DUAL EXECUTION AND ITS APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Dohyeong Kim

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Dongyan Xu

Department of Computer Science

Dr. Ninghui Li

Department of Computer Science

Dr. Roopsha Samanta

Department of Computer Science

Approved by:

Dr. Clifton Bingham

Head of the School Graduate Program

This thesis is dedicated to my family, Sijeong, Ethan, and Lillian. Thank you all for the support.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Kim, Dohyeong PhD, Purdue University, May 2020. Dual Execution And Its Applications. Major Professor: Xiangyu Zhang.

Execution comparison techniques compare multiple executions from the same program or highly similar programs to identify state differences including control flow differences and variable value differences. Execution comparison has been used to debug sequential, concurrent, and regression failures, by reasoning about the causality between execution differences and input differences [1], thread scheduling differences [2], and syntactic differences among program versions [3–5], respectively.

However, execution comparison techniques have several limitations. First, executions may have benign differences, which are not related to the behavior differences that the user can observe. Second, huge storage spaces are required to record two independent executions. Third, the techniques can only compare executions from the same or similar programs.

In this dissertation, we present an execution comparison technique that (1) removes benign differences, (2) requires less space, and (3) can compare two different programs implementing similar algorithms. Also, we present that the execution comparison technique can be used in identifying and extracting a functional component out of a binary.

First, we present a dual execution engine that executes multiple executions at the same time and only introduces the desired differences. Also, the engine compares the executions on-the-fly and stores the differences only. Second, we present a technique to compare two programs written by two different programmers. Especially we will show that this technique can compare the buggy program from a student and the correct from the instructor and can reason about the errors.

1 INTRODUCTION

Execution comparison techniques compare multiple executions from the same program or highly similar programs to identify state differences including control flow differences and variable value differences. Execution comparison has been used to debug sequential, concurrent, and regression failures, by reasoning about the causality between execution differences and input differences [1], thread scheduling differences [2], and syntactic differences among program versions [3–5], respectively. Execution comparison is also used in malware behavior analysis [6]. Advanced malware often has logic to decide whether or not to activate its payload depending on the environment such as the platform, the running applications on the victim machine, the current date and time, and the presence of debuggers or virtual machines. To understand the activation logic and the payload, malware is executed in various environments and the resulting executions are compared. Software diversification creates executables with different structures, e.g. different stack layouts [7]. At runtime, multiple such versions are executed simultaneously. The executions are compared to detect intrusion because an exploit to one version often crashes others.

However, execution comparison techniques have several limitations. First, there are benign execution differences that are not related to the different behavior that users intend. For example, if a program uses a random value that is not related to a bug, even though the program may behave differently depending on the random value, but it is not important. Also in programs having interactive user interfaces, execution may differ depending on when the user presses a key or clicks a button.

Second, there is huge space overhead in execution comparison. To compare multiple executions, we need to store the entire executions beforehand. In a fine-grained execution comparison, every data and control dependencies of each executed instruction instance should be recorded. We need tens of bytes for every instruction

instance and it may cost a few gigabytes with a few seconds of execution. This also can cause huge time overhead as well.

Last, traditional execution comparison techniques have limited ability in comparing executions from different programs. In order to compare executions, we should know if two locations or two variables in the executions should be aligned with each other. When comparing executions from the same program, this is not an issue. However, when comparing executions from different programs, there is no perfect solution.

In this dissertation, we will focus on the issues in traditional execution comparison techniques. First, we propose a dual execution system that runs multiple executions at the same time and compares them on-the-fly. Second, we propose an APEX, automated bug explanation system that can compare executions from different implementations of the same problem.

1.1 Thesis Statement

Traditional execution comparison can only compare executions from the same have limitations regarding space overhead and lack of ability to handle executions from different programs. This dissertation shows that a fine-grained execution comparison can be performed on the executions from different programs while avoiding huge space overhead.

1.2 Contributions

The contributions of this dissertation are:

- We propose a dual execution system that runs multiple executions at the same time while feeding the same input as long as needed. The user can provide inputs different only for targeted fields. Thus, this system can solve the problem of unintended differences. Also, this system compares executions on-the-fly and stores only the differences observed. Therefore it does not cause huge space overhead.

- We propose an APEX, automated bug explanation system. This technique compares executions from different programs with symbolic expressions. We will show that this technique can be applied to education and can generate automated feedback for buggy implementations.

1.3 Outlines

This dissertation presents an execution comparison system that can avoid benign differences and space overhead, and propose a technique to align executions from different implementations of the same problem with symbolic expressions. The rest of this dissertation is organized as follows.

- Chapter 3 presents a dual execution engine that runs executions at the same time and performs a fine-grained execution comparison on-the-fly. We present details of the system and the evaluation in this chapter.

- Chapter 4 proposes an automated bug explanation system called APEX which compares executions from different implementations with symbolic expressions.

2 DUALSLICING

2.1 Introduction

Developers have long struggled with the desire to reuse previously implemented features within new code. By reusing an old implementation, developers can avoid creating new bugs and can create easier to maintain programs [8–11]. Implementation reuse can also be crucial when a new program must replicate features within a legacy system, but the specification for the legacy system no longer exists. Driven by this desire to reuse existing implementations, previous research has delved into techniques for both *locating* the implementation of a feature within a body of source code [11–23] and *extracting* that source implementation into a conveniently reusable function [9, 14, 24–28].

These techniques generally assume the availability of the original source code, but in practice the source code itself may no longer be available or may no longer even exist. Indeed, facing the maintenance of legacy programs without source code, DARPA recently called for a solution to this exact problem [29]. For example, some components are provided to developers only in the binary form, and the source of these programs or libraries may not be available due to intellectual property restrictions [30]. In other cases, companies may have existing programs that implement *de facto* specifications, but both the original source code and any documentation of the specifications have been lost over time [29]. Finally, when reverse engineering the behavior of a foreign program, a program from a third party, security researchers sometimes wish to extract certain features, such as encoding/decoding routines [31] or anti-debugger techniques [32] from a foreign program in binary form. Reusing these features from foreign binaries allows the security analysts to gain insight into the behavior of malicious code and potentially develop defensive techniques [31]. In each of these scenarios, a developer needs to

locate and extract the existing implementation of a feature that exists only in binary form within an existing program. Although source code provides rich information about the behavior and structure of a program, much of this information is stripped away when the program is compiled to a binary form.Thus, techniques for locating and extracting features that rely on source code analysis and manipulation no longer apply.

New solutions must be found for both locating and extracting features. Prior work on locating a desired feature includes techniques built on statement coverage information [13] and dynamic slicing techniques [33]. While both of these techniques apply to both source code and binaries, they may both locate features too coarsely, including more of the original implementation than is necessary or desirable. Dynamic slicing techniques compute transitive closures over the dynamic dependence graph of an execution [34–36]. These closures are known to be large in practice [37]. Statement coverage techniques contrast the statements performed within an execution that exhibits a desired feature against those performed in an execution that does not exhibit the feature. The intuition is that statements executed only, or more frequently, within the execution exhibiting the feature should implement the feature itself. We observe and later show that such approaches can be too coarse grained and identify portions of the original implementation that are unnecessary for implementing the feature.

Once the functions implementing a feature have been identified, they may be extracted from the original binary, but extraction alone is insufficient. To reuse the extracted component, we must provide an interface through which it may be invoked, but even state of the art binary analysis tools have difficulty reverse engineering such interfaces. We show that the original interface for a component can also involve complex heap structures, and the parameters that correspond to a feature of interest may be deeply embedded within these heap structures and subject to subtle constraints. Developers should not need to deal with such complexities when reusing an extracted component.

In this paper, we propose a novel approach to locating modular functions that correspond to a desired feature and providing a usable interface for the extracted component. We describe the desired feature through multiple executions that use the feature. The user executes the feature twice with different inputs, and our technique semantically contrasts the executions to discover where the different input values are used and which part of the code produces the desired output from the input. Our technique then isolates the feature using *concretization* to replace the original parameters of the function with values from a real run. Finally, it wraps the original interface of the binary with a new and simpler interface for the developer to invoke and then uses *redirection* to ensure that the parameters of the new interface are used consistently throughout the extracted function.

Our main contributions are highlighted as follows.

- We provide a way to precisely represent a desired feature using multiple executions that exhibit the feature. Providing these executions is intuitive to a user who knows how to use the existing binary.

- We perform a semantic comparison of the provided executions using dual slicing [38]. This allows us to precisely locate the desired feature within the code.

- We propose a technique called *interface casting* that uses *concretization* to isolate desired parameters for a function. It then wraps an extracted binary feature with an adapter and exploits *redirection* to use the parameters of the adapter. This provides the developer a convenient means of invoking the extracted feature.

- We implement and evaluate a prototype of the approach. We apply our technique to 8 applications and extract 10 reusable components from the binaries. We show that even when there are originally no parameters to the extracted functions, our technique still applies.

2.2 Motivating Example

Suppose a developer desires to extract and reuse the email sending feature of `pine`, an email client. Because `pine` has many diverse email features, this *reuser* must first locate the function that contains the desired feature. To reuse the function, they must also uncover the function's interface or prototype. Knowing the interface, they can provide parameters such as the sender address, recipient, subject, and body for a sent email. In this section, we show how to locate the desired function by using dual slicing. We then show how to extract the function into an isolated component with a reusable interface by using concretization and redirection.

2.2.1 Function Location

Before extraction, we must locate the function responsible for sending an email. To find the function, we contrast two different executions of `pine`, each of which sends a different email. We follow the same steps both times, except that the sender addresses, recipients, subjects, and bodies of the sent emails differ. Thus, we choose the same menu items in the same order, and we provide the same sequence of key strokes except for the four parameters of interest. As a result, the two executions follow the same paths through the program except for differences related to the differing user input.

Dual slicing is a technique that contrasts two executions and identifies only those instructions that both behave differently across the two executions and contribute to their different outputs. Intuitively, the user inputs for the two executions of `pine` differ only with respect to the emails sent, so the two executions should mainly differ in the portion of code that is responsible for processing and sending the different emails. Thus, the differences identified by the dual slice should be the behaviors of `pine` that we wish to extract.

Next, our technique find the function that encloses all of the relevant differences between the two executions. Fig. 2.1 shows the part of the dynamic call tree containing the dual slice. Each node represents a function and each arrow represents a caller-callee

relationship between functions. Shaded nodes represent those functions containing the relevant instructions within the dual slice. The topmost shaded function is `call_mailer()`, and it transitively calls all other shaded functions. Since the shaded functions are necessary for the mail sending component, we identify `call_mailer()` and its callees as the components of **`pine`** to extract.

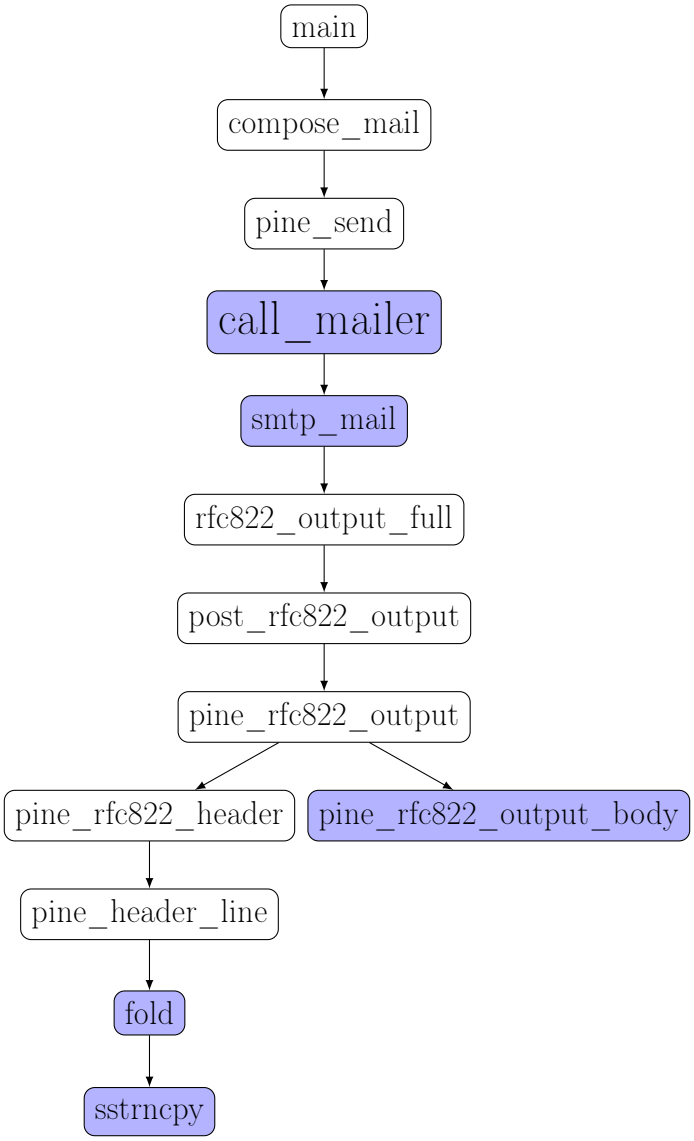


Figure 2.1.: Dual slice of the mail sending feature in **`pine`**. Shaded nodes show the dual slice within the call tree

2.2.2 Interface Casting

We must next extract `call_mailer()` and provide it with a usable interface. The new interface is particularly important because the original interface is complex and does not match the expectations of reuser.

```
struct BODY {
  PARTTEXT contents;     /* body part contents */
  union {   /* different ways of accessing contents */
    PART *part;   /* body part list */
    MESSAGE *msg; /* body encapsulated message */
  } nested;
  ...
};
struct PART {
  BODY body;  /* body information for this part */
  PART *next; /* next body part */
};
struct MESSAGE {
  BODY *body; /* message body */
  PARTTEXT text /* body text */
  ...
};
struct PARTTEXT {
  unsigned long offset;  /* offset from body origin */
  struct {
    unsigned char *data; /* text */
    unsigned long size;  /* size of text in octets */
  } text;
}
int
call_mailer(METAENV *header, BODY *body,
  char **alt_smtp_servers, int flags,
  void (*bigresult_f)(char *, int),
  void (*pipecb_f)(PIPE_S *, int, void *))
```

Figure 2.2.: Interface of `call_mailer()`

Suppose that we tried to invoke the extracted function directly. Fig. 2.2 presents the original interface of `call_mailer()`. The function has 6 parameters. The first and

the second arguments are pointers to internal data structures, and we would need to reverse engineer those data structures to reuse the original interface. In particular, the body of an email is stored in `body->contents.text.data` and the size of the body is stored in `body->contents.text.size`. To specify the body, we would first need to allocate memory regions for the `BODY` structure and its child data structures, e.g. `PART` and `MESSAGE`, and specify correct values for both `data` and `size`. This also requires understanding the semantic relationship between `data` and `size`. To reuse `call_mailer()`, we would further need to correctly initialize the entire data structure and identify the semantics of each field *even if the field is unrelated to the four parameters we want to provide*. In fact, there is even more complexity, as email contents may be specified in two different ways: one through the `contents` field of `BODY` and the other through a further nested field of `BODY`. Expecting the reuser to manage this complexity on their own is unrealistic.

To provide a usable interface for the function, we must simplify away the unnecessary parameters and introduce new parameters matching the reuser's intentions. We call this process *interface casting*. To simplify existing parameters, our technique first statically concretizes those values generated outside `call_mailer()` and used inside `call_mailer()`. Thus, if `call_mailer()` is invoked by the reuser, the parameter `bigresult_f` does not actually take a variable argument. Instead, we provide it a concrete value observed in one of the original executions. We concretize not only the values of all function parameters but any memory values defined outside `call_mailer()`. By concretizing all of the direct and indirect inputs, we hermetically seal the function of interest. That is, by providing concrete values for all inputs of a function, we ensure that it behaves the same way every time.

To provide the parameters desired by the reuser, our technique then relaxes this seal to allow only the chosen parameters to again affect the function's behavior. We redirect accesses of the original inputs to use memory locations for the new parameters provided by an interface that we construct. Since we already concretize the parameters and memory values, the location for a parameter is statically fixed. For example, the

value of the parameter `body` may be concretized to `0x0408CC00`, and the subject of an email may be concretized to `0x0409DB00`. Thus, the program will always read the email subject from the same location in memory, and we can redirect accesses of that memory location to use a new memory location that contains a new subject string.

Before we can relax and redirect accesses of inputs, we must first allow the reuser to determine just which data should be parameters for the extracted function. Once again, the reuser can use dual slicing to provide this information. Recall that in the function identification phase we contrasted two executions with all desired parameters changed to identify the code to extract. In contrast, to identify the instructions that read each input, we need only change one input at a time. This way the dual slice between the original execution and the execution with one differing input will capture only those instructions processing the changed input.

Once we identify all desired parameters of `call_mailer()` and create a new interface using concretization and redirection, we can simply extract the function from its original binary by using binary rewriting tools [39]. The new interface we provide acts as a wrapper that invokes this binary function, allowing the reuser to call it like any other library function.

2.3 The Reuse Process

In this section we discuss the details of reusing functions from binary code. We present algorithms for both locating the function that contains a feature through dual slicing and for providing a reusable interface through concretization and redirection.

2.3.1 Component Location

We first present background information on dual slicing to clarify details of our approach for locating components within a binary. We then explore our algorithms for locating components and why they can localize a component to a more concise portion of code than existing techniques.

Dual slicing. Dual slicing is a slicing technique that contrasts two executions and produces a slice containing only those differences between the two executions that are responsible for some observably different behavior [38]. Alg. 1 presents the core algorithm. Given a slice criterion (e_1, e_2) that identifies some output differences across the two executions, the algorithm computes a set of dynamic dependencies from both executions. (e_1, e_2) denotes that two execution points e_1 and e_2, in the first and the second executions respectively, align or correspond across the executions [40]. $(e_1, \perp)$ denotes that there is no execution point in the second execution that aligns with e_1 in the first execution.

