1}, l) \neq branch(\mathcal{T}^{e_2}, l)$, evaluate the statement normally and follow the computed branch.
3) When l is a conditional and it was in only one execution e_x, follow the branch that was taken in e_x.
4) Otherwise, evaluate l as in a regular execution model.

Fig. 6: Semantics of the Approximate Execution Model.

Observe that the semantics does not require the runtime of the dual slice or dependence/value traces for runtime checking, but rather just the control flow trace. This can be very efficiently represented and accessed by using bit streams that simply record the sequence of boolean branch outcomes. It does not skip statements. It hence avoids instrumenting all statements to decide if one can be skipped at runtime.

Theorem 3: The approximate execution model is free of control flow confounding.

The theorem can be inferred from the semantic rules. We implemented the approximate semantics and in practice it was able to suppress all confounding in our experience.

V. Evaluation

We implemented our technique using LLVM 3.0. We have also implemented the CSM and dual slicing approaches [7]–[9], [13] for comparison. Both implementations reflect the latest published designs [8], [9]. The evaluation is in the context of automated debugging. The techniques contrast buggy and correct executions, using explanations for their different behavior as explanations of bugs. First, we compute explanations for a set of real world bugs by chaining together the computed causes. We contrast the explanations computed by the three different techniques. Second, we examine in depth how the problems that CSM faces affect its results in practice.

We used real world bugs taken from the repositories of open source programs. They include all deterministic bugs from `tar`, `grep`, and `make` in a one year period that we were able to reproduce. All the bugs in our study were non-crashing, semantic bugs that produce incorrect outputs. Table I presents the full set of programs and bugs. The first three columns identify the buggy program, bug ID, and the version of the program that actually contains the bug. The SSLOC column contains the static source lines of code computed with `sloccount`. The Alt. column identifies how a second, correct execution was selected. We used a correct input when the bug report also provided it, otherwise, we used predicate switching [17] to automatically synthesize a correct execution from the failing one. More information on acquiring a correct execution from a given failing execution resides in Sumner's survey paper [15]. We performed all experiments on a 64-bit 2.4GHz CPU with 12GB RAM using one core.

A. Full Explanation Comparison

Our first experiment uses each of the three techniques to compute an explanation for each bug. For each bug, we first identify the last observable failure and use that as the initial

target state. CC and CSM select the last preceding definition of a target effect as the cause point to compute the causes. They also proceed transitively, using the computed causes as the new target state and the current cause point as the new effect point until there are no more causes to identify (e.g. the two executions have no state differences).

We contrast the results of the different techniques through their quality, scale, and efficiency. We measure *quality* through precision and recall with respect to a relevant, sufficient, and minimal explanation of why the correct and buggy executions differed. This is manually checked at each step of the computation. Precision (P) is the proportion of the dynamic statements in the computed explanation for a technique that coincide with the statements in the correct explanation. Recall (R) is the proportion of the dynamic statements in the correct explanation that are also identified by the computed explanation. We have to resort to manual inspection due to the lack of an automated oracle to tell us the ideal explanations for execution differences. As we show later, such ideal explanations are small enough for line by line human inspection.

We have done the following to mitigate threats to validity. First, we cross referenced the computed explanations with the root causes identified by the bug fixes or reports. Second, we calibrated our system using the Siemens suite before our experiments. We computed the explanations for the over 10,000 failing runs in Siemens using the corresponding passing executions of the provided correct versions and validated that these explanations capture the injected faulty statements as the root causes. The results are publicly available at `https://www.sites.google.com/site/explainedbugs/`. Third, we also release the experimental results of the real world bugs at the same site for interested readers.

We measure the *scale* of a technique by the number of dynamic statements (Stmts) in the computed explanation. Finally, we measure efficiency in three ways: the number of steps or rounds of causal inference, the clock time required in seconds, and the number of reexecutions needed. Note that the clock time of CC includes dual slicing time. Table I shows the results. From these, we make several observations.

1) CC consistently yields the highest quality explanations. Dual slicing generally has good recall but poor precision because it doesn't minimize. CSM is unpredictable because it can arbitrarily include or exclude causes, however, it frequently fails to identify causes for even a single step of an execution. We shall explore the unpredictability of CSM further in the next section. In contrast, CC yields high precision and high recall for every computed explanation. For the bugs, it captures 11 of 15 root causes whereas CSM *fails* to do so in 11 of 15 cases. Where CC failed to identify root causes, denoted by -, it still explained why the two *executions* differed, thus the precision and recall. In those cases, the second execution was too different to meaningfully explain the bugs as well.

2) The extra reexecutions for CSM make it slower than CC, even when it computes fewer steps.

On average, CSM takes just over 13.8 minutes to compute an explanation, even though it produces less of the correct explanation. In contrast, CC takes just over 2.5 minutes on average. This is because the extra dual slice information allows it to avoid considering all memory differences as potential causes. This reduces the number of necessary reexecutions by up to two orders of magnitude.

3) CC produces more concise explanations than dual slicing. The precision numbers show that CC is more precise than dual slicing, 1.0 vs 0.14. On average, CC produces explanations of 35 dynamic statements, while dual slicing produces 330 statements.

This experiment illustrates that CC produces superior explanations in terms of quality, efficiency, and scale.

B. Why and how CSM fails

A single incorrect cause at any point of the full chain computation can cascade through the rest of the computation, causing more incorrect causes. It is hence difficult to determine the reasons behind the incorrectness by simply looking at the full chains. Our second experiment examines *why and how* CSM missed or erroneously included causes on a per-step basis. Note that CC does not encounter these problems for the given benchmarks, and dual slicing does not do minimization. Thus, we focus only on CSM for this experiment.

We first computed the causes for each step using CSM as in the first experiment. For each step, we also supply the same (CSM) target state and the same cause point to CC and compare the resulting causes from the two approaches. This allows us to quickly observe any effects from confounding.

In this per-step fashion, we carefully checked the results of CSM for missing causes (M), extra causes (E), or even failure to identify any causes (F). These are the different ways that the technique can fail. We also checked *why* these failures occurred, including control flow confounding (CFC), data flow confounding (DFC), and execution omission (O). Table II contains these results.

1) CSM suffers from all three problems. It misses causes in almost all benchmarks (12 out of 15), has extra causes in 8 out of 15, and fails to produce any causes for a step in 6 out of 15 cases. These failures resulted both from omission and from confounding, although confounding was the more frequent cause.

2) Control flow confounding causes errors in most of the CSM explanations. In 11 out of 15 cases, the CSM explanations are directly impacted by control flow confounding. This shows that control flow confounding is a real world challenge that we must address.

2) Data flow confounding does not directly impact CSM. While close inspection indicates that some data flow confounding occurs, it impacts the executions only through control flow confounding. As CC prevents control flow confounding, the impact of the corresponding data flow confounding is also suppressed. For example, data flow confounding may lead to an incorrect branch, but CC forces the execution back to the correct branch through its execution model.

Together, these fine grained comparisons allow us to see that omission and confounding do indeed impact existing

TABLE I: Comparison of full explanations. Averages are arithmetic except for P and R, which are geometric. - means that the root cause could not be captured.

Program	ID	Version	SSLOC	Alt.	CC							CSM							Dual Slicing		
					Steps	Time	Tests	Stmts	P	R	Roots	Steps	Time	Tests	Stmts	P	R	Roots	Stmts	P	R
find	1	4.5.7	73k	switch	7	12	15	6	1.0	1.0	X	1	253	1260	0	0	0	-	185	0.03	1.0
gnuplot	2	4.5.0	144k	switch	11	44	33	10	1.0	1.0	X	11	141	469	10	1.0	1.0	X	148	0.06	1.0
gnuplot	3	4.4.0	139k	input	35	200	323	48	1.0	1.0	X	1	51	208	0	0	0	-	464	0.07	1.0
gnuplot	4	4.2.4	134k	input	146	961	337	129	1.0	1.0	-	127	950	1888	121	0.97	0.91	-	368	0.33	1.0
gnuplot	5	4.2.4	134k	switch	24	140	130	33	1.0	1.0	-	31	931	3012	38	0.87	1.0	-	237	0.14	1.0
grep	6	2.5.4	12k	switch	59	114	186	62	1.0	1.0	-	24	8263	1012	23	0.96	0.35	-	153	0.51	1.0
grep	7	2.5.4	12k	switch	45	156	327	69	1.0	1.0	-	33	183	1734	32	1.0	0.46	-	109	0.62	1.0
grep	8	2.5.4	12k	switch	27	49	78	27	1.0	1.0	X	24	168	1546	23	0.96	0.81	-	95	0.26	1.0
make	9	3.81.90	30k	switch	27	342	62	27	1.0	1.0	X	18	416	543	17	1.0	0.63	-	38	0.66	1.0
tar	10	1.22.90	20k	switch	5	22	8	3	1.0	1.0	X	5	50	221	3	1.0	1.0	X	3	1.0	1.0
tar	11	1.22.90	24k	input	30	124	125	48	1.0	1.0	X	1	110	332	0	0	0	-	61	0.79	1.0
tar	12	1.22.90	20k	input	9	53	121	20	1.0	1.0	X	1	66	296	0	0	0	-	1239	0.01	1.0
tar	13	1.22.90	20k	switch	11	43	28	10	1.0	1.0	X	6	439	2117	5	1.0	0.5	-	1270	0.01	1.0
tar	14	1.23	21k	input	17	80	87	23	1.0	1.0	X	5	165	709	15	0.73	0.48	X	25	0.92	1.0
tar	15	1.23	21k	switch	5	22	15	4	1.0	1.0	X	5	228	1283	4	1.0	1.0	X	557	0.01	1.0
				Average	30.5	157.4	125	34.6	1.0	1.0	-	19.53	827.6	1108.7	19.7	0.22	0.26	-	330.1	0.14	1.0