The algorithm first ensures that the slice criterion exists in the first execution. Lines 2-6 process data dependencies at the slice criterion. Here, $\{(e_1, e_2) \longrightarrow (e_1', e_2')\}$ denotes that e_1 has a data dependence upon e_1' in the first execution, e_2 has a data dependence upon e_2' in the second execution, and e_1' aligns with e_2'. e_2' can be $\perp$ when e_1' does not align with any point in the second execution or e_2 is not data dependent on the alignment of e_1'. On line 3, if the data dependence exists only in the first execution or if the values of two data dependencies differ, the data dependence is added to the dual slice. The algorithm proceeds to include the dual slice from (e_1', e_2') recursively, similar to traditional dynamic slicing. Lines 7-10 process the control dependence of the slice criterion, denoted as $\Longrightarrow$. Similar to the data dependence, if the control dependence exists only in the first execution or the branch outcomes differ, the control dependence and the recursive dual slice of the control dependence are added to the dual slice. So far, the algorithm considers only data dependencies and control dependencies when e_1 is not null. Lines 12-14 compute the dual slice when e_2 is not null.

Component location using dual slicing. To use dual slicing to identify the component that corresponds to the desired feature, we use two executions that each exercise the desired feature but that use different inputs. For example, in the `pine` case study, we send an email in both executions, but the emails have different recipients, subjects, and bodies. The resulting dual slice contains only those instructions that

Algorithm 1 Dual slicing

Input: e_1, e_2- slice criteria

Output:$\mathcal{D}$- the dual slice, a set of deps in either execution

```
1:  procedure DUALSLICE(e1, e2)
2:      if e1 ≠ ⊥ then
3:          for all data dep dd ← {(e1, e2) ⟶ (e1', e2')} do
4:              if e2' ≡ ⊥ or values at e1' and e2' differ then
5:                  D ← D ∪ dd ∪ DUALSLICE(e1', e2')
6:          control dep cd ← {(e1, e2) ⟹ (e1', e2')}
7:          if e2' ≡ ⊥ or branch outcomes at e1' and e2' differ then
8:              D ← D ∪ cd ∪ DUALSLICE(e1', e2')
9:      if e2 ≠ ⊥ then
10:         /* operations symmetric to when e1 ≠ ⊥ */
11:     return D
```

process the input because all the execution differences originate from the differing inputs. The slice also includes only instructions that help produce the desired output because slicing excludes instructions unrelated to the output.

Alg. 2 presents the component identification algorithm. The algorithm requires two executions, E_1 and E_2, which exercise the same feature but with different inputs. Lines 1 and 2 choose the output corresponding to the desired feature as the slice criterion. In the `pine` example, we use the network packet containing the composed email as the slice criterion. Line 3 computes the dual slice, and line 4 trims off a prefix of the slice that only moves the arguments around without using them for any computation. Line 5 locates the function that contains this trimmed dual slice. It chooses the closest common ancestor function in the dynamic call tree of those functions whose instructions reside in this dual slice. Line 6 further selects this common ancestor as well as all functions that it transitively called in E_1 and E_2 as

targets of extraction. In other words, the identified components comprise nodes of the dynamic call tree with the identified common ancestor function as their root.

Algorithm 2 Component location

Input: a pair of executions E_1 and E_2 with both exercising the target functionality but with different inputs.

1: **procedure** IDENTIFICATION(E_1,E_2)
2: (O_1,O_2) = outputs corresponding to the desired feature in E_1 and E_2, resp.
3: (e_1, e_2) = execution points that emit O_1 and O_2, resp.
4: $\overline{ds}_e$ = DUALSLICE(e_1, e_2)
5: $\overline{ts}_e$ = TRIM($\overline{ds}_e$)
6: *func* = the modular function that encloses $\overline{ts}_e$
7: *extract* = *func* and all user functions directly/indirectly called by *func*
8: **return** *extract*

Suppose that we wish to identify the function containing the 'email sending' functionality of the sample program in Fig. 2.3 that models `pine`. `load_config()` first initializes global variables that will be used in `pine_send()`. `menu()` waits for an input from a user with timer of 1 second. If the timer expires, `menu()` performs background tasks. If the user instead selects the send menu option, `pine_send()` calls `editor()` to edit the recipient, subject, and body of an email. Later `call_mailer()` composes an email with the information from `editor()` and sends it to an SMTP server. In this example, `call_mailer()` has the email sending functionality since `editor()` only stores the user input into a buffer and does not apply any calculation or transformation to the given inputs.

To get two execution traces for dual slicing, we run the program twice with different recipients, subjects, and body texts. We run the program in exactly the same way the second time except for those inputs. Those three parameters are all that we want our extracted component to require, and we want to use the same values for other configurations such as the SMTP server address and sender address.

```
1  main() {
2    load_config();
3    menu();
4  }
5  load_config() {
6    smtp_server = x.x.x.x;
7  }
8  menu() {
9    t = timer(1);
10   while (true) {
11     c = select(stdin, t);
12     if (c == t) // timer expired
13       do_something();
14     else if (c == stdin) {
15       command = read(stdin);
16       if (command == SEND) {
17         pine_send();
18         log("send mail");
19       }
20       else if (command == CANCEL)
21         continue;
22     }
23   }
24 }
25 pine_send() {
26   ENVELOPE env;
27   BODY body;
28   editor(&env, &body);
29   call_mailer(smtp_server, env, body);
30 }
31 editor(ENVELOPE* env, BODY* body) {
32   env->recipient = read();
33   env->subject = read();
34   body->text = read();
35 }
36 call_mailer(char* server, ENVELOPE* e, BODY* b) {
37   s = connect(server);
38   send_to(s, compose_mail(e->recipient,
39                           e->subject, b->text));
40 }
```

Figure 2.3.: Simplified program modeling 'email sending' in **pine**

Fig. 2.4a and Fig. 2.4b present the resulting traces. Line 43 is the slice criterion because it sends the packet containing the email to the SMTP server. The dual slice includes that line because the sent packets differ in the two executions. Line 43 further depends on lines 42, 36, 37, and 38. Line 42 uses the same value of `smtp_server` in both executions, so the dual slice excludes it. Because lines 36, 37, and 38 produce value differences, the dual slice includes them. Also, line 43 is (directly or transitively) control dependent upon lines 12, 14, 16, and 18, but those lines do not reflect differences across the two executions, so the dual slice excludes them.

The dual slice shows that lines 36, 37, 38, and 43 are important for the mail sending functionality. Note, however, that lines 36, 37, and 38 simply copy the input to a buffer. Because they only move the input around and do not make decisions or perform computations with it, these lines form an *irrelevant prefix* of the desired behavior. They reflect preparatory bookkeeping work rather than behavior of the desired component itself. We can thus omit them entirely and still locate the functions containing the behavior we wish to extract. The TRIM function removes such instructions from the front of the dual slice up until the first decisions or computations with inputs that differ across the executions. In practice, this localizes the component to a smaller portion of code. In our example, the only remaining instruction is line 43, so the technique identifies that the email sending feature is located within `call_mailer()`.

(a) execution 1

(b) execution 2

Figure 2.4.: Dual slice of the simplified **pine** example from Fig. 2.3

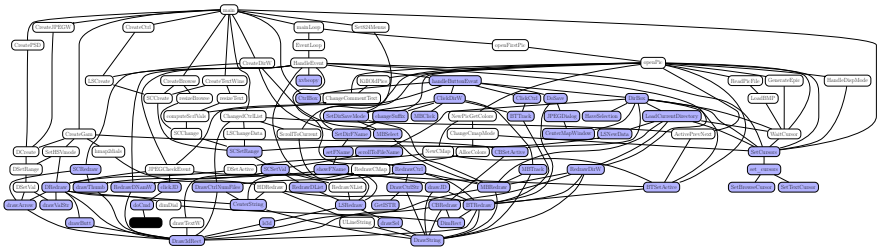
Coverage based approaches. Prior work on feature location computed the difference in statement coverage between two executions: one sending an email and the other not sending the email [13]. This coverage based comparison can identify more functions than we desire because the reuser cannot control the program's behavior at a fine-grained level. That is, a small behavioral difference to the user may correspond to many differences in terms of which functions a program executes, only a few of which may be interesting.

Consider `xv`, an image viewer that can convert one image format to another. Suppose our target functionality is converting a BMP format image to JPEG format. To compute the coverage difference, we load the same file in both executions. We convert the file into JPEG format in one execution but cancel the conversion in the other. Fig. 2.5a presents the coverage comparison results. The results show a large call graph with many functions related to processing the user interface and handling user input such as mouse clicks in addition to the important function, `writeJPEG()`. Furthermore, the approach misses the function `LoadBMP()`, which is responsible for loading a BMP image. In contrast, dual slicing computes a concise set of functions for the conversion and identifies `LoadBMP()` as well. Fig. 2.5b shows the dual slice, which highlights only the important functions: `writeJPEG()` and `LoadBMP()`. We later discuss pruning the extracted component to contain only these two functions and their callees.

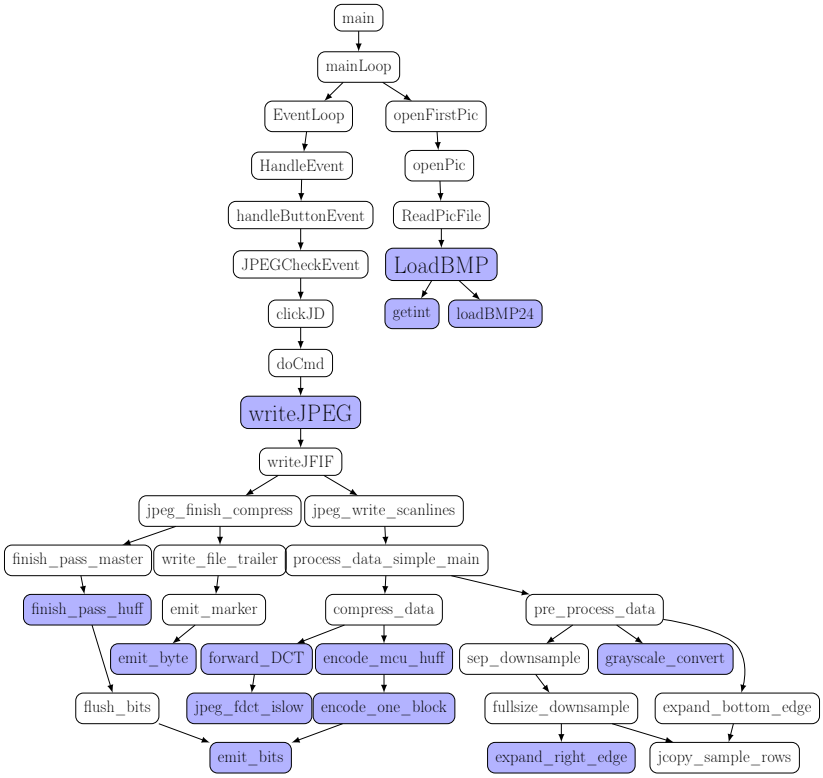
The simple coverage difference includes many non-essential functions because the reuser cannot control every detail of program behavior by enabling and disabling the target feature. Hence, in this paper we use dual slicing [38] to focus more concisely on the interesting differences.

2.3.2 Interface Casting

Once we have located the function the reuser wishes to extract, we must take the potentially complicated interface of that original function and compose a simpler alternative interface with only the reuser's desired parameters. Our approach to this



(a) Coverage difference between converting BMP to JPEG and not converting BMP to JPEG. Shaded nodes are the functions executed only when converting BMP to JPEG and the lone black node is `writeJPEG`, which is important for the conversion



(b) The result of dual slicing. The call graph is concise and highlights only two functions, `LoadBMP()` and `writeJPEG()`

Figure 2.5.: Call graph from `xv` case study

problem is to first concretizing all of the values that feed into the selected function to hermetically seal and isolate the function's behavior. This makes the function behave the same way every time it executes. Our technique then relaxes this seal for only the reuser's desired inputs by redirecting accesses of the original inputs so that they instead access inputs of a freshly constructed interface.

Alg. 3 presents an overview of the interface casting process. The algorithm takes three parameters: (1) the code to extract, a result of the component identification algorithm, (2) an execution E that exercises the desired feature, and (3) one additional execution for each parameter we wish to specify. In the `pine` case study, if we wished to specify the recipient, subject, and body of an email, we would need three additional executions. One would send an email with the same subject and body as execution E but with a different recipient. Another would send the email with a different subject. The last would send the email with a different body. These additional executions identify those instructions that access each of the different specified parameters.

Line 1 computes the set of instruction instances with *external* dependencies. If an instruction reads a value from memory that was written outside the modular component, it is an external dependence. Since the selected component does not create values for external dependencies, they must be provided for the component to execute correctly. Line 2 concretizes all external memory dependencies to seal the behavior of the function. This replaces values of accesses with those observed in E. Extracting the function at this point would create a new function with no arguments that behaves as in E every time it is called. The loop in lines 3-8 considers each parameter specified by the reuser and identifies all instructions with external dependencies upon each parameter. The loop redirects those accesses to instead use new memory locations that hold the values of the parameters within a wrapper function that matches the reuser's demands.

Concretization. To seal the function and remove undesired inputs from the interface, we concretize the values of those inputs by monitoring memory accesses. For example, in Fig. 2.2, `call_mailer()` has a variable `alt_smtp_servers` that holds alternative SMTP

Algorithm 3 Interface casting

Input: *extract* denotes the code to extract, which is identified in the previous section; an execution E exercising the target functionality; a list S of pairs (E_i, T_i) with E_i the same as E except that E_i has a different value for the ith input (with type T_i) intended by the user.

```
1: procedure INTERFACECASTING(extract,E,S)
2:     ext_dep= instruction instances in E that are part of extract and have external dependencies
3:     CONCRETIZE(extract, ext_dep)                ▷ Seal off all external dependencies
4:     for each (E_i, T_i) ∈ S do                  ▷ Patch to allow reuser specified inputs
5:         (e, e_i)= instruction instances emitting the feature related output in E and E_i, resp.
6:         diff= E's instruction instances in DUALSLICE(e, e_i)
7:         if = instruction instances in diff ∩ ext_dep
8:         REDIRECT(extract, if, T_i)
```

server addresses. We do not want the function we extract to expose this complex behavior to the reuser. To provide an interface without this parameter, we concretize the value of the parameter, so `alt_smtp_servers` always holds the same value in the extracted version of the function. Note that if the reuser intends the SMTP server to be an input of the extracted component, he/she could simply provide an additional execution that differs from the original execution only at the SMTP server address.

Alg. 4 explains the concretization process. Lines 1-5 process the instructions with external dependencies. By the definition of an external dependence, we can assume that the instruction i will have the form `MOV` r_2`, [`r_1`]` because it reads external memory. Lines 3-4 replace the original instruction with (1) a guard to see if the dynamic instance of the instruction uses external memory and (2) new `MOV` instructions to redirect the memory access to a saved value if so. Lines 6-13 process the instructions

that write to external memory. In other words, such an instruction writes a value to some location in memory allocated outside the identified function in the original execution E, implying that the address is invalid in the extracted binary. Thus, we first map the observed memory address of the access into the address of a new variable that we create within the data section of the extracted binary on lines 9-10. Line 12, similar to the read case, replaces the instruction to use this new mapped address instead of the original one.

Redirection. In order to redirect parameters, we first identify the parameters using dual slicing. The approach is similar to the one used for locating the desired function. For each parameter, we use two inputs that differ only with respect to that parameter. For example, we may use two inputs that have different recipients to identify the instructions responsible for the recipient parameter.

After our technique identifies these parameter providing instructions, it redirects the memory accesses in the instructions to new locations. When an instruction reads a parameter from memory, it instead redirects the memory access to a new variable or buffer prepared to hold the parameter.

Alg. 5 presents the redirection algorithm. Line 1 adds a new variable to the data section of the binary. This new variable will hold the input for the extracted version of the function, so accesses of the original data must be redirected to this new variable. We break the inputs down into two different categories during the process: (1) *scalar* variables, which are always accessed through their starting address, and (2) *buffer* variables, which have many internal addresses that may be accessed independently. Lines 2-7 process scalar variables. On line 3, the algorithm iterates over the instructions $\overline{if}_i$ discovered by dual slicing with two inputs that identify one parameter. On line 6, it replaces the instruction. If the instruction reads from the location that we identified as the ith parameter, it is redirected to instead read the new variable prepared for the ith parameter on line 1. External dependencies through registers are handled similarly and thus elided. Lines 8-14 process a variable holding a buffer that may be read from at any consecutive memory locations within the buffer.

Algorithm 4 Patch the extracted code for external dependencies through concretization

Input: *extract* denotes the code to extract; $\overline{ext\_dep}$ denoting instruction instances in E that are part of *extract* and have external dependence

1: **procedure** CONCRETIZE(*extract*, $\overline{ext\_dep}$)

▷ concretize reads of external dependencies

2: **for all** unique instruction i in $\overline{ext\_dep}$ **do**

3: **let** i be '`MOV` r_2`,[`r_1`]`'

4: **let** $T = \overline{\{addr \mapsto val\}}$ be a map from addresses to the values of i's external dependencies

5: replace i with the following:

```
if r1 ∈ T:
  MOV r2,T[r1]
else: MOV r2,[r1]
```

▷ patch instructions writing through addresses derived from external dependencies

6: **for all** instruction i in *extract* that may write to an address that is directly/indirectly computed from an external dependence in E **do**

7: **let** i be '`MOV [`r_2`],`r_1'

8: **for all** external address a that i has written to **do**

9: add an entry x_a to the data section

10: $map[a] = x_a$

11: replace i with the following:

```
if r2 ∈ map:
  MOV [map[r2]],r1
else: MOV [r2],r1
```

Many strings provided by the user fall into this category, including the body `data` parameter in the **pine** case study. Similar to scalar variables, the algorithm iterates over and replaces the discovered instructions. If the instruction reads from a memory address corresponding to the ith parameter, it is redirected to read from the new location. Because the instruction reads from a buffer, we must guard the redirection by checking whether the address lies between the lowest address observed for the ith parameter and the size of the new parameter.

Next we use the **pine** example to illustrate concretization and redirection. Fig. 2.6 presents a portion of code from **pine** that reads the email subject along with the corresponding binary code and the rewritten binary after concretization and redirection. On line 3, `call_mailer` reads `header->env`, on line 14 `smtp_mail` reads `env->subject`, and on line 23 `rfc822_output_...()` reads a subject from a buffer where `subject` is pointing.

To find the instructions that read the email subject in this example, we slice two executions with different subjects. The resulting dual slice includes only line 23 because that instruction reads the subject, and the value changes when the user input changes. Although lines 3 and 14 are not in the dual slice, the values they use come from outside `call_mailer()`, causing external dependencies. Hence, our technique automatically concretizes accesses in lines 3 and 14 and redirects the one in line 23, as shown in lines 5 (for `header`), 7 (for `header->env`), 16 (`env->subject`), and 25 (`subject`).

The key idea of concretization and redirection is that the concretized values are used only as keys for redirection and never actually dereferenced. Concretized pointer values ensure the same original buffer address is accessed and the accesses can simply be redirected. For example, we concretize the instruction on line 4 that reads the `header` parameter. On line 6, **pine** reads `header->env` through `header`. Since the `ebx` register is already concretized on line 4, `ecx` is also concretized on line 6. Note, however, that one instruction may execute many times, and some instances of the instruction should be concretized while other instances should not, we ensure that the instruction performs the original behavior as necessary through the else branch of the instrumentation.

```
1  call_mailer(METAENV* header, ...) {
2    ...
3    smtp_mail(..., header->env, ...);
4  /* 633b98: MOV ebx, [ebp + 0x8]
5     → MOV ebx, 0x7ffee00
6     633ba0: MOV ecx, [ebx]
7     → if ebx == 0x7ffee00:
8           MOV ecx, 0x7ffef40
9        else: MOV ecx, [ebx] */
10   ...
11 }
12 smtp_mail(..., ENVELOPE* env, ...) {
13   ...
14   rfc822_output_header_line(..., env->subject);
15 /* 642182: MOV eax, [ecx + 8]
16    → if ecx + 8 == 0x7ffab40:
17          MOV eax, 0x7fff080
18       else: MOV eax, [ecx + 8] */
19   ...
20 }
21 rfc822_output_header_line(..., char * subject) {
22   ...
23   while(n-- > 0 && (**d = *subject++) != '\0')
24 /* 664fc9: MOV eax, [edx]
25    → if 0x7fff080 ≤ edx < 0x7fff080 + 10:
26          MOV eax, nSubject[edx-0x7fff080]
27       else: MOV eax, [edx] */
28   ...
29 }
```

Figure 2.6.: Source code of **pine** reading subject from data structure

Algorithm 5 Redirect instructions related to the ith input.

Input: *extract* denotes the code to extract; $\overline{if}_i$ the instructions load the ith input; T_i the type of the ith input

1: **procedure** REDIRECT(*extract*, $\overline{if}_i$, T_i)
2: add a global variable v_i of type T_i to the data section
3: **if** T_i is a scalar type **then**
4: **for all** unique instruction x **in** $\overline{if}_i$ **do**
5: **let** x be '`MOV` r_2`,[`r_1`]`'
6: **let** *addr* be the address accessed by x in $\overline{if}_i$
7: replace x with the following:

```
if r1 == addr:
  r1 = &xi
  MOV r2,[r1]
  r1 = addr
else: MOV r2,[r1]
```

8: **else if** T_i is a buffer type **then**
9: **for all** unique buffer access instruction x in $\overline{if}_i$ **do**
▷ x must be an instruction repetitively executed to access a buffer
10: **let** x be '`MOV` r_2`,[`r_1`]`'
11: **let** *addr* be the *lowest* address accessed by x in $\overline{if}_i$
12: replace x with the following:

```
if addr ≤ r1 < addr + Ti.size:
  t = r1
  r1 = &xi + (r1 - addr)
  MOV r2,[r1]
  r1 = t
else: MOV r2,[r1]
```

2.4 Practical Challenges

Nondeterminism. In the previous section, we claimed that the dual slice presents only differences originating from input differences. However, when a program is

non-deterministic, there can also be differences caused by that non-determinism. For example, Fig. 2.7 shows that two executions of **pine** that send an email can have differences as a result of non-determinism within an event handling loop controlled by a timer. In the first execution, we select the send command of the menu before the timer expires in iteration A, but in the second execution, we select the send command after the first timeout in iteration B and before a second timeout in iteration C. Thus, lines 16-20 in the first execution do not align with any lines in the second execution. The resulting dual slice includes lines 16-18 because line 19 control depends upon lines 16 and 18, and line 18 data depends upon line 17. This is not a desired result because lines 16-18 also execute in iteration C of the second execution, and these lines show no value differences.

To address this issue, our technique identifies possible non-determinism through a calibration phase. We execute the program twice with the same input. Differences between the two executions show that the program has non-deterministic behavior, and specific differences indicate where non-determinism occurs. When an instruction has a value difference across the executions, i.e. their occurrences in the two executions align but have different values, the value difference originates from non-determinism and the instruction can be ignored during the component location process.

In contrast, control flow differences, i.e. unaligned instruction instances, usually arise from non-determinism in event handling loops. In our **pine** example, the send menu item may be selected in either the first or second iteration of a loop depending on the timer. Thus, our technique first finds the loop containing the non-deterministic behavior through calibration. In our example, the `while` loop starting on line 12 is identified as the non-deterministic event handling loop.

During component identification, when aligning executions with *different* inputs, our technique does not simply align each iteration of a non-deterministic loop in order, but rather based on edit-distance [41]. The technique finds which alignment of iterations in both executions yields the fewest misaligned instructions. If multiple

iterations align equally well, the earliest is selected. In Fig. 2.7, our technique aligns iterations A of the first execution and C of the second.

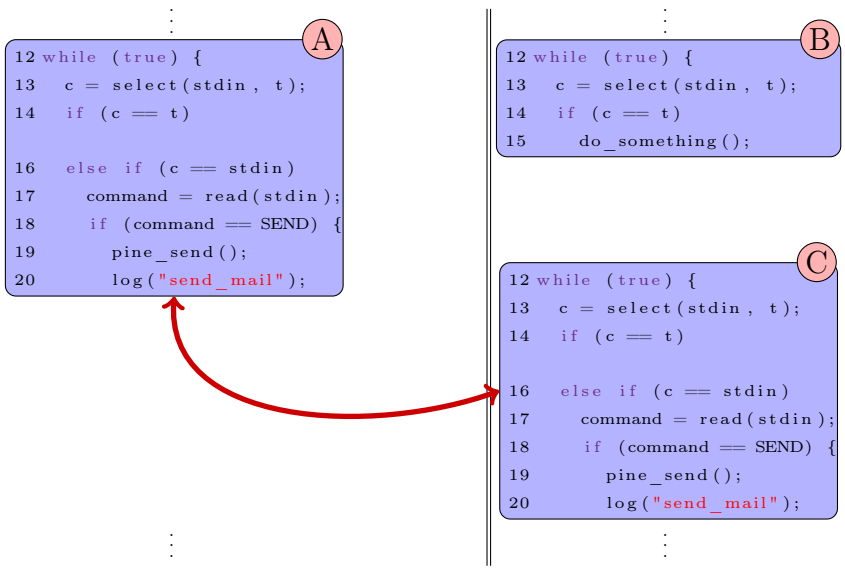


Figure 2.7.: Two executions with nondeterminism. Iteration A should align with iteration C

Locating Multiple Components. In some cases, the component containing the dual slice includes most of the binary. For these cases, our technique supports identifying *multiple* modular functions to extract instead of just one function, in order to reduce the size of the component. Recall, for example, the `xv` case in Fig. 2.5b. Locating a single component will extract everything called by the function `mainLoop()`, which is undesirable. Instead, our technique can extract only `writeJPEG()` and `LoadBMP()`, which precisely capture exactly the behavior of interest.

When extracting multiple functions, the data flow between the functions must be property connected. For example, `writeJPEG()` must use image data generated by `LoadBMP()`. The concretization and redirection process handles this. To simplify our discussion, assume two functions A and B are extracted with A writing to a chunk of memory m, which is later read by B. The memory is allocated outside both A and B. During concretization, instructions writing values to m (in A) are replaced with writes

to a new and valid memory location. Instructions reading m (in B) are redirected to the new memory. If data flow between A and B goes through another function C, e.g. C modifies the values between A and B, C will be included in the dual slice as well, and thus the aforementioned process still applies.

Concretizing Other Resources. During concretization, accesses to external memory get replaced with valid accesses to freshly created variables. However, not all external dependencies may be on memory. For example, the extracted component may depend upon a file handle that is not affected by the reuser's chosen parameters. In such cases, we must identify the external dependencies on other necessary resources and safely acquire them as well. Pragmatically, our present implementation handles dependencies on external files and sockets, but it can be extended to other resources once they have been identified.

2.5 Experiments and results

We have implemented a prototype of our system using Pin [42] for tracing, while the dual slicing is written in C. We use Bistro [39] for extracting the desired components as well as for binary rewriting to perform concretization and redirection. Note that Bistro is a robust binary transformation tool. It can safely extract portions from or patch arbitrary binaries by correcting internal references in the binary like indirect jump and call targets. Using our prototype, we were able to extract 10 components of interest from 7 real world programs into object files. We were further able to link with and invoke those components from new programs that reused the extracted behavior.

2.5.1 Observations

Table 2.1 presents the 10 components we examined in our study. Although we used binaries with debugging information to clarify the dual slicing results during experimentation, our technique does not rely on any debugging symbols or information and can be applied to stripped binaries.

Table 2.1.: Extraction results

Program	Size	Feature	Cov	Cov F. Size	DS	F. Size	Funcs	C. Size	C. Instrs.	R. Instrs.
alpine	5.4MB	Send mail	17210	556KB	360	350KB	1	10KB	314	4
		Create a directory	5168	508KB	204	100KB	2	4KB	195	2
centerim	2.1MB	Login	21814	414KB	96	80KB	2	90KB	142	2
		Send message	11745	529KB	20	15KB	2	4KB	60	2
murofet	3.9MB	Domain flux	-	-	18	12KB	1	20KB	19	1
mysql	3.1MB	Get a list of databases	1000	116KB	270	39KB	2	5KB	187	1
ncmpc	1.2MB	Add a song to playlist	1245	160KB	65	1KB	1	500B	10	1
smbc	7.5MB	Connect to samba server	6090	407KB	1609	210KB	1	4KB	174	2
		Create a directory	6090	407KB	1609	110KB	1	300KB	354	1
word97	5.3MB	Convert DOC to text	19752	754KB	725	52KB	2	68KB	823	2
xv	2.7MB	Convert BMP to JPEG	6083	346KB	3196	50KB	2	5KB	337	5

We first examine results for locating desired features using the coverage based approach and our dual slicing based approach. 'Cov' is the number of instructions covered by the execution exhibiting the feature and *not* covered by the execution not exhibiting the feature [13]. These locate the feature in the coverage based approach. Extracting the functions located by this approach yields the extracted component size 'Cov F. Size'. In contrast, our approach uses the dual slice to locate the desired component. 'DS' presents the number of instructions in the dual slice. Note that it is usually orders of magnitude smaller than 'Cov'. Extracting the functions identified by the dual slice yields the extracted component size 'F. Size'. This is also smaller than the size of components extracted using coverage based techniques. We note that in some cases, like Murofet, dual slicing can locate the desired component even when there are no coverage differences. Thus, dual slicing can have greater generality than a coverage based approach. Also note that existing coverage based approaches do not inherently support working on binaries or performing function extraction [13].

The 'Funcs' column is the number of the modular functions ultimately discovered with our algorithm. In one half of cases, we identified one function, but in the other half, we identified two modular functions to extract. These numbers do not include the callees of the identified functions, which are also extracted. This indicates that many features require multiple functional components. Extracting such features requires understanding relationships such as the data flow between components. Our technique automates this through concretization as presented in the previous section.

The 'C.Size' and 'C.Instrs' columns list the sizes of concretized memory and the number of concretized instructions respectively. Observe that the size of the concretized memory is relatively high compared to the number of concretized instructions. The `pine` case study in the first row shows that a concretized instruction reads roughly 32 bytes on average. This indicates that some of the concretized instructions are in loops, so they execute multiple times to read additional data. Moreover, the size of the concretized memory indicates that the extracted functions require substantial data and that extracting functions without considering these data is unlikely to work.

This concretized information mostly reflects pointers through data structures and global configuration data, such as the SMTP server in `pine` example. The smaller number of concretized instructions further indicates that the size overhead caused by concretization is low.

The 'R.Instrs' column lists the number of redirected instructions. It shows that a very small number of instructions are responsible for reading the parameters of the extracted function and must be redirected for the desired interface.

For the centerim and smbc cases, extracting the feature required two components. For centerim, the login component does not have practical use. However, the send-message component relies upon the login component. To locate the two centerim components, we leveraged experience using the program. When sending a message with centerim, the program requires a password. From this, we infer that we must log into the message server to send a message. Thus, we use two inputs with different login credentials to locate the login component, and we use another two inputs with different messages to locate the send-message component. However, this knowledge is not always available. For smbc, we use two inputs different both in login information and directory name. From the resulting callgraph, we can locate both login and create-directory components.

2.5.2 Case Study: Murofet Worm

Many worms such as Torpig [43], Conficker [44], and Murofet use a technique called *domain flux*, which generates a list of domain names at runtime to hinder analysis of communication between a worm and the attacker. With domain flux, the defender cannot simply block IP addresses to stop the malicious behavior of the worm. Extracting the proprietary domain flux algorithm helps defend against the worm by predicting malicious domain names in advance. To further clarify the use and practicality of our technique, we have posted a demo of this Murofet case study online [45].

When Murofet executes, it generates a domain name and connects to the domain to check whether another malicious payload exists on the server. If the connection fails, the worm repeats the process until it finds a valid server.

To apply our technique, we need two executions generating domain names with different input. Since the worm does not provide a user interface for the domain flux algorithm, we cannot just change the input values. Moreover, we do not *know* the inputs for the domain flux algorithm. From multiple executions, we observe that the worm produces different domain names each time it executes, so the worm uses different inputs whenever it executes. With two executions that produce different domain names, our technique found the function responsible for generating domain names. We used the generated domain name in the DNS lookup packet as the slice criterion. From the dual slice, our technique found the modular function corresponding to the domain flux process and also found that the input for the process came from the system time by analyzing the data dependencies in the slice.

This case study shows that even when a program has no user interface for providing parameters, our technique can still identify and extract the target feature. It can create an interface for parameters as long as the inputs can change across multiple executions. Once it identifies the modular component, we can also narrow our focus to that component to ease further manual analysis of the feature.

2.5.3 Case Study: Word97

Example. Consider an example in Fig. 3.7. When the user selects the menu item `write message`, the program reads a message and writes it to a file with a name provided earlier. If the file already exists, the program first makes a backup. Suppose the user wants to compare two executions with two different file names `A` and `B`, and there is already a file with name `B`. Fig. 3.8 shows the master and the slave executions. Note that since the file name is different at 2_1, the following syscalls at 10_1, 11_1, 12_1,

and 13_1 in the slave are executed instead of copied, which correctly exercises the intended different semantics. In contrast, the syscalls at 5_1 and 9_1 are copied.

The dynamic call tree containing the dual slice forks into two main branches. The first reads and stores the file content into memory. The second then stores this data in a text file. We thus extracted the common ancestor function within each branch into a new component that can read a DOC file and write it into a specified text file. Not only that, but the extracted component is self-contained and works even across different versions of the Windows platform.

This component also exemplifies the necessity of concretizing additional resources as noted in Sec. 2.4. Word97 creates many handles that are initialized by the kernel and that are used by the underlying components that read and write the DOC files. These external dependencies on kernel objects must be valid for the extracted component to run, so we extended our resource concretization system to handle these kernel-level resources as well to create the kernel objects before we execute the extracted component.

2.5.4 Limitations

Our technique shows that it is feasible in practice to locate and extract complex binary features into reusable software components. However, dynamic analysis imposes some limitations. In particular, we only extract a portion of the full semantics of a function. For example, the accesses observed in the provided executions are used by the technique to determine what portions of memory to concretize and make available to the extracted component. The provided executions may not access the full range of addresses in a global table. In this case, we can only guarantee that we extract the portion of the table observed in the provided executions, and accesses outside this range lead to undefined behavior.

3 DUAL EXECUTION FOR ON THE FLY FINE GRAINED EXECUTION COMPARISON

Execution comparison can be classified into *online* and *offline* techniques. In online comparison, the executions are compared on-the-fly. The intrusion detection technique from Salamat in 2009 is one such online approach [7]. However, such comparison is at the system event level. In particular, the executions of the multiple diversified versions are driven by the same input event sequence and the output event sequences are then compared. This technique assumes all executions consume the same input, which does not hold for debugging, where executions may have different paths and hence consume different sequence of input events. Most execution comparison techniques are offline [2, 5, 6, 46]. They first collect traces and compare them offline. A prominent challenge is hence the space required to store such traces, which can grow to a few GB within a few milliseconds of execution. Since executions are performed independently, many execution differences are caused by non-determinism such as input events being received at different times. These differences are not helpful in understanding functional differences.

Note that many existing logging and replay techniques [47,48] cannot both preclude non-determinism and allow for functional differences. Viennot et al. [49] can handle functional differences but it requires explorations and it cannot be used in online analysis. For UI applications, comparing their executions also requires repeated manual effort.

In this chapter, we propose a novel *dual execution* engine to overcome the aforementioned limitations. It performs on-the-fly instruction level comparison of two executions and only stores traces for parts of executions that are different.

There are several challenges in on-the-fly execution comparison. The executions may diverge because of (1) different inputs, (2) different outcomes from interactions

with the environment, and (3) non-determinism in the program. Our technique allows the executions to differ and consume different inputs. It suppresses the differences caused by interactions with the environment and non-determinism.

In particular, it has three execution modes. Initially, the two executions run in a *coupled* mode, in which they are synchronized and one of them, the *slave*, receives most of its system events from the other, the *master*, instead of from the environment. When the user indicates that different inputs should be provided, the two executions get into the *decoupled* mode, in which they execute independently. After introducing the differences, the user indicates the two executions should be coupled again such that they can share the same future inputs to avoid unnecessary non-determinism. Since the two executions may be at different points when the coupling signal is received, in the *resynchronization* mode, the engine blocks the faster execution until the other one catches up. After that, the two execute in the *coupled* mode again. Since the two executions then have different states, even though they execute in the *coupled* mode, their paths and system event sequences may differ, so our engine detects and handles all these differences.

3.1 Motivation

In this section, we use a real world example to explain why dual execution is preferable in execution comparison and illustrate some of the technical challenges in dual execution.

Suppose we want to identify the functional components that change an email subject and log a sent email in `pine`, an email client. In `pine`, a user can get to the email composition interface through a sequence of menu operations, where he/she can provide the email body, subject, and recipient. He/she can also choose whether a copy of the email will be saved in the `sent_mail` file after it is sent, through the `ctrl-r` hot-key. During the whole process, `pine` periodically pulls incoming emails

from the server through a timer control. While an email is being sent, **pine** shows busy messages on the screen.