TABLE II: CSM difficulties. This includes symptoms: (M)issing causes, (E)xtra causes, and complete (F)ailure. It also lists reasons why: control and data flow confounding (CFC/DFC) or (O)mission.

ID	M	E	F	CFC	DFC	O
1	X	-	X	X	-	-
2	-	-	-	-	-	-
3	X	X	X	X	-	-
4	X	X	-	X	-	X
5	X	X	-	X	-	-
6	X	-	X	X	-	X
7	X	-	-	-	-	X
8	X	X	-	X	-	X
9	X	X	-	X	-	X
10	-	-	-	-	-	-
11	X	X	X	X	-	-
12	X	X	-	X	-	-
13	X	-	X	X	-	-
14	X	X	X	X	-	-
15	-	-	-	-	-	-

techniques. Furthermore, taken with the results in Table I they show that CC is resilient when faced with them.

C. Example of Resulting Explanations

Next, we demonstrate a failure explanation generated by CC and explain how CSM fails to compute that explanation. This chain is for bug 13. Version 1.22.90 of `tar` has a bug when using the `--backup` option. When extracting files from an archive, this option copies any already existing files into a backup directory, preventing these files from being overwritten. When extracting a directory that already exists, however, it *appears* to incorrectly prevent files from being extracted.

We used predicate switching to dynamically patch the buggy execution and derive a correctly behaving execution. Both the buggy and the switched executions first extract some files before trying to extract a directory that already exists. The switched execution renames the extracted directory so that it does not conflict with the existing one, and it correctly extracts files to the new directory without error. However, the buggy

Code Summary

```
1  int read_header_primitive():
2    file_name = "dir" vs. "dir2"
3
4  int extract_dir(file_name):
5    tmp = mkdir(file_name)
6    ...
7    status = tmp
8    if status:
9      if errno == EEXIST && IS_DIR(file_name):
10       pass
11     elif !maybe_recoverable(filename):
12       mkdir_error(file_name);
13   return status
14
15 void extract_archive():
16   ...
17   status = extract_dir(file_name)
18   if status && backup_option:
19     undo_last_backup()
```

Explanation

At 2, file_name **is** "dir" vs. "dir2".
At 5, tmp **is** 0 vs. -1.
At 7, status **is** 0 vs. -1.
At 13, the **return** value **is** 0 vs. -1.
At 17, extract_dir returns **is** 0 vs. -1.
At 18, (status && backup_option) **is** False vs. True.
So "undo_last_backup()" **is** called, overwriting the extracted files with the original ones.

Fig. 7: Example of a derived explanation using our technique

execution appears to have not extracted any files at all, even the previously extracted ones.

Fig. 7 shows a simplified version of the relevant code, as well as the explanation by CC, which is slightly shortened for readability. First, predicate switching renames one of the extracted directories from "dir2" to "dir". Next, a call to mkdir() fails in the buggy execution, returning −1 because "dir2" already exists. In contrast, the call succeeds in the switched execution and returns 0. This difference (0 vs. −1) gets propagated through the variable status back into extract_archive(), where it

makes the condition on line 18 **True** only in the failing execution, indicating an error when extracting "dir2". So the buggy execution calls undo_last_backup(). This actually *replaces all of the extracted files* with the original backups. As a result, all of the files extracted before "dir2" *appear* to never have been extracted, even though they were. In fact, the original bug reports for this failure assumed the files had not been extracted, as well, but our generated explanation clearly shows that they were first extracted and then incorrectly overwritten.

The root cause is that extract_dir() should not fail even if mkdir() fails due to the existence of the directory, because extracting to an existing directory should not cause problems. A `tar` developer can see this from the computed explanation (on the bottom of Fig. 7) and know how to construct a fix. Indeed, the applied fix set status to 0 on line 10.