To identify the desired components, the user provides two executions. In the first, he/she composes an email and sends it, and a copy of the email is saved by default. The second execution is almost identical, except that the user changes the subject and re-configures **pine** so that a copy is not saved.

We first use an offline technique similar to that of Kim et al. [46], in which the two executions run independently, and we collect instruction level traces for offline comparison. We observe the following problems. *First*, we must repeat the almost identical sequence of user interactions. We must be very careful to not introduce any human error. For example, in the second execution, when we try to type the same email, suppose we mistype a character and use **backspace** to fix it. Although the two emails are identical after the mistake is fixed, the instruction level trace faithfully records the interaction error, which will be identified as a difference during comparison.

```
busy_cue() {
  if (status_message_remaining()) ...
}
status_message_remaining() {
  return display_time - time() + min_time > 0;
}
```

Figure 3.1.: Non-determinism in **pine**

Second, even if we manage to avoid introducing human error, there is substantial low level non-determinism, e.g. from timers, that leads to unnecessary execution differences. Fig. 3.1 shows a code snippet from **pine** that has non-deterministic behavior. `busy_cue()` is a function that shows a busy message on the screen. Before showing a message it checks whether there already is a message by calling `status_message_remaining()` at line 2, which checks whether the current message is shown on screen for at least

a minimum amount of time (line 5). The program behavior is thus dependent on execution timing that is non-deterministic. There are additional such timing related behaviors in `pine`. We show in Section 3.4 that execution differences caused by non-determinism can be as large as half of the overall differences.

Third, we observe that the trace for even one of these executions is over 30GB even though the execution is already very small. It is not surprising given the large number of instructions executed within a second by a modern CPU. Storing and processing such huge trace files is a heavy burden even for modern systems. Note that the two traces are mostly identical.

Next, we use our dual execution engine. Initially, the engine spawns two executions of `pine` at the same time. Then, we only interact with one of the executions, called the *master*. All the interactions between the master and the environment, including user interactions, are relayed to the *slave*. This allows us to avoid repeating the same error-prone user interactions. In addition, the two executions run exactly the same way, without any differences caused by non-determinism. Since only differences are recorded, we also avoid tracing in this phase.

After composing the email, we press a predefined hot-key. Now we can control the master and the slave separately. We provide different subjects to the two executions and set the `save_to_sent_mail` option differently. After that, we couple the two executions again by pressing another hot-key. Once again we only interact with the master to confirm sending the email and terminate the program. In this phase, since the engine aligns the two executions at the instruction level and executes them in lock-steps, instruction level differences are detected on the fly and recorded. The resulting trace file is only less than 90MB.

We note several technical challenges. *First*, our system relays system events in the master to the slave, but we cannot relay every event. For example, if the slave did not execute the `write()` system call but rather simulated it using a relayed result during coupled execution, it could not show any interface and thus we could not provide

different inputs when decoupled. The engine must identify the events that can be relayed.

```
pico() {
  while(...) {
    c = GetKey();
    if (c == empty || time_to_check())
      check_new_mail();
    execute (c);
  }
}
GetKey() {
  if (ReadyForKey(STDIN, timeout))
    return read(STDIN);
  else
    return empty;
}
```

Figure 3.2.: Input handling loop in **pine**

Second, when we indicate our intent to re-couple the executions after providing different inputs, the engine cannot simply resume relaying events because the two executions may be in different stages due to the different inputs. Fig. 3.2 shows a code segment in which **pine** receives user inputs. Lines 2-7 use a loop to handle user inputs. At line 3, it reads a key from the user. At lines 4-5, if a certain amount of time has passed, the program checks whether there is a new email. At line 6, the program processes the keystroke. Therefore, depending on the time we spent providing the different inputs, the two executions will be in the different instances of the loop upon re-coupling. They may even be in different child functions of the `pico()` function. The engine needs to re-synchronize the two executions at the instruction level.

Third, even though the engine manages to re-couple the two executions, they may execute differently due to the input differences. Fig. 3.3 shows another code snippet from **pine**. At line 1, the program executes `pico()` to allow the user to compose the email. It also contains the confirmation menu. Re-coupling happens inside `pico()`.

```
pico();
...
call_mailer();
...
if (fcc)
  write_fcc(sentmail);
```

Figure 3.3.: Send-mail function from **pine**

At line 3, the program calls `call_mailer()`, which constructs the email packet and sends it to the SMTP server. After that at line 5, the program checks `fcc`, which corresponds to the `save_to_sent_mail` option. If it is set, **pine** makes a copy of the sent mail (line 6). Since we set `fcc` differently in the two executions, one execution will execute `write_fcc()`, but the other will not. Note that `write_fcc()` relies on several events such as file open and file write. Therefore in this case, the engine needs to detect such differences and provide the events appropriately.

3.2 Design

Our discussion follows the order of the three execution modes: the *coupled* mode in which the master and the slave share system events; the *decoupled* mode in which they execute independently; and the *resynchronization* mode in which the faster execution is blocked until the slower one catches up, when the user wants to re-couple the two executions.

3.2.1 Coupled Execution Mode

The master and slave are in coupled mode when they start and also after different inputs are provided and the executions are re-synchronized. Note that after differences are introduced, the two executions may take different paths and have different syscalls,

even though they are re-synchronized. Thus we focus on detecting and handling such differences that may prevent the executions from sharing events.

We first present the semantics of the master and slave executions and then discuss the monitor that coordinates their behavior.

$TraceEntry$	$t ::=$	$\langle INSTR, l, v\rangle$
		$\mid \langle SYSCALL, l, sid, P\rangle$
$Label$	$l ::=$	$\{l_1, l_2, l_3, ...\}$
$Value$	$v ::=$	$\{true, false, 0, 1, 2, ...\}$
$SysCallId$	$sid ::=$	$\{1, 2, ...\}$
$SysCallParameters$	$P ::=$	$\overline{v}$

Figure 3.4.: Trace syntax

Master Execution in the Coupled Mode.

In coupled mode, the master tracks each instruction execution and sends a trace entry to the monitor. It performs all syscalls faithfully and sends syscall parameters and outcomes to the monitor, who decides if the syscall outcomes need to be relayed to the slave. The semantics is shown in Table 3.1. We model instructions into two kinds: system calls and others (or *regular instructions*). A regular instruction, with opcode op, takes n operands and produces the result in y. According to rule M-INSTR, the instruction is first executed, then a trace entry is sent to the monitor.

Trace Entry Syntax. As shown in Fig. 3.4, there are two kinds of trace entries, identified by the types $INSTR$ (denoting regular instructions) and $SYSCALL$, respectively. A regular instruction entry consists of the label (or the program counter) of the instruction and the left hand side value of the instruction. A syscall entry consists of the label, the syscall id, and the parameters. □

Table 3.1.: Semantic rules for master execution

Instruction	Action	Rule
regular instruction $y=^{l}$ instr(op, x_1,...x_n)	execute the instruction; `send_trace_entry`($\langle INSTR, l, \sigma[y]^1 \rangle$);	M-INSTR
$y=^{l}$ `syscall`(sid, x_1,...x_n);	`send_trace_entry`($\langle SYSCALL, l, sid, \sigma[x_1], ..., \sigma[x_n] \rangle$); execute the system call; `send_syscall_outcome`($\sigma[y]$);	M-SYSCALL

1. σ stands for the store that maps a variable to a value.

Upon a syscall invocation (rule M-SYSCALL), the master first sends the corresponding trace entry containing the syscall id and the parameters to the monitor. It then executes the syscall and sends the outcome to the monitor.

Slave Execution in the Coupled Mode.

As shown in Fig. 3.2, the slave handles a regular instruction in the same way as the master. For a syscall, the slave sends the trace entry containing the parameters to the monitor, which allows the monitor to decide if the slave should *copy* the syscall outcome from the master or *execute* the syscall. After that, the slave receives the decision from the monitor and acts accordingly. `recv_syscall_outcome()` is a blocking call, preventing the situation in which the slave execution is faster than the master and manages to perform a syscall before the master reaches the corresponding syscall.

Monitor in the Coupled Mode.

The monitor is the most complex component. It is responsible for synchronizing the two executions, comparing their trace entries on the fly, and instructing the slave how to handle its syscalls.

$\boxed{ExecIndex \quad \mathcal{I} ::= \bar{l}}$

$\boxed{\texttt{update\_index} : Label \times ExecIndex \rightarrow ExecIndex}$

$\texttt{update\_index}(l, nil) = l$

$$\texttt{update\_index}(l, \mathcal{I}_{head} \cdot l_{tail}) = \begin{cases} \mathcal{I}_{head} \cdot l_{tail} \cdot l & \texttt{if } l \text{ control dep. on } l_{tail} \\ \texttt{update\_index}(l, \mathcal{I}_{head}) & \texttt{otherwise} \end{cases}$$

$\boxed{\texttt{dyn\_ipdom} : ExecIndex \rightarrow ExecIndex}$

$\texttt{dyn\_ipdom}(\mathcal{I}_{head} \cdot l_{tail}) = \mathcal{I}_{head} \cdot l,$ with l the immediate postdom. of l_{tail}

Figure 3.5.: Execution indexing primitives

Table 3.2.: Semantic rules for slave execution

Instruction	Action	Rule
regular instruction $y =^{l}$ instr(op, x_1,...x_n)	execute the instruction; `send_trace_entry`($\langle INSTR, l, \sigma[y] \rangle$)	S-INSTR
$y=^{l}$ `syscall`(sid, x_1,...x_n)	`send_trace_entry`($\langle SYSCALL, l, sid, \sigma[x_1], ..., \sigma[x_n] \rangle$); t=`recv_syscall_outcome`(); `if` ($t ==$ EXEC) execute the system call; `else` $\sigma[y] = t$;	S-SYSCALL

Background: Execution Indexing. Our execution synchronization is built on *execution indexing* (EI) [50]. For completeness, we briefly introduce the EI technique. It computes a unique identifier for an execution point called an *execution index*, or index. Execution points across multiple runs align when they have the same index. The index of an execution point is analogous to the calling context of the point, but instead of describing the nesting of function calls, the index describes the nesting of the control dependence regions of the execution point. Note that multiple instances of an instruction may share the same calling context (and hence calling contexts cannot be used to distinguish the instances), but they must have different control dependence region nestings.

Algorithm 6 Monitor algorithm for the coupled mode

Input: a pair of executions e_m, e_s

Definition: $\mathcal{I}_{m/s}$ denotes the execution index of the current trace entry from master/slave;

```
1:  procedure MONITOR(e_m, e_s)
2:      while e_m and e_s are not finished do
3:          t_m = recv_trace_entry(e_m)
4:          I_m^p = I_m
5:          I_m = update_index(t_m.l, I_m)
6:          t_s = recv_trace_entry(e_s)
7:          I_s^p = I_s
8:          I_s = update_index(t_s.l, I_s)
9:          if t_m.type==SYSCALL then
10:             y_m = recv_syscall_outcome(e_m)
11:             if policy(t_m.sid)==COPY && t_m.P==t_s.P then
12:                 send_2_slave(y_m)
13:             else
14:                 send_2_slave(EXEC)
```

```
15:     if t_m.l != t_s.l then
16:         /* Lines 16-21 run parallel with lines 22-27*/
17:         while I_m != dyn_ipdom(I_m^p) do
18:             record(t_m, nil)
19:             t_m = recv_trace_entry(e_m)
20:             I_m = update_index(t_m.l, I_m)
21:             if t_m.type==SYSCALL then
22:                 y_m = recv_syscall_outcome(e_m)
23:         while I_s != dyn_ipdom(I_s^p) do
24:             record(nil, t_s)
25:             t_s = recv_trace_entry(e_s)
26:             I_s = update_index(t_m.l, I_s)
27:             if t_m.type==SYSCALL then
28:                 send_2_slave(EXEC)
29:     if t_m.v != t_s.v then
30:         record(t_m, t_s)
```

The syntax of execution indexing and the primitives to compute indices are described in Fig. 3.5. Syntactically, an index is a sequence of labels (or PCs). It is constructed by the `update_index()` primitive, which takes the label of the current execution point and the index of the previous point, produces the new index. According to the rules, if the previous index is *nil*, the resulting index is the label. If the previous index is not *nil*, and if the current label l is control dependent on the tail label l_{tail} of the previous index, indicating l is nesting in the region of l_{tail}, l is appended to the previous index. Otherwise, labels at the tail of the previous index are popped one by one until l finds its control dependence region.

Consider Fig. 3.6. A code snippet is presented in (a) and two executions are presented in (b) and (c). The two executions differ due to some state differences

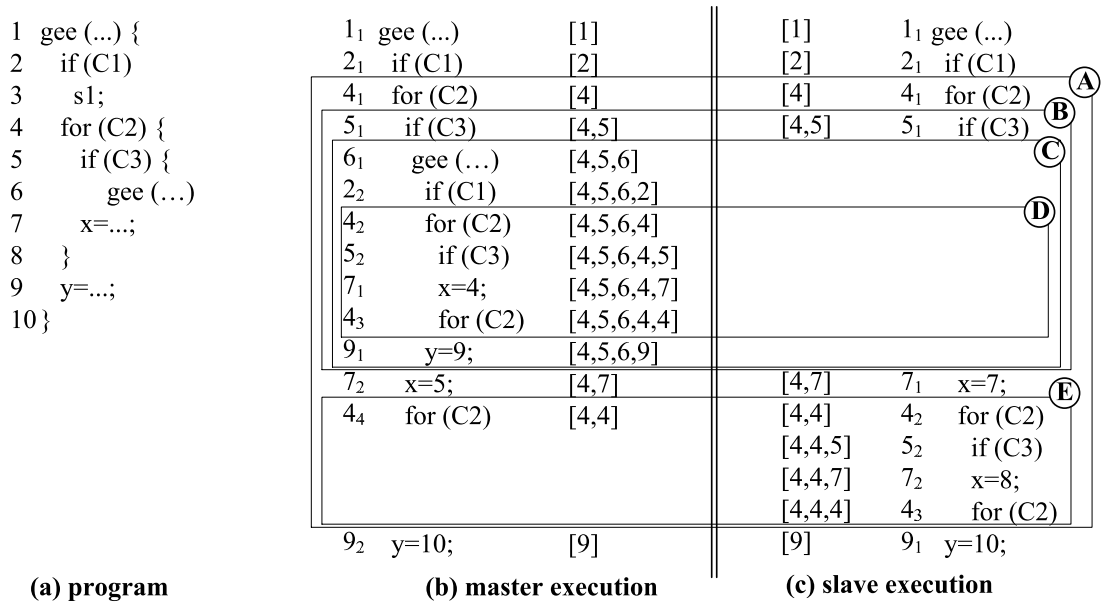


Figure 3.6.: Example for execution indexing and synchronization

introduced earlier. The indices of the execution points are presented in the center for easy comparison. Lets look at the master execution (b). Initially, the index of the first instance of statement 1 $\mathcal{I}(1_1) = [1]$ as its previous index is *nil*; then `update_index`$(2, \mathcal{I}(1_1))$= `update_index`$(2, [1])$= `update_index`$(2, nil)$=$[2]$, because 2 is not control dependent on 1; `update_index`$(5, \mathcal{I}(4_1))$= `update_index`$(5, [4])$=$[4, 5]$ because 5 is control dependent on 4. Intuitively, it means 5 is nesting in the region of 4, denoted by the box Ⓐ. Similarly, $\mathcal{I}(5_2) = [4, 5, 6, 4, 5]$, denoting 5_2 is nesting inside regions Ⓓ, Ⓒ, Ⓑ, and then Ⓐ. Note that our system only maintains the index for the current execution point, similar to maintaining the call stack.

The indices for the slave (c) are similarly computed. By comparing the indices, only $1_1, 2_1, 4_1, 5_1, 7_2, 4_4, 9_2$ in the master have correspondence in the slave. We can see the essence of EI is to align the control dependence regions. For example, the alignment of 5_1 in both executions indicates the Ⓑ regions align. But inside the region, different branch outcomes are taken such that there are no further alignments. □

Synchronizing Master and Slave Executions. The monitor component continuously updates the current indices of the master and the slave based on the trace entries received. The indices allow the monitor to achieve lockstep synchronization of the two executions. Initially, the two executions follow the same path such that they are perfectly synchronized, indicated by identical indices. When a predicate is encountered but its branch outcomes are different in the two runs, the paths start to diverge. As such, the monitor decouples the two executions, allowing them to proceed independently to the immediate post-dominator of the predicate. Note that since the predicates in the two runs share the same immediate post-dominator, they are perfectly synchronized again.

Details are in Algorithm 6, in which a **while** loop continuously processes trace entries until the two executions finish. In the loop, it first receives an entry from the master and updates the current index for the master (lines 2-4). We use $t_m.l$ to represents the label field of the trace entry. Lines 5-7 do the same thing for the slave. In normal cases, the two entries from the two respective runs have the identical labels. But if the previous label is a predicate with different branch outcomes, the pair of entries will have different labels (line 14). In this case, the monitor processes trace entries from the two executions in parallel (lines 16-21 and 22-27), until the *dynamic* immediate post-dominator (IPDOM) of the previous predicate is encountered (line 16 and line 22). Note that the trace entries are recorded (line 17 and line 23) because they denote *control flow differences*. In contrast, in lines 28-29, if the values are different, the two entries (with the same label) are also recorded. Note that the monitor must leverage indices to identify the *dynamic* IPDOM (lines 16 and 22). It cannot simply looks for the next occurrence of the *static* IPDOM, because in the presence of recursive functions, the next occurrence of the static IPDOM may not mean the end of the control dependence region. In Fig. 3.5, `dyn_ipdom`($\mathcal{I}$) computes the dynamic IPDOM of $\mathcal{I}$ by replacing the tail label, a predicate label, with its static IPDOM label, demanding the prefixing control dependence regions remain the same.

Example. Consider the example in Fig. 3.6. The first four pairs of trace entries have identical labels. Only the entries for 5_1 are recorded as the branch outcomes are different. The monitor detects that the fifth pair have different labels, i.e. 6_1 from the master and 7_1 from the slave. As such, it computes `dyn_ipdom`($\mathcal{I}[5_1]$)=`dyn_ipdom`($[4,5]$)=$[4,7]$. Thus, both executions proceed to index $[4,7]$, that is, 7_2 in (b) and 7_1 in (c). Note that if the static IPDOM 7 were used, due to the recursive call of `gee()`, we would mistakenly consider 7_1 to be the end of region Ⓑ. □

Handling Syscalls. The monitor also needs to handle syscalls because we want the two executions to share the sequence of external inputs as much as possible to reduce non-determinism and hence the meaningless differences caused by such non-determinism. It also reduces error-prone manual efforts. From the earlier discussion, we know both the master and the slave send their syscall parameters to the monitor. In addition, the master also sends its syscall outcome. For a syscall that occurs in both executions, the monitor compares their parameters, e.g. the size of a file read. If they are identical, the monitor relays the syscall outcome from the master to the slave. In the case that they are different due to differences introduced earlier, the monitor instructs the slave to execute the syscall instead of copying from the master.

In Algorithm 6, when the monitor receives a pair of syscall trace entries (line 8), it first retrieves the syscall outcome from the master (line 9). Then it consults a *policy table* that defines the actions for different kinds of syscalls. While the details of the policy table are discussed in Section 3.3, all we need to know now is that there are two possible actions for each syscall ID. COPY indicates the slave should copy the result from the master, and EXEC indicates the slave should execute the syscall. At line 10, the algorithm checks whether the action is COPY and if the parameters are identical. If so, it relays the outcome from the master (line 11). Otherwise, it instructs the slave to execute (line 13). When the two executions take different branches, the monitor retrieves and discards the syscall outcome from the master (lines 20-21), and instructs the slave to always execute its syscall (lines 26-27).

```
/*different file names are provided*/
filename = read();
...
while (1) {
  menu = selectMenu();
  if (menu == MENU_QUIT)
    quit();
  else if (menu == MENU_WRITE_MESSAGE) {
    message = read();
    if (exists(filename))
      rename(filename, filename + ".bak");
    fd = open(filename);
    write (fd, message);
  }
}
```

Figure 3.7.: Example for syscall handling

Example. Consider an example in Fig. 3.7. When the user selects the menu item `write message`, the program reads a message and writes it to a file with a name provided earlier. If the file already exists, the program first makes a backup. Suppose the user wants to compare two executions with two different files named `A` and `B`, and there is already a file with a name of `B`. Fig. 3.8 shows the master and the slave executions. Note that since the file name is different at 2_1, the following syscalls at 10_1, 11_1, 12_1, and 13_1 in the slave are executed instead of copied, which correctly exercises the intended different semantics. In contrast, the syscalls at 5_1 and 9_1 are copied. □

3.2.2 Decoupled Execution Mode

When the user wants to decouple the two executions and provide different inputs, he/she presses `ctrl-c` in the master execution, which sends an interrupt to the master. Note that the user does not interact with the slave execution (in most cases). The master checks for the interrupt before executing a system call. If it was received, the master informs the monitor. When the slave is about to execute the corresponding

Master		Slave		
2_1	filename = read();	2_1	filename = read();	E
4_1	while (1)	4_1	while (1)	
5_1	menu = selectMenu()	5_1	menu = selectMenu()	C
6_1	if (menu == ...)	6_1	if (menu == ...)	
8_1	else if (menu == ...)	8_1	else if (menu == ...)	
9_1	message = read();	9_1	message = read();	C
10_1	if (exists (filename))	10_1	if (exists (filename))	E
-		11_1	rename(filename, ...);	E
12_1	fd = open(filename);	12_1	fd = open(filename);	E
13_1	write (fd, message);	13_1	write (fd, message);	E

Figure 3.8.: Executions of the example in Fig. 3.7. The last column shows if the slave executes (E) the syscall or copies (C) result. The boxed entries are affected by the differences from 2_1.

system call, the monitor notifies the slave to start executing its syscalls. In other words, the two start to interact with the environment directly. Both continue to send their trace entries to the monitor to update their indices.

3.2.3 Re-synchronizing Master and Slave Executions

After providing the different inputs, the user presses another hot-key in both the master and the slave to indicate that he/she wants to re-synchronize the two executions such that they can share the same input events again to reduce non-determinism and manual interactions. However, when the interrupts are received, the two executions may be at different locations. The monitor needs to determine which execution is faster and block it until the slower one catches up. After the two re-synchronize, they resume in the coupled mode.

$$\boxed{\texttt{ahead\_of} : \mathit{ExecIndex} \times \mathit{ExecIndex} \ \rightarrow \ \mathit{Boolean}}$$

$$\texttt{ahead\_of}(l_1 \cdot \mathcal{I}_1, l_2 \cdot \mathcal{I}_2) = \begin{cases} \mathit{false} & \texttt{if } l_1 \neq l_2 \ \wedge \text{ there is not a path from } l_2 \text{ to } l_1; \\ \mathit{true} & \texttt{if } l_1 \neq l_2 \ \wedge \text{ there is a path from } l_2 \text{ to } l_1; \\ \texttt{ahead\_of}(\mathcal{I}_1, \mathcal{I}_2) & \texttt{otherwise i.e., } l_1 \equiv l_2 \end{cases}$$

$$\boxed{\texttt{normalize} : \mathit{ExecIndex} \times \mathit{ExecIndex} \times \mathit{ExecIndex} \rightarrow \ \mathit{ExecIndex} \times \mathit{ExecIndex}}$$

$$\texttt{normalize}(l_1 \cdot \mathcal{I}_1, \ \ l_2 \cdot \mathcal{I}_2, \ \ \mathit{nil}) = \begin{cases} \langle l_1 \cdot \mathcal{I}_1, \ \ l_2 \cdot \mathcal{I}_2 \rangle & \texttt{if } l_1 \neq l_2 \quad \text{[N-INIT-NEQ]} \\ \texttt{normalize}(\mathcal{I}_1, \ \ \mathcal{I}_2, \ \ l_1) & \texttt{if } l_1 \equiv l_2 \quad \text{[N-INIT-EQ]} \end{cases}$$

$$\texttt{normalize}(l_1 \cdot \mathcal{I}_1, \ \ l_2 \cdot \mathcal{I}_2, \ \ \mathcal{I}_p \cdot l_p) =$$

$$\begin{cases} \langle \mathcal{I}_p \cdot l_p \cdot l_1 \cdot \mathcal{I}_1, \ \ \mathcal{I}_p \cdot l_p \cdot l_2 \cdot \mathcal{I}_2 \rangle & \texttt{if } l_1 \neq l_2 \ \wedge \ l_2 \neq l_p \ \wedge \ l_1 \neq l_p \quad \text{[N-END]} \\ \texttt{normalize}(l_1 \cdot \mathcal{I}_1, \ \ \mathcal{I}_2, \ \ \mathcal{I}_p \cdot l_p) & \texttt{if } l_1 \neq l_2 \ \wedge \ l_2 \equiv l_p \quad \text{[N-RM-SECOND]} \\ \texttt{normalize}(\mathcal{I}_1, \ \ l_2 \cdot \mathcal{I}_2, \ \ \mathcal{I}_p \cdot l_p) & \texttt{if } l_1 \neq l_2 \ \wedge \ l_1 \equiv l_p \quad \text{[N-RM-FIRST]} \\ \texttt{normalize}(\mathcal{I}_1, \ \ \mathcal{I}_2, \ \ \mathcal{I}_p \cdot l_p \cdot l_1) & \texttt{if } l_1 \equiv l_2 \quad \text{[N-EQ-HEADER]} \end{cases}$$

Figure 3.9.: Re-synchronization primitives

We leverage execution indices to determine which execution is ahead of the other. Intuitively, an index represents the nesting of control dependence regions. Two indices sharing some common prefix means that the current execution points in the two runs are nesting in a common set of control dependence regions, called *aligned regions*. By comparing the relative positions of the two points inside the aligned regions, we can decide which execution is ahead. Consider Fig. 3.6. Point 7_1 in (c) is ahead of 5_1 in (b) because $\mathcal{I}_m(5_1) = [4, 5]$ and $\mathcal{I}_s(7_1) = [4, 7]$; they share a prefix [4] and 7 is ahead of 5 inside the region of 4. Similarly, 7_2 in (c) is ahead of 5_2 in (b). In particular, their indices share the prefix [4]; 7_2 in (c) is in the region denoted by index $[4, 4]$, i.e. Ⓔ representing the second iteration of the loop, whereas 5_2 in (b) is in the region denoted by $[4, 5]$, i.e. Ⓑ representing the `if` statement at line 5 in the first iteration of the loop). Ⓔ is ahead of Ⓑ.

The `ahead_of()` primitive in Fig. 3.9 defines how to decide index order. If the two indices share the same head label, it recursively checks the order of their tails. If the head labels l_1 and l_2 are different, and there is a path from l_2 to l_1, the first index is ahead. In Fig. 3.6, $\begin{aligned} &\texttt{ahead\_of}(\mathcal{I}_s(7_2), \mathcal{I}_m(5_2)) = \texttt{ahead\_of}([4, 4, 7], [4, 5, 6, 4, 5]) \\ &= \texttt{ahead\_of}([4, 7], [5, 6, 4, 5]) = \mathit{true}. \end{aligned}$ There are

cases where neither execution is ahead of the other if they are in the different branches of a predicate. In this case, the monitor will re-synchronize the two executions at the dynamic IPDOM of the predicate.

Dealing With Event Handling Loops. When the user sends a re-synchronization signal, the two executions are likely inside some event handling loop, which receives and handles external events. During the independent executions in decoupled mode, the two runs may have received different events, e.g. different numbers of key strokes, causing them to execute a different number of iterations. Index based re-synchronization recognizes that the execution that iterated more is ahead[1]. As such, it tries to execute the other that iterated less, but in fact the other execution cannot make progress as it has received all the needed external inputs and expects no more. When the

```
while (1) {
  if (select(STDIN, ...) == SUCCESS)
    /* process the user input */
    s1;
    ...
}
```

Figure 3.10.: Event handling loop example

user signals re-synchronization, he/she knows that the two executions have received the different inputs, and they should start to receive the same inputs again from then on. Hence, the iteration number differences of the event loop are not important. We introduce a normalization step to remove the excessive entries (in the indices) corresponding to unnecessary iteration differences. The primitive is defined in Fig. 3.9. It takes three indices, with the first two requiring normalization and the third an auxiliary parameter whose initial value is *nil*, and it produces two normalized indices. The auxiliary parameter represents the current common prefix during computation. According to the rules, it initially compares the two head labels. If different, the

[1]In indexing, an iteration nests within the region of the previous iteration such that the one iterating more has a longer index.

differences are not caused by the event loop, so it returns the two input indices as normalized (rule N-INIT-NEQ). Rules N-INIT-EQ and N-EQ-HEADER detect equivalent heads and move it to the tail of the common prefix. In rule N-RM-FIRST, if the two heads are different but the head of the first index is identical to the tail of the prefix, indicating repetitive entries, i.e. loop iterations, the head is removed. Rule N-RM-SECOND is symmetric. Rule N-END states the termination condition, in which the difference is not caused by repetition. The final normalized indices are the common prefix concatenated with the two current indices.

Example. Fig. 3.10 shows a very simple event loop. Assume it iterates two times and stops at line 5 in the master when the user signals re-synchronization, and it iterates four times and stops at 3 in the slave, yielding the indices $\mathcal{I}_m = [1, 1, 2, 5]$ and $\mathcal{I}_s = [1, 1, 1, 2, 4]$, respectively. Recall that loop iterations produce consecutive entries in the indices as an iteration directly nests in the region of the previous iteration. We have the following:

$$
\begin{array}{ll}
\quad \texttt{normalize}(\mathcal{I}_m, \mathcal{I}_s, \mathit{nil}) & \\
= \texttt{normalize}([1,2,5],\ [1,1,2,4],[1]) & \text{[N-INIT-EQ]} \\
= \texttt{normalize}([2,5],\ [1,2,4],\ [1,1]) & \text{[N-EQ-HEADER]} \\
= \texttt{normalize}([2,5],\ [2,4],\ [1,1]) & \text{[N-RM-SECOND]} \\
= \texttt{normalize}([5],\ [4],\ [1,1,2]) & \text{[N-EQ-HEADER]} \\
= [1,1,2,5],\ [1.1,2,4]) & \text{[N-NED]}
\end{array}
$$

The rules applied are presented on the right. Note that at the second step, the heads of the indices are different, but the second head is the same as the tail of the prefix, indicating repetition. The head is thus removed. Observe that after normalization, the two executions are within the same control dependence region, and the master is ahead.

Overall Re-synchronization Procedure. The procedure is informally described as follows. Assume $\mathcal{I}_m$ and $\mathcal{I}_s$ are the indices when a re-synchronization signal is received. The monitor first normalizes $\mathcal{I}_m$ and $\mathcal{I}_s$. If one execution is ahead of the other, it is blocked until the other one catches up. If neither is ahead, meaning they are in the two branches of some predicate, the monitor allows both to execute independently until the dynamic IPDOM of the predicate.

Syscall Policy for Slave Execution in Coupled Mode. In previous discussion, whether the slave should execute a syscall mainly depended on whether the syscall corresponded to one in the master and whether they had different parameters. However, the decision also depends on the type of the syscall. This is reflected by the policy table look up on line 10 of Algorithm 6. For example, we would like to execute user interface (UI) related syscalls such that the slave has its own UI.

Table 3.3.: Category of syscalls and the default policy.'E' and 'C' stand for execute and copy, respectively.

Category	Description	Policy
Memory	Allocate/free memories (`mmap()`, `munmap()`, ...)	E
Process	Create/kill/block processes (`fork()`, `clone()`, ...)	E[1]
Open	Open a file descriptor (`open()`, `create()`, ...)	E
Input	Read data from a file/environment (`read()`, `stat()`, ...)	C
Output	Write data to a file (`write()`, ...)	C,E[2]
GUI	Special case of network syscalls with X-Windows server	E,C[1,3]
Utility	Non-deterministic library functions (time(), ...)	C

1. there is ID translation during execution.
2. outputs to stdout and UI are executed, others copied.
3. requests and replies are executed and events are copied.

We categorize syscalls into 7 groups. Table 3.3 presents the groups and their policies. There are two possible policies: *execute* and *copy*. The former means that the slave executes the syscall whereas the latter means that the slave copies the result from the master and does not execute the syscall.

Process management syscalls follow the execute policy. In addition, the slave and the master both manage a mapping between process IDs. Syscalls creating new processes such as `fork()` and `clone()` return different process IDs in the two executions. To prevent propagating these different values to other parts of the executions, we

would like the corresponding processes in the two runs to have the same IDs. In particular, when `clone()` is called to create a thread, both runs assign the same logical thread ID to the newly created thread and map the real thread ID returned by the syscall to the logical ID. Our wrapper around `clone()` then returns the logical ID. Later, when the logical ID is used in other syscalls such as `exit()`, our wrapper maps it to the original real ID.

Open syscalls also use an execute policy because the descriptors they return may be used by other syscalls such as `mmap` or when the user introduces differences between the executions. It is possible that the master succeeds in opening a file but the slave fails to open the same file, for instance, when the master creates a file with the `O_EXCL` option first. In this case, the slave opens a new temporary file. Since I/O syscalls in the slave mostly copy their outcomes from the master, opening a different file will not affect the slave execution.

Most *input/output syscalls* follow the copy policy. One exception is that we *execute* output syscalls emitting to standard output or user interfaces to allow the slave to have its interface. In addition, both the master and the slave record the buffer content of an output syscall and meta information such as the definition point of the buffer, for further analysis, e.g. slicing.

GUI applications in Linux communicate with the X server through sockets. There are three types of messages: *requests*, *replies*, and *events*. Requests are sent from a client to the server to request a service such as creating a window and querying properties. Upon a request, the server sends the requested information with a reply. Events are sent by the server without the corresponding requests, denoting user interactions. Requests/replies are handled in a way similar to process management syscalls, which use the execute policy and ID translation, to provide the proper user interface. The master and the slave must map their window IDs to the same logical ID when the windows correspond to each other. Events have the copy policy, meaning the slave copies events from the master and ignores its own events.

The slave also copies signals from the master in coupled mode, ignoring its own signals except segfaults. We support threads by implementing a deterministic scheduler [51] that allows one thread to execute at a time. This suppresses non-determinism caused by different thread schedules.

3.4 Evaluation

We implement a prototype on Pin [42]. It can work on stripped binaries as it can generate dynamic control flow graphs (for indexing). In our experiments, we first quantitatively evaluate the space and time savings by our technique in comparison with offline techniques. We then apply the prototype to three comparative analysis tasks.

Table 3.4.: Applications in feature identification

Program	Feature Description
alpine-1	Send an e-mail
alpine-2	Create a directory in remote imap server
xv	Convert two different bitmap images to jpeg images
smbc	Create a directory in remote samba server
ncmpc	Add a song to playlist

3.4.1 Examined Comparative Tasks

Feature identification. As seen in Sec. 3.1, feature identification involves locating the portion of a program responsible for a feature [15–19, 21–23], and this can be done by comparing executions of a program over different inputs [46]. Table 3.4 presents the full set of examined feature identification tasks, which have appeared in published work [46].

Table 3.5.: Subjects in comparative debugging and regression understanding. The regression bugs are tagged with *

Program	Bug #	Description
grep-1	12128	Numeric parameters are incorrectly interpreted
grep-2	16567	-i option does not work with a regular expression
grep-3	27919	
grep-4	27919	
grep-5	21199	-w option causes incorrect search results
make-1	25493	Incorrectly handle dependencies in rules
make-2	18622	User-defined rules conflict with default rules
tar-1	508199	Cannot restore files from backup
tar-2	598636	Cannot handle broken symbolic links
tar-3	637085	Cannot handle longer filenames
grep-6*		Search incorrect if -i, -n options are used together
grep-7*	36567	-i does not work with multi-byte characters
grep-8*		
find-1*	34976	Fail to save working directory
find-2*	29949	-execdir does not change working directory
make-3*	31155	Incorrect order in parsing patterns
make-4*	39310	Commandline options are applied multiple times
rm*		-I, –interactive=once does not work same
seq*		[seq 1 3 1] treated as [seq 1 3]
cp*		–no-preserve=mode exits 1
cut*		Incorrect error message
expr*		Incorrect computation with negative value

Comparative Debugging. To help developers, we integrate our technique into `gdb` and provide a few new `gdb` primitives that allow developers to modify variables and compare the execution with changes to an execution without them. Details on the changes to `gdb` are presented in a later case study.