Note, CSM cannot construct this explanation. Once it gets to line 13, confounding prevents any further analysis of the bug. First, the condition on line 9 only executes in the failing execution, where it is **True**. CSM tries to replace the value of tmp at line 7 to produce the failing status at line 13, but the condition on line 9 evaluates to **False** this time because it also requires a failing value for errno. Hence, CSM proceeds to line 12, which reports an unrecoverable error and terminates. This confounding prevents the identification of tmp alone as the cause. Additional confounding not shown here also prevents replacing both errno and tmp from inducing the failing status.

D. Threats and Limitations

We have shown that CC is effective at explaining why two executions are different, but there are limits to the technique, our evaluation, and what may be inferred from it.

We first note that explaining why a buggy and correct execution differ does not always provide a useful explanation of a bug, as observed in 27% of our generated explanations.

Also, manual examination of execution differences risks human error. Most of the explanations generated by CC are short enough that we can be confident of our inspection.

Finally, again, comparative causality is presently limited to examining deterministic bugs. This inherently follows from exploiting reexecution within the technique.

VI. Related Work

The most relevant work is causal state minimization (CSM) that was originally introduced by Zeller [7], and subsequently improved by others [8], [11], [13]. In contrast to CSM, our SCC model avoids confounding, handles execution omission by symmetric analysis, and is much more efficient.

Recently, Rößler et al. also noted problems with Zeller's original approach, although they did not delve into what these problems were [18]. They also produce a technique for explaining bugs, but it is based on test generation and requires a strong oracle to evaluate each new test.

Traditional dynamic slicing [19] is a technique that captures dynamic data and control dependences. It has been extensively examined for its usefulness in debugging [20]. Dynamic slices are usually problematically large and suffer from execution omission. Dual slicing is a kind of dynamic slicing technique that *compares two executions* and extracts the differing dependencies between the two [9], [16]. It forms the initial basis of our technique. In contrast, our computed explanations are much smaller due to state replacement and minimization.

Several satisfiability based techniques also strive to precisely explain failures, either within a single program [21], [22] or when comparing correct and incorrect versions [23], [24]. The present limitations in constraint solving, however, have thus far mostly limited these techniques to programs of a few thousand lines of code. In contrast, our technique explains failures in programs with well over 100K lines.

Our technique requires that the executions of interest be reproducible. Tools that aid failure reproduction, for instance, can make this more feasible in practice [25],

VII. Conclusions

We presented a novel causal inference technique called comparative causality. It allows precise and concise explations for the differences between two executions at a very fine granularity. It advances the state of the art in three aspects: it improves *robustness* of underlying state replacement techniques by preventing confounding through novel execution models; it handles execution omission errors by analyzing two executions symmetrically; and it substantially improves efficiency by leveraging dual slicing. Evaluation on a set of real world bugs shows that the proposed technique can generate high quality explanations at low cost.

VIII. Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This research is supported in part by the National Science Foundation (NSF) under grants 0917007 and 0845870. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] I. Douven, "Abduction," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed.
[2] J. Klein, "Francis bacon," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed.
[3] P. Menzies, "Counterfactual theories of causation," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed.
[4] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002.
[5] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," in *FSE*, 2011.
[6] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *PLDI*, 2004.
[7] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002.
[8] W. N. Sumner and X. Zhang, "Memory indexing: canonicalizing addresses across executions," in *FSE '10*, 2010.
[9] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan, "Analyzing concurrency bugs using dual slicing," in *ISSTA '10*, 2010.
[10] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *TSE*, vol. 28, no. 2, pp. 183–200, 2002.
[11] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE*, 2005.
[12] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *PLDI*, 2008.

[13] W. N. Sumner and X. Zhang, "Algorithms for automatically computing the causal paths of failures," in *FASE*, 2009.

[14] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," in *PLDI*, 2007, pp. 415–424.

[15] W. N. Sumner, T. Bao, and X. Zhang, "Selecting peers for execution comparison," in *ISSTA*, 2011.

[16] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential slicing: Identifying causal execution differences for security applications," in *IEEE S&P*, 2011.

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

[18] J. Roessler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *ISSTA*, 2012.

[19] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, 1988.

[20] F. Tip, "A survey of program slicing techniques," *J. Prog. Lang.*, vol. 3, no. 3, 1995.

[21] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in *PLDI*, 2011.

[22] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *STTT*, vol. 8, no. 3, pp. 229–247, 2006.

[23] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: an approach for debugging evolving programs," in *FSE*, 2009.

[24] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang, "Golden implementation driven software debugging," in *FSE*, 2010.

[25] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *ECOOP*, 2008.