Understanding regression. We also apply our technique to executions of an old, working version of a program and a new, failing version to understand regressions.

All the examined bugs are real world bugs from [3, 52]. Table 3.5 describes them with GNU Savannah bug IDs if any.

3.4.2 Disk Usage and Performance

In this experiment, we examine the impact of dual execution on the running time and disk usage, when compared with offline techniques. We first collect traces of the two executions for all the subjects (in the three tasks). We then perform offline comparison. The results are used as the base line. We then use our prototype to generate difference traces on the fly and compare the results. We used an Intel Core i7 machine running Arch Linux 3.15.5 with 16GB RAM.

Table 3.6 presents the trace size comparison. The Traces column shows the size of full traces. The Diffs (W/O Dual Exec.) column shows the size after offline trace comparison. The Diffs (W/ Dual Exec.) column shows the size of difference traces generated online. The % Full column shows the size of the difference traces as a percentage of the full traces.

Note that we do not use any trace compression technique. However, our technique is orthogonal to those techniques and users can combine the techniques to achieve smaller traces. Also many trace compression techniques targets specific trace abstractions but our technique can be used on a fine-grained trace including data and control dependencies.

Observe that using dual execution always produces much smaller (difference) traces than the *full traces* (5.41% on average). This is because the full traces include every

Table 3.6.: Trace size comparison

Program	W/O Dual Exec.		W/ Dual Exec.	
	Traces	Diffs	Diffs	% Full
seq	62MB	15MB	15MB	24.19
make-3	1.8GB	409MB	358MB	19.42
make-4	4.6GB	887MB	886MB	18.81
grep-7	4.0GB	978MB	653MB	15.94
find-1	4.2GB	562MB	541MB	12.58
find-2	4.0GB	462MB	447MB	10.91
grep-8	4.0GB	432MB	424MB	10.35
grep-5	12.5GB	2.7GB	1.0GB	8.00
cut	79MB	6.0MB	6.0MB	7.59
grep-4	13GB	438MB	428MB	3.22
rm	3.8GB	117MB	117MB	3.00
xv	14GB	559MB	380MB	2.65
cp	68MB	1.4MB	1.3MB	1.91
tar-1	377MB	17MB	6.8MB	1.80
ncmpc	6GB	120MB	93MB	1.51
tar-2	457MB	6MB	4.0MB	0.88
smbc	55GB	627MB	432MB	0.77
make-1	15.7GB	1.5GB	105.2MB	0.65
expr	58MB	306KB	305KB	0.51
make-2	7.3GB	104.9MB	32.2MB	0.43
alpine-2	57GB	320MB	205MB	0.35
grep-3	14GB	41.2MB	40.7MB	0.28
grep-6	4.0GB	9.1MB	7.7MB	0.19
tar-3	11.4GB	35MB	6.4MB	0.055
grep-1	10.6GB	8.6MB	832KB	0.0074
alpine-1	60GB	344MB	170MB	0.0003
grep-2	14.8GB	3.4KB	1KB	0.000006
			Geometric Mean	5.41

instruction of both executions, while dual execution only records differences. For tasks comparing executions, there is often substantial similarity between the executions. There is also a lot of similarity across executions from initialization and configuration behavior that is irrelevant to the analysis tasks.

Dual execution also consistently produces smaller differences than the differences by offline comparison. This occurs because many applications have nondeterministic behaviors that the user cannot control, which introduce additional differences. For example, `alpine` reads and uses the system time quite often, causing a lot of non-deterministic differences. Dual execution eliminates such differences by sharing system events as much as possible.

Table 3.7 presents the time comparison. The Native column shows the original native execution time, i.e. sum of the master and the slave. Tracing and Comp present the time taken for whole execution tracing and offline comparison. Total_1 is their sum. Total_2 presents the time for dual execution. The last two columns shows the comparison. Running times of interactive programs are not comparable against Native, denoted by '*'. Entries in the table are sorted by $\text{Tot}_2/\text{Tot}_1$. A lower number means that dual execution improved the running time.

Observe that the time spent on dual execution is almost always lower than the time for full execution tracing and offline comparison, 0.49% on average. The benefits are more obvious for larger programs and longer runs. These improvements are mostly due to the smaller traces that are actually recorded when using dual execution. The slowdown of our system is 2022× relative to native on average. It is more suitable for inhouse testing and debugging.

3.4.3 Case Studies

Feature Identification

In this case study, we use our prototype to identify 5 functional features from 4 real world applications. We use our prototype to trace and compare executions and

Table 3.7.: Execution time comparison

Program	Native	W/O Dual Execution			W/ Dual Execution		
		Tracing	Comp	$Total_1$	$Total_2$	Tot_2/Tot_1	Tot_2/Nat
expr	.002s	3s	0.5s	4s	6s	1.50	3000
seq	.002s	4s	0.5s	5s	7s	1.40	3500
cut	.002s	4s	0.7s	5s	7s	1.40	3500
grep-6	.01s	10s	29s	39s	49s	1.26	4900
cp	.01s	4s	0.5s	5s	6s	1.20	600
make-3	.01s	19s	21s	40s	36s	0.90	3600
grep-7	.01s	11s	42s	53s	42s	0.79	4200
find-2	.03s	11s	39s	50s	38s	0.76	1267
find-1	.02s	12s	46s	58s	44s	0.76	2200
ncmpc	14s	1m 12s	1m 3s	2m 15s	1m 42s	0.76	*
tar-1	.03s	18s	1s	19s	12s	0.63	400
tar-2	.01s	20s	2s	22s	12s	0.55	1200
grep-5	.05s	1m 16s	1m 45s	3m 1s	1m 32s	0.51	1840
make-4	.01s	42s	38s	80s	40s	0.50	4000
grep-8	.01s	12s	45s	57s	28s	0.49	2800
rm	2s	11s	32s	43s	19s	0.44	*
tar-3	.02s	1m 0s	1m 58s	2m 58s	1m 15s	0.42	3750
xv	16s	2m 20s	3m 42s	6m 2s	2m 31s	0.42	*
grep-3	.03s	1m 10s	1m 9s	2m 19s	45s	0.32	1500
grep-2	.02s	1m 16s	1m 11s	2m 27s	47s	0.32	2350
grep-4	.02s	1m 6s	59s	2m 5s	38s	0.30	1900
make-2	.03s	52s	39s	1m 31s	27s	0.30	900
smbc	40s	5m 32s	7m 2s	12m 34s	3m 10s	0.25	*
grep-1	.02s	1m 0s	1m 15s	2m 15s	26s	0.19	1300
alpine-2	16s	5m 20s	6m 31s	9m 51s	1m 47s	0.18	*
alpine-1	12s	6m 42s	8m 59s	15m 41s	2m 14s	0.14	*
make-1	.01s	1m 28s	1m 59s	3m 27s	20s	0.10	2000
					Geometric Mean	0.49	2022

use an offline technique similar to [46] to identify functions from the trace. For each case, we compare two executions that exercise the feature with different inputs by using our prototype, and we identify the relevant differences, which are supposed to correspond to the features. For example, in alpine-1 case, we compare two executions sending two different mails. The differences between the executions should be the feature responsible for sending different emails if there is no nondeterminism that is not relevant to the input differences. Our prototype suppresses those nondeterminism. Once we get the changes, we consider the highest function(s) on the call graph that can cover all the differences to be the functional component for the feature. The differences generated by our tool allow us to precisely locate the correct functions. For example, all the differences in **xv** can be covered by two functions with the names of `LoadBMP()` and `SaveJPEG()`, which clearly indicate they are the intended functions. For **alpine**, we have located `call_mailer()` for email sending and `add_new_folder()` for directory creation. For **smbc**, `RcreateDir()`. For **ncmpc**, `mpd_async_send_command_v()`.

All these applications contain event handling loops that behave differently between two executions because different user input timings or packet arrival timings perturb the loop behavior, causing undesirable execution differences.

```
_XReply() {
    while (1) {
        reply = poll_for_reply();
        if (reply->sequence == expected) break;
        ...
}
poll_for_reply() {
    while (poll() == -1) ...;
}
```

Figure 3.11.: Simplified event handling loop in libX11

For the **xv** case, from the results in Table 3.6, 559MB-380MB=179MB trace differences are due to such non-determinism, which largely originates from an event

handling loop for reply messages from the X server, as shown in Fig. 3.11. In the de-coupled mode, the different sequences of X messages make the loop at line 2 iterate different times. As a result, full traces cannot be properly aligned, leading to lots of undesirable differences during offline comparison. In contrast, the normalization step in the re-synchronization mode in our technique suppresses such differences.

Comparative Debugging Using GDB

Table 3.8.: New `gdb` commands supported by dual execution

Command	Interface
`dslice`	`dslice [instruction address] [instance]`
`dset`	`dset [variable] [value1] [value2]`
`dprint`	`dprint [variable]`

We integrated our prototype with `gdb` and provided new debugging commands as shown in Table 3.8. `dslice` generates a dual slice [2] of the given instruction and instance. The slice contains execution differences causally related to the difference at the given slicing criterion. `dset` sets a variable in two executions to different values; `dprint` shows two values of a variable side by side. Basic commands such as setting a breakpoint and continuing the execution are applied to both executions by default. We then used the enhanced `gdb` to debug the 10 non-regression failures. Each required fewer than 15 manual steps to capture the causality of the failure. We also tried to use the vanilla `gdb` to achieve the same results for three cases, `grep-5`, `tar-2`, and `tar-3`, but failed due to the prohibitively tedious manual interactions involved.

Next, we present our experience with the `grep-5` bug. The "-w" option in `grep` selects lines containing whole word matches. The buggy program prints out only a substring of a matched line. We launched the buggy program in our enhanced `gdb`. Two processes were automatically created and run in coupled mode. We set

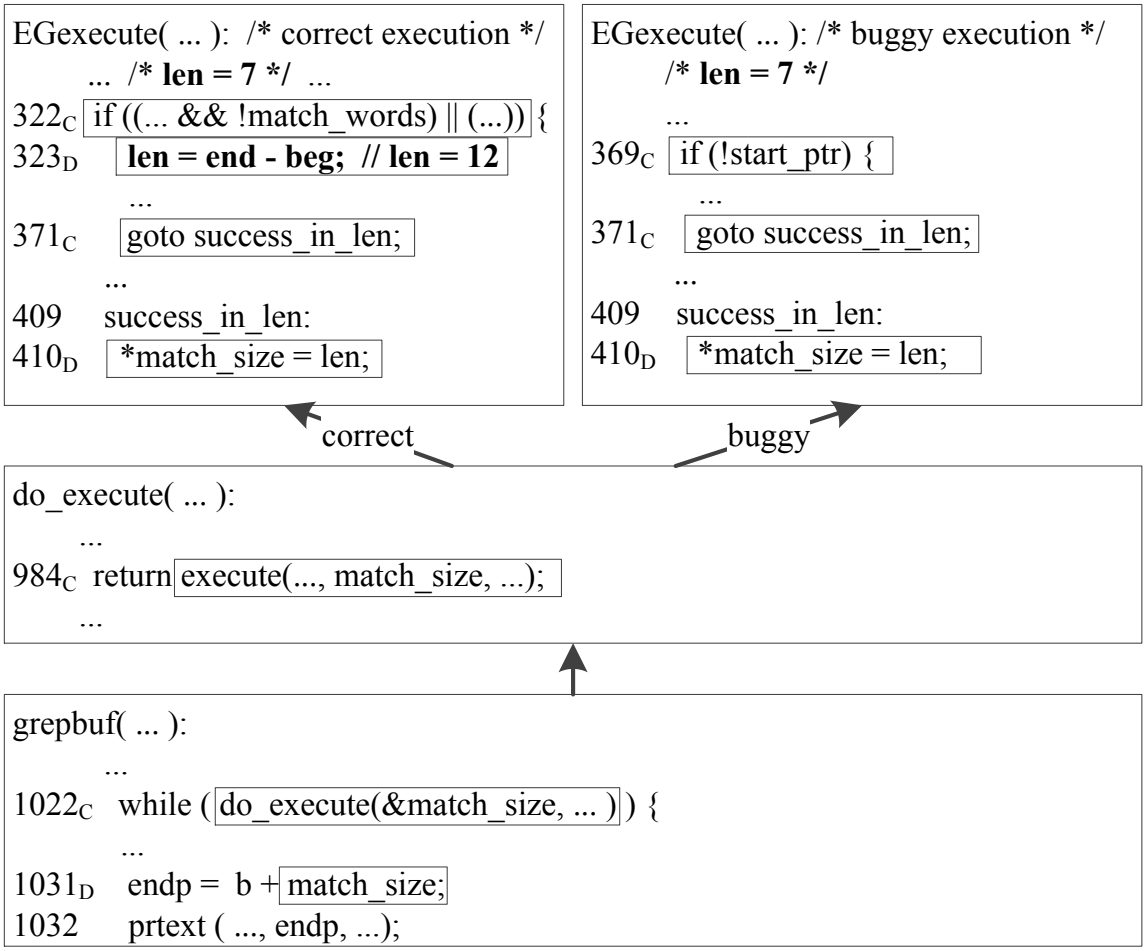


Figure 3.12.: Slicing results for `grep`

a breakpoint at `prtext()`, which printed the incorrect output. Few places in the immediate source code were affected by the "-w" option; the `match_word` variable was one of them. It had value 1 in the buggy run. We wanted to perturb its value and observe the effect, and more important, the causality. We used `dset` to set it to 0 and 1 in the two respective runs. At the breakpoint, we observed output differences: the execution with `match_word` set to 0 produced the whole line. We then used `dslice` to slice from this output difference. Fig. 3.12 presents the resulting dual slice. The results from the correct run are on the left side and those from the buggy run are on the right. `do_execution()` and `grepbuf()` are common to both executions, though

they have different dependencies in `EGexecution()`. The subscript C and D on line numbers represent control and data dependencies respectively. The bug manifests when `prtext()` prints different results at line 1032 in `grepbuf()` because of the differences in the variable `endp`. The slice result shows that the two executions have different `match_size` values, 12 versus 7. It also shows that the control dependencies include `do_execute()`, `execute()`, and `EGexecution()`. Observe that in the correct run, `len` is defined at line 324, which eventually allows printing the whole line, whereas there is no such definition in the buggy run. Therefore, the root cause is that such a definition is missing when the option is set, which is confirmed by the bug report.

Next, we show our experience in doing the same comparative debugging with the vanilla `gdb`. We started two executions of the buggy program and attached them to two separate `gdb` instances. Then we set the `match_word` variable to 0 and 1 again. But the challenges lie in monitoring the propagation of the value differences. We first tried to single-step the two executions. But the definition point of `match_word` and the output point are separated by a substantial amount of computation in `EGexecute()`. Even worse, there were control flow differences due to our perturbations such that we had to somehow manually align the two executions. The process quickly became unmanageable. Another attempted option was to identify related variables and set break-points and watch-points. However, we could not solely use watch-points as many related variables were stack variables. However, using break points was also problematic. In particular, when we set a break point at the access to `match_size` in `do_execute()`, we found that the access occurs more than 200 times and only the 150th instance shows a difference across the two runs. The process is prohibitively tedious and error-prone. In contrast, our new commands and the underlying dual execution engine make interactive comparative-debugging feasible.

We also point out that the enhanced `gdb` is more flexible than a stand-alone slicing tool as it allows interactively perturbing program state at any point.

Table 3.9.: Dual slicing regressions

Program	# of Instr. in differences	# of Instr. in slice	Time
grep-6	44K	35K	0.1s
grep-7	15K	7.7K	16s
grep-8	213K	76K	6s
find-1	42K	2.0K	6s
find-2	172K	120K	5s
make-3	888K	127K	4s
make-4	753K	108K	3s
rm	3.4K	214	1.5s
seq	902	516	0.2s
cp	2179	332	0.05s
cut	952	541	0.05s
expr	787	62	0.03s

Understanding Regressions

In this case study, we applied an existing dual slicing [2] tool on the difference traces generated by the dual execution engine to understand regressions. As mentioned earlier, dual slicing computes the execution differences related to the difference at the slicing criterion. In the experiment, we use a text differencing tool to generate syntactic mappings between statements in the two program versions and propagate such mappings down to the instruction level to facilitate our engine. The slicing criteria are output differences. In the case of crashing bugs, they are the pointer de-references. The results are shown in Table 3.9. The second column shows the total instructions in the difference traces, which are already a very small portion of the full traces (Table 3.6). The third column shows the slice size. Observe that the slices are much smaller than the differences for most cases. We also confirm that all of them include the root causes. We suspect the slice sizes can be further reduced if we use a better syntactic mapping algorithm. But that is beyond our scope.

3.5 Related Work

Execution comparison. Execution comparison is used in debugging [1, 53–55], concurrency failure understanding [2], vulnerability detection [6], and binary reuse [46]. Comparative causality [3] produces bug explanations by replacing program states on the fly. Sieve [5] compares traces from program versions to identify the root causes of regressions. Hoffman et al. proposed a semantic-aware trace analysis [56] for understanding executions, particularly identifying the cause of regressions. In comparison to these works, the dual execution engine avoids generating full traces. Instead, it performs online comparison and only records differences.

Execution Replication and Replay. Execution replication has been widely studied [57–62]. The premise is similar to n-version programming [63], which runs different implementations of the same service specification in parallel. Then, voting is used to produce a common result tolerating occasional faults. Vandiver et al. [64] proposed a technique that handles Byzantine faults in database transaction processing using replicated systems. Chun et al. [65] run diversified replicas on multi-core processors to handle Byzantine faults. There are also many security applications [7, 66–68] of n-variant execution. McDermott et al. [69] proposed a defense technique based on logical replication. They re-execute commands on each replicated system and detect differences among the replicas. TightLip [70] runs a replicated process in parallel to an original process and analyzes the replica to prevent information leakage. There is also a large body of works in execution replay [71–76] that aim to faithfully reproduce an execution. Compared to these works, our technique allows differences in executions and handles the complex consequences of these differences.

Viennot et al. [49] proposed a technique that replays events from one version of a program with another version of the program. It hence also allows sharing syscalls across different executions. However, it requires exploration steps to find the best replay. In contrast, our technique exploits fine-grained traces and can align executions on-the-fly.

Execution Alignment. Alignment techniques identify corresponding points [50, 77, 77, 78] and memory locations [79] across different executions. Xin et al. proposed Execution Indexing (EI) [50] to precisely locate corresponding points across executions. Our technique is built on EI. We have overcome many new challenges such as lockstep synchronization, re-synchronization, and syscall dispatching for our purpose.

4 APEX: AUTOMATIC PROGRAMMING ASSIGNMENT ERROR EXPLANATION

In this chapter, we presents APEX, that can automatically generate explanations for programming assignment bugs, regarding where the bugs are and how the root causes led to the runtime failures. It works by comparing the passing execution of a correct implementation (provided by the instructor) and the failing execution of the buggy implementation (submitted by the student). The technique overcomes a number of technical challenges caused by syntactic and semantic differences of the two implementations. It collects the symbolic traces of the executions and matches assignment statements in the two execution traces by reasoning about symbolic equivalence. It then matches predicates by aligning the control dependencies of the matched assignment statements, avoiding direct matching of path conditions which are usually quite different. Our evaluation shows that APEX is every effective for 205 buggy real world student submissions of 4 programming assignments, and a set of 15 programming assignment type of buggy programs collected from `stackoverflow.com`, precisely pinpointing the root causes and capturing the causality for 94.5% of them. The evaluation on a standard benchmark set with over 700 student bugs shows similar results. A user study in the classroom shows that APEX has substantially improved student productivity.

4.1 Introduction

According to a report in 2014 [80], computing related job opportunities are growing at two times the CS degrees granted in US. The US Bureau of Labor Statistics predicts there will be one million more jobs than students in just six years. As a result, CS enrollment surges in recent years for many institutes. With the skyrocketing

enrollments, the luxury of one-to-one human attention in grading programming assignments may no longer be afforded. Automating grading is of a pressing need. In the current practice, automated programming assignments grading is mainly by running the submissions on a test suite. Failing cases are returned to the students, who may have to spend a lot of time to debug their implementation if they receive no hints about where the bug is and how to fix it. While the instructor may manually inspect the code and provide such feedback, these manual efforts can hardly scale to large classes.

In a recent notable effort [81], researchers have proposed to use program synthesis to correct buggy programming assignments. Given a correct version and a set of correction rules, the technique tries to sketch corrections to the buggy programs so that their behavior match with the correct version. Despite of the effectiveness of the technique, the demand of providing the correction rules adds to the burden of the instructor. Later in [82], a technique was proposed to detect the algorithm used in a functionally correct student submission and then suggest improvement accordingly. The onus is on the instructor to prepare the set of possible algorithms and the corresponding suggestions.

In this chapter, we aim to develop an automatic bug explanation system for programming assignments. It takes a buggy submission from the student, a correct implementation from the instructor, and a failing test case, then produces a bug report that indicates the root cause and explains the failure causality. Since the submission and the correct implementation are developed by different programmers, they are usually quite different. Different variable names, control structures, data structures, and constant values may be used (Section 4.2). Note that the faulty statements are part of such differences. Recognizing them from the benign differences is highly challenging.

Debugging by comparing programs and program executions is not new. Equivalence checking [83, 84] was leveraged in [85] to derive simple and partial fixes to internal faulty state, guided by a correct execution. However, substantial structural changes

between versions often make fixing internal state difficult. Weakest pre-conditions that induce behavioral differences across versions were identified and used to reason about bugs [86]. This technique relies on SMT solver and focuses on finding root cause conditions. It hardly explains the causality of failures, which is equally important. Another kind of techniques is dynamic analysis based. Comparative causality [3], dual slicing [2], and delta debugging [87] compare a passing run with a failing run and generate a causal explanation of the failure. However, they often assume the executions are from the same program to preclude syntactic differences that are difficult for dynamic analysis.

APEX is built on both symbolic and dynamic analysis, leveraging the former to handle syntactic differences and using the latter to generate high quality trace matches and causal explanations. It works by comparing the passing execution from the correct implementation and the failing execution from the buggy implementation. It collects both concrete execution traces and symbolic traces. The latter captures the symbolic expressions for the values occurring during execution. It then uses a novel iterative algorithm to compute matchings that map a statement instance to some instance(s) in the other version. The matchings are computed in a way aiming to maximize the number of equivalent symbolic expressions and respect a set of well-formedness constraints. A *comparative dependence graph* is constructed representing the dynamic dependencies from both executions. It merges all the matched statement instances and their dependencies to single nodes and edges respectively, and highlights the differences. A *comparative slice* is computed starting from the different outputs. A bug report is derived from the slice to capture the root cause and failure causality. Our contributions are summarized as follows.

- We formally define the problem and identify a few key constraints in constructing well-formed matchings. Specifically, we have formulated the main challenge of generating statement instance matchings as a *partial maximum satisfiability* (PMAX-SAT) problem.

- We develop an iterative algorithm that guarantees well-formedness while approximating maximality.
- We develop a prototype APEX. Our evaluation on 205 buggy real world buggy student submissions from 4 programming assignments and a set of 15 programming assignment type of programs collected from [88] shows that APEX can correctly identify the root causes and causality in 94.5% of the cases and generate very concise bug reports. The evaluation on a standard benchmark set [89] with over 700 student bugs shows similar results. A user study in the classroom shows that APEX has substantially improved student productivity.

4.2 Motivation

In our context, the buggy and the correct implementations are developed by different programmers. As such, they often have substantial differences representing the various ways to implement the same algorithm. We call them the *benign differences*. However, they are mixed with *buggy differences*. Our tool needs to distinguish the two. We classify popular benign differences into two categories.

Type I: Syntactic Differences. The two implementations may use different variable names and different expressions, such as `int pivot= low + (high - low) / 2` versus `int pivot= (hi - lo) / 2` . These differences may be eliminated by comparing their symbolic expressions.

Type II: Semantic Differences. (1) Different conditional statements or loop structures may be used.

Example. Consider the code snippets in Fig. 4.1. They are part of two programs collected from `stackoverflow.com` that compute the sum of even fibonacci numbers. The initial numbers are $N0$, $N1$, and the upper bound is N. Program (b) represents a correct implementation. The buggy version in (a) leverages that an even fibonacci number occurs after every two odd numbers. It hence uses a for loop in lines 4-12 to compute fibonacci numbers in groups of three and add the last one (the even number)

```
int sum_of_even_fibonacci(int N0, int N1, int N) {
  int i0=N0, i1=N1, sum=0, n=N, eSum=2, status=1;
  while(i1 < n && status == 1) {
    for(int cycle = 3; cycle > 0; cycle--) {
      sum = i0 + i1;
      i0 = i1;
      i1 = sum;
      if((i1 + i0) > n) {/*buggy, should be (i1 > n)*/
         status = 0;
         break;
      }
    }
    if (status == 1)
        eSum += sum;
  }
    return eSum;
}
```

(a) Buggy implementation

```
int sum_of_even_fibonacci(int N0, int N1, int N) {
  int n0=N0, n1=N1, n2=0, n=N, sum=2;
  for(;;) {
    n2 = n0 + n1;
    if ( n2 > n )
      break;
    if ( n2 % 2 == 0 )
      sum += n2;
    n0 = n1; n1 = n2;
  }
  return sum;
}
```

(b) Correct implementation

Figure 4.1.: Sum of even fibonacci numbers from `stackoverflow.com` [90]. Both assume `N0`=1 and `N1`=2 so that `eSum` starts with 2

to the sum. The bug is at line 8. While the predicate should test if the new fibonacci number exceeds the upper bound, the developer forgot that `i1` has been updated at line 7 and mistakenly used `i1+i0` to denote the new number. As a result, the execution terminates earlier, missing a fibonacci number in the sum. Observe that the two implementations have different control structures. □

(2) Different values may be used in achieving similar execution control. For example, in two Dijkstra implementations collected from *stackoverflow.com*, one uses a boolean array `visited[i]` to denote if a node i has been visited whereas the other uses an integer array `perm[i]` with values `MEMBER` and `NONMEMBER` to denote the same thing.

(3) Various data structures may be used. These differences, when mixed with the differences caused by bugs, make it very challenging to meet our goal. Note that equivalence checking [83] that reasons about the symbolic equivalence of *final outputs* is less sensitive to these differences as it does not care about equivalence of internal states. However in our context, we need to align internal states to generate failure explanations.

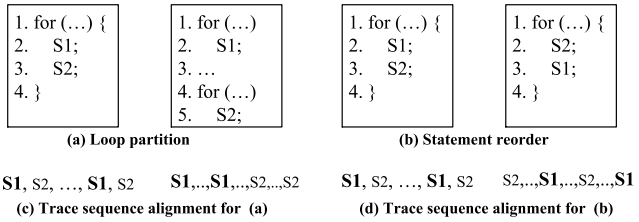


Figure 4.2.: Program differences difficult for sequence alignment. Only the highlighted entries in (c) and (d) are matched

Limitations of Sequence Alignment. A widely used approach to aligning program traces is sequence alignment [91] that identifies the longest common subsequence of two traces. It seems that we could extend the algorithm to match the sequences of symbolic expressions to identify the parts that are bug-free. However, we found that such an algorithm did not perform well in our context because the two programs

are often quite different. Consider Fig. 4.2. In (a), the loop in the left program is partitioned to two in the right program, which are semantically equivalent to the original loop. As a result, statements `S1` and `S2` have different orders in the traces in (c). The traces cannot be fully matched by sequence alignment although they are semantically equivalent. Similarly, the statement reordering in (b) also leads to that trace entries cannot be fully matched in (d). Furthermore, the two versions may use completely different path conditions (e.g., Fig. 4.1). As such, sequence alignment cannot match the symbolic expressions of the predicates even though they may serve the same functionalities.

Illustrative Example. Next, we are going to use the example in Fig. 4.1 to illustrate the results produced by APEX.

Fig. 4.3 shows part of the traces for the implementations in Fig. 4.1. The first columns show the dynamic labels (e.g. 5_2 denotes the second instance of statement 5). The second column presents the *dynamic control dependencies* (DCD). For instance, DCD(4_1)=E-3_1 means that 4_1 is dynamically control dep. on 3_1, which is further control dep. on the entry E. The third columns show the executed statements. The fourth columns present the symbolic expressions with respect to the input variables (for the assignment statements). The last columns show the values. From the symbolic traces, our tool will identify the equivalent symbolic expressions leveraging a SMT solver, as illustrated by the lines in Fig. 4.3. The tool then matches the DCDs of the matched symbolic expressions. Note that we cannot match DCDs by the symbolic equivalence of the predicate expressions as they are often different. Instead, we match them by well-formedness constraints (Section 4.3).

The lines in Fig. 4.1 represent the computed statement matchings. Lines 3, 4 and 8 in (a) are matched with 5 in (b), as they are the loop conditions. Line 5 in (a) is matched with line 4 in (b), denoting the computation of the new fibonacci number. Lines 13-14 in (a) are matched with lines 7-8 in (b), both updating the sum. These statement matchings can be considered as a common sub-program of the two versions.

label	DCD	code	symb expr	c.value
3_1	E	i1 < n	-	True
4_1	E-3_1	cycle > 0	-	True
5_1	E-3_1-4_1	sum=i0+i1	N0+N1	3
8_1	E-3_1-4_1	(i1 + i0) > N	-	False
4_2	E-3_1-4_1-8_1	cycle > 0	-	True
5_2	E-3_1-4_1-8_1-4_2	sum=i0+i1	N1+N0+N1	5
8_2	E-3_1-4_1-8_1-4_2	(i1+i0)>N	-	False
...	...	...	...	...
5_3	E-3_1-4_1-8_1-4_2-8_2-4_3	sum=i0+i1	N1+N0+N1+ N0+N1	8
...	...	...	...	...
13_1	E-3_1	status ≡ 1	-	True
14_1	E-3_1-13_1	eSum += sum	N1+N0+N1+ N0+N1+2	10

label	DCD	code	symb expr	c.value
4_1	E	n2= n0 + n1	N0+N1	3
5_1	E	n2 > n	-	False
7_1	E-5_1	n2%2==0	-	False
4_2	E-5_1	n2=n0+n1	N1+N0+N1	5
5_2	E-5_1	n2>n	-	False
7_2	E-5_1-5_2	n2%2==0	-	False
4_3	E-5_1-5_2	n2=n0+n1	N1+N0+N1+N0+N1	8
5_3	E-5_1-5_2	n2>n	-	False
...	...	...	...	
7_3	E-5_1-5_2-5_3	n2%2==0	-	True
8_1	E-5_1-5_2-7_3	sum+=n2	3×N1+2×N0+2	10

Figure 4.3.: Part of the symbolic and concrete traces for Fig. 4.1 where N0=1, N1=2, N=32. The copy statements are precluded

Intuitively, we reduce the problem to analyzing the two executions of the common sub-program.

From the trace matching results, a *dynamic comparative dependence graph* (DCDG) is constructed. The graph represents dynamic dependencies in both executions. It merges statement instances and dependencies that match. In the presence of bugs, a statement may have some of its instances matched but not the others. These instances that are supposed to match but do not are called the *aligned but unmatched* instances. They are usually bug related. For example in Fig. 4.1, line 8 in (a) has all its instances matched with line 5 in (b) except the last one, which took the wrong branch outcome due to the bug. The last one is hence an aligned but unmatched instance. We also merge such instances in the graph but highlight their different values. Statement instances that are neither matched nor aligned are represented separately. Note that in the paper, words "*align*" and "*match*" have different meanings.

Fig. 4.4 presents the DCDG for the executions in Fig. 4.3. Plain nodes represent matched statement instances. Green nodes are instances that are aligned but unmatched. Each plain/green node contains instances from both runs. Red/yellow nodes are those only present in the buggy/correct run, each containing only one instance. Label "a-5_1" means the first instance of line 5 in version (a). The concrete values are also presented in the right side of the nodes. Observe that the computations of the fibonacci numbers 3, 5, 8, 13, 21, and 34 are matched (i.e. a-5_1 vs. b-4_1,..., a-5_6 vs. b-4_6). The corresponding loop conditions and the first updates of the sum (i.e. a-14_1 vs. b-8_1) are also matched.

The loop conditions a-8_5:`if (i1+i0>n)` and b-5_6:`if (n2>n)` are aligned but not matched (i.e. the first green node). Hence the buggy execution exits the loop whereas the correct execution continues. Consequently, the conditions guarding the updates of the sum are also aligned but not matched (i.e. the second green node). As such, the sum was updated in the passing run but not in the failing run.

A *comparative slice* is computed starting from the two different outputs, denoting the causal explanation of the differences. A bug report is generated from the slice,

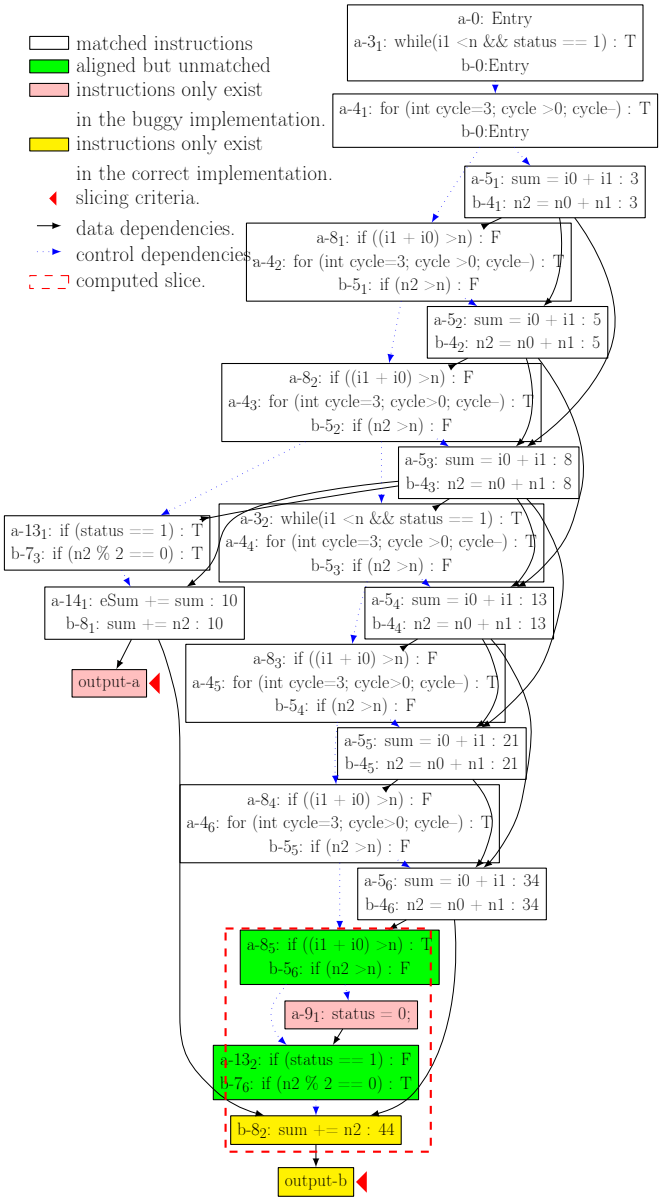


Figure 4.4.: DCDG for the example in Fig. 4.1

explaining (1) what the buggy version has done wrong and (2) what is the correct thing to do. Since part (2) is usually derived from the correct version invisible to the student, our tool translates it using the variable names in the buggy version. In Fig. 4.4, the two output nodes with triangles are the slicing criterion. The dotted box represents the slice. The root of the slice is exactly the buggy statement and its

DEFINITIONS:

$Label \quad \ell, t \qquad LabelInst\ \ell_i/t_j$: the i/j-th instance of label ℓ/t

$ispred(\ell_i)$: if ℓ_i is a predicate instance

$sym\_expr(\ell_i)$: symbolic expression of ℓ_i

$DCD(\ell_i)$: predicate instances that ℓ_i directly/transitively control dep. on

$\ell_i \rightsquigarrow \ell'_k$: ℓ'_k is directly/transitively dependent on ℓ_i

$\ell_i \leftrightarrow t_j$: the statement instance at ℓ_i matches with that at t_j

WELL-FORMEDNESS CONSTRAINTS:

[WF-SYM]

$$\ell_i \leftrightarrow t_j \implies (\neg ispred(\ell_i) \implies sym\_expr(\ell_i) \equiv sym\_expr(t_j))$$

[WF-CD]

$$\ell_i \leftrightarrow t_j \implies (\forall \ell'_k \in DCD(\ell_i),\ \exists t'_l \in DCD(t_j)\ \ \ell'_k \leftrightarrow t'_l) \wedge (\forall t'_l \in DCD(t_j),\ \exists \ell'_k \in DCD(\ell_i)\ \ \ell'_k \leftrightarrow t'_l)$$

[WF-X]

$$\ell_i \leftrightarrow t_j \implies \neg\exists \ell'_k, t'_l,\ \ell'_k \leftrightarrow t'_l \wedge ((\ell'_k \rightsquigarrow \ell_i \wedge t_j \rightsquigarrow t'_l) \vee (\ell_i \rightsquigarrow \ell'_k \wedge t'_l \rightsquigarrow t_j))$$

Figure 4.5.: Definitions and constraints for instance matching

alignment (a-8_5 vs. b-5_6), which have different branch outcomes. The interpretation of the slice, in the language of the buggy version, is that "*Statement 8* `if(i1+i0>n)` *should have taken the false branch. As a result, statement 9* `status=0` *should not have been executed. Consequently, statement 13* `if(status==1)` *should have taken the true branch,* `eSum+=sum` *should have been executed, and eventually* `eSum` *should have been 44 instead of 10*". It precisely catches the root cause and failure causality, and provides strong hints about the fix. Note that the yellow node b-8_2:`sum+=n2` is translated to `eSum+=sum` in the buggy version.

4.3 Problem Formalization

The key challenge is to generate statement instance matchings. We use labels ℓ and t to denote statements in the two respective implementations. Due to implementation differences, an instance in one execution can match with multiple instances in the other execution.

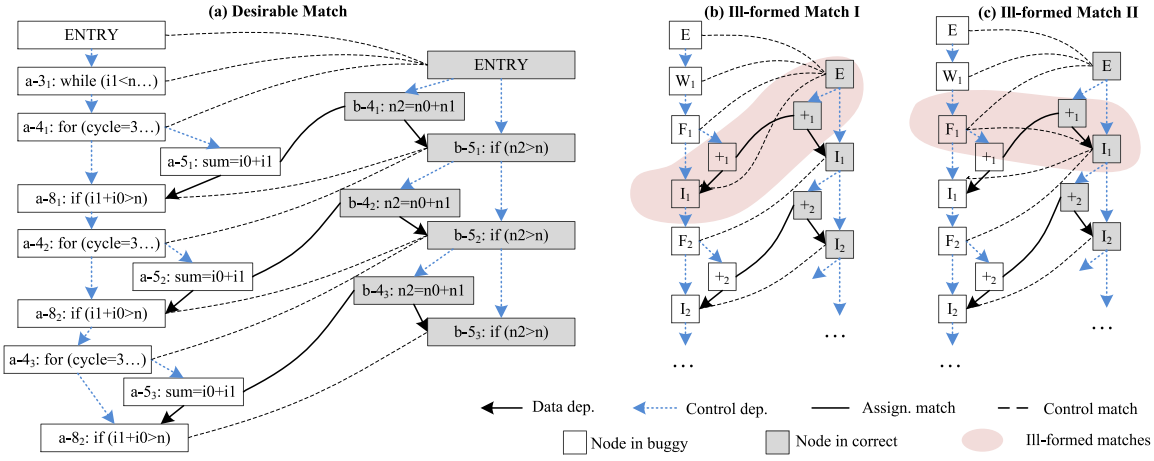


Figure 4.6.: Instance matching for the example in Fig. 4.1. The nodes in grey are from the correct implementation. In (b) and (c), node 'W_1' stands for the first instance of the while loop in (a). Similarly, 'F', 'I', '+' nodes in (b) an (c) stand for the for loop, if conditions, and the addition operations

Intuitively, if two assignment instances match, their symbolic expressions should be equivalent. Furthermore, their control dependencies need to match. It does not mean the corresponding comparison expressions (at the control dependence predicates) need to be equivalent. In fact they are often different. Hence, we ignore the expressions in predicates, treating them as place holders. We match the label instances of these predicates based on the matchings of the assignments control dependent on the predicates and a set of *well-formedness* constraints defined in Fig. 4.5. Rule [WF-SYM] denotes that if the matching is for assignments, the symbolic expressions must be equivalent. Rule [WF-CD] means that if two instances ℓ_i and t_j match, a (transitive) dynamic control dependence of ℓ_i/t_j must match with a (transitive) control

dependence of t_j/ℓ_i. Rule [WF-X] indicates that if ℓ_i and t_j match, there must not be another match ℓ'_k and t'_l such that ℓ_i is dependent on ℓ'_k and t'_l is dependent on t_j, or vice versa. Otherwise, a cycle of dependence is formed, which is impossible in program semantics. Note that matching of transitive data dependencies is already implicitly enforced by the symbolic equivalence in [WF-SYM], because two symbolic expressions are equivalent means that their computations (i.e., data slices) are equivalent.

Example. Fig. 4.6 shows part of the executions from Fig. 4.3 in their dependence graph view. The lines across executions denote matches. Figure (a) shows matchings satisfying the well-formedness constraints. To satisfy a-$5_1 \leftrightarrow$ b-4_1, their dynamic control dependencies need to match (Rule [WF-CD]). The only legitimate matchings are $ENTRY \leftrightarrow ENTRY$, a-$3_1 \leftrightarrow ENTRY$ and a-$4_1 \leftrightarrow ENTRY$. To satisfy the second assignment matching a-$5_2 \leftrightarrow$ b-4_2, the DCDs of a-5_2, including $ENTRY$, a-3_1, a-4_1, a-8_1 and a-4_2, should match with those of b-4_2, including $ENTRY$ and b-5_1.

Figures (b) and (c) show two options. In (b), $I_1 \leftrightarrow E$ (i.e. a-$8_1 \leftrightarrow ENTRY$). However, this matching and the assignment matching $+_1 \leftrightarrow +_1$ together violate Rule [WF-X], because of $+_1 \rightsquigarrow I_1$ on the left (line 8 depends on line 7 and then line 5 according to Fig. 4.1(a)) and $ENTRY \rightsquigarrow +_1$ on the right. Since the match edges are bi-directional, the four edges in the shaded region form a cycle. Similarly, (c) shows another ill-formed matching in which $F_1 \leftrightarrow I_1$ induces a cycle. The only legitimate matching is the one shown in figure (a), in which a-$8_1 \leftrightarrow$ b-5_1 and a-$4_2 \leftrightarrow$ b-5_1. □

Since one execution is buggy, total matching is impossible. Our goal is hence to maximize the number of matches. We reduce the problem to a *partial maximum satisfiability* (PMAX-SAT) problem. Given an UNSAT conjunction of clauses, the *maximum satisfiability* (MAX-SAT) problem aims to generate assignments to variables that maximizes the number of clauses that are satisfied. PMAX-SAT is an extension of MAX-SAT, which aims to ensure the satisfiability of a subset of clauses while maximizing the satisfiability of the remaining clauses. In our context, we want to maximize the number of assignment matchings while assuring the well-formedness constraints are satisfied. *We consider assignments more essential than predicates*

because the symbolic expressions of predicates are often quite different across programs even when they serve the same purpose. Our problem statement is hence formulated as follows.

Definition 4.3.1 *Given two executions, let $\ell_i \leftrightarrow t_j$ be a boolean function for each pair of assignment statement instances denoted by ℓ_i and t_j, with $\ell_i \leftrightarrow t_j = 1$ meaning they match.*

$$F = \boxed{\bigwedge_{\forall \neg ispred(\ell_i),\ \neg ispred(t_j)} \ell_i \leftrightarrow t_j}^{(1)} \ \wedge \ \boxed{C_{WF\text{-}SYM} \wedge C_{WF\text{-}CD} \wedge C_{WF\text{-}X}}^{(2)}$$

, with $C_{WF\text{-}SYM}$, $C_{WF\text{-}CD}$ and $C_{WF\text{-}X}$ the instantiations of well-formedness constraints using the relevant label instances. Our goal is to solve F while ensuring part (2) must be satisfied and maximizing the satisfiability of part (1).

PMAX-SAT is NP-hard. The formula has quantifiers and is cubic to the execution length. Solving it is prohibitively expensive.

4.4 Design

The design of APEX consists of three phases and features an approximate solution to the statement instance matching (PMAX-SAT) problem.

In phase (1), an iterative matching algorithm is applied. In each iteration, sequence alignment is used to match the symbolic expression traces. APEX then matches the dynamic control dependencies of the matched expressions and checks well-formedness. In the following iterations, APEX repeats the same procedure to match the residues, until no more matches can be identified. This is to handle statement reordering as exemplified in Fig. 4.2. In particular, in the first round, it matches the `S1` sequences. Then in the second round, it matches the `S2` sequences.

In phase (2), the (bug related) residues are further aligned (not matched) based solely on control structure, without requiring the symbolic expressions to be equivalent. Particularly, APEX summarizes all the matches identified in the previous phase to

generate a matching at the statement level (not the instance level). Intuitively, this statement level matching identifies the common sub-program of the two versions. We then leverage the statement mapping to identify the entries that are supposed to match but their symbolic expressions are different. These entries are likely bug related. Note aligning these entries allows us to not only identify buggy behavior, but also suggest the corresponding correct behavior.

In phase (3), a dynamic comparative dependence graph is constructed and the comparative slice is computed to generate the bug report.

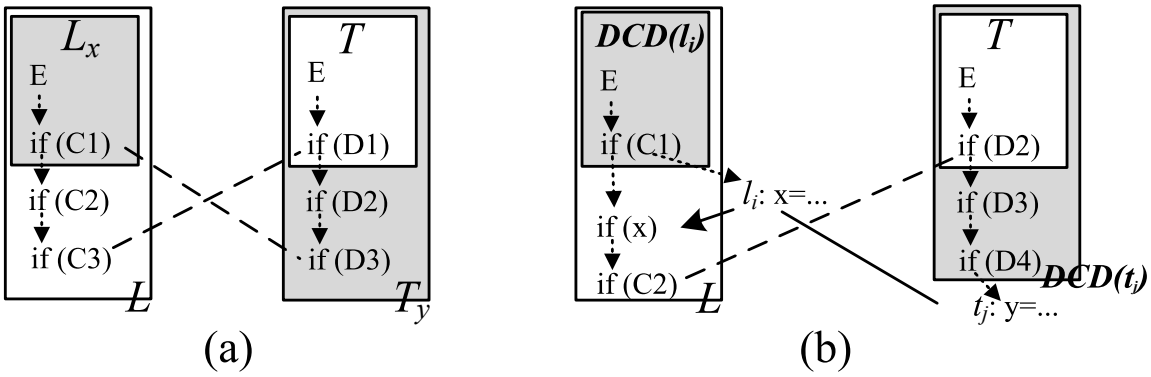


Figure 4.7.: Cycles in matchings. Boxes denote control deps

4.4.1 Phase (1): Iterative Instance Matching

The matching procedure in this phase is iterative. In each round, we first extend sequence alignment to match the symbolic expression sequences. Two expressions can be matched if they are equivalent. APEX then traverses the matched expression pairs in the generated common sub-sequence to match their dynamic control dependencies (DCDs) by the well-formedness constraints (not by the symbolic equivalence of predicates). Due to Type II differences (e.g. control structure differences), it is often difficult to match a predicate uniquely to another predicate in the other version. We hence construct matchings between predicate sequences. Such matchings may be

coarse-grained at the beginning (e.g. $E\text{-}3_1\text{-}4_1 \leftrightarrow E\text{-}5_1$). They are gradually refined (e.g. the previous matching becomes $E\text{-}3_1 \leftrightarrow E$ and $4_1 \leftrightarrow 5_1$).

$LabelInstSeq\ L, T := \overline{\ell_i}|\overline{t_j}$ $\quad InstMap\ := \mathcal{P}(LabelInstSeq \times LabelInstSeq)$ $\quad SeqAlignment\ \mathbb{C} := \overline{\langle \ell_i, t_j \rangle}$
$InstMap\ \mathbb{I}$: mappings between dynamic control dep. $\quad InstMap\ \mathbb{V}$:the resulting instance matchings $\quad LabelInstSeq\ DCD(\ell_i)$: dynamic control dep. of ℓ_i

$\textsc{wellformed}(L, T, \mathbb{I})$ determines if $L \leftrightarrow T$ is a well-formed matching.

$$\textsc{wellformed}(L, T, \mathbb{I}) = \begin{cases} false & \exists L_x, T_y,\ L_x \leftrightarrow T_y \in \mathbb{I} \wedge ((L_x \subset L \wedge T \subset T_y)\ \vee\ (L \subset L_x \wedge T_y \subset T)) \\ false & \exists \ell_i, t_j \neg ispred(\ell_i) \wedge \neg ispred(t_j) \wedge \ell_i \leftrightarrow t_j \in \mathbb{V} \wedge DCD(\ell_i) \leftrightarrow DCD(t_j) \in \mathbb{I} \wedge \\ & \quad DCD(\ell_i) \subset L \wedge T \subset DCD(t_j) \wedge \ell_i \rightsquigarrow \textsc{last}(L) \\ false & \exists \ell_i, t_j \neg ispred(\ell_i) \wedge \neg ispred(t_j) \wedge \ell_i \leftrightarrow t_j \in \mathbb{V} \wedge DCD(\ell_i) \leftrightarrow DCD(t_j) \in \mathbb{I} \wedge \\ & \quad DCD(t_j) \subset T \wedge L \subset DCD(\ell_i) \wedge t_j \rightsquigarrow \textsc{last}(T) \\ true & otherwise \end{cases}$$

$\textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j)$ splits all the statement instance matchings based on the single instance matching $\ell_i \leftrightarrow t_j$.

$$\textsc{split}(L \leftrightarrow T \cup \mathbb{V}, \ell_i \leftrightarrow t_j) = \begin{cases} \{L \leftrightarrow T\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & \ell_i \notin L \vee t_j \notin T \\ \{L_1\text{-}\ell_i \leftrightarrow T_1\text{-}t_j, L_2 \leftrightarrow T_2\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & L \equiv L_1\text{-}\ell_i\text{-}L_2 \wedge T \equiv T_1\text{-}t_j\text{-}T_2 \\ \{L_1\text{-}\ell_i \leftrightarrow T_1\text{-}t_j, L_2 \leftrightarrow t_j\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & L \equiv L_1\text{-}\ell_i\text{-}L_2 \wedge T \equiv T_1\text{-}t_j \\ \{L_1\text{-}\ell_i \leftrightarrow T_1\text{-}t_j, \ell_i \leftrightarrow T_2\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & L \equiv L_1\text{-}\ell_i \wedge T \equiv T_1\text{-}t_j\text{-}T_2 \end{cases}$$

$\textsc{maxpref}(L_1, L, T_1, T, \mathbb{I})$ determines if L_1 and T_1 are the maximum prefixes of L and T that are also shared by some previously matched pairs in $\mathbb{I}$.

$$\textsc{maxpref}(L_1, L, T_1, T, \mathbb{I}) = \begin{cases} true & L_1 \subset L \wedge T_1 \subset T\ \wedge\ \exists L', T',\ (L' \leftrightarrow T' \in \mathbb{I}\ \wedge\ L_1 \subset L'\ \wedge\ T_1 \subset T') \wedge \\ & \neg \exists L_x, T_y, L'', T'',\ (L_1 \subset L_x \subseteq L\ \wedge\ T_1 \subset T_y \subseteq T \wedge\ L_x \subset L'' \\ & \wedge\ T_x \subset T''\ \wedge\ L'' \leftrightarrow T'' \in \mathbb{I}) \\ false & otherwise \end{cases}$$

$\mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T$: the cross product of the current instance matching $\mathbb{V}$ with a new control dep. matching $L \leftrightarrow T$, which may introduce new matchings.

[C-NEW]
$\mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T = \mathbb{V} \cup \{L \leftrightarrow T\}$, if $\neg\exists L_1 \neq nil, T_1 \neq nil,\ \textsc{maxpref}(L_1, L, T_1, T, \mathbb{I})$

[C-DUP]
$\mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T = \mathbb{V}$, if $\textsc{maxpref}(L, L, T, T, \mathbb{I})$

[C-SPLIT]
$\mathbb{V} \otimes_{\mathbb{I}} L_1\text{-}L_2 \leftrightarrow T_1\text{-}T_2 = \textsc{split}(\mathbb{V}, \textsc{last}(L_1) \leftrightarrow \textsc{last}(T_1)) \cup \{L_2 \leftrightarrow T_2\}$, if $\textsc{maxpref}(L_1, L_1\text{-}L_2, T_1, T_1\text{-}T_2, \mathbb{I})$

[C-TAILA]
$\mathbb{V} \otimes_{\mathbb{I}} L_1\text{-}L_2 \leftrightarrow T = \textsc{split}(\mathbb{V}, \textsc{last}(L_1) \leftrightarrow \textsc{last}(T)) \cup \{L_2 \leftrightarrow \textsc{last}(T)\}$, if $\textsc{maxpref}(L_1, L_1\text{-}L_2, T, T, \mathbb{I})$

[C-TAILB]
$\mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T_1\text{-}T_2 = \textsc{split}(\mathbb{V}, \textsc{last}(L) \leftrightarrow \textsc{last}(T_1)) \cup \{\textsc{last}(L) \leftrightarrow T_2\}$, if $\textsc{maxpref}(L, L, T_1, T_1\text{-}T_2, \mathbb{I})$

[UNMATCHED-EXPR]

$$\frac{\neg \textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I})}{\langle \ell_i, t_j \rangle \cdot \mathbb{C}, \mathbb{I}, \mathbb{V}\ \longrightarrow\ \mathbb{C}, \mathbb{I}, \mathbb{V}}$$

[MATCHED-EXPR]

$$\frac{\textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I})\quad \mathbb{I}' = \mathbb{I} \cup DCD(\ell_i) \leftrightarrow DCD(t_j) \quad \mathbb{V}' = (\mathbb{V} \otimes_{\mathbb{I}} DCD(\ell_i) \leftrightarrow DCD(t_j)) \cup \ell_i \leftrightarrow t_j}{\langle \ell_i, t_j \rangle \cdot \mathbb{C}, \mathbb{I}, \mathbb{V}\ \longrightarrow\ \mathbb{C}, \mathbb{I}', \mathbb{V}'}$$

Figure 4.8.: Instance matching rules. Symbol '-' in $L_1\text{-}L_2$ means concatenation

Well-formed Matching of Control Dependencies. Next we focus on explaining how APEX matches the DCDs and checks well-formedness. The algorithm traverses

the longest common sub-sequence $\mathbb{C}$ produced by sequence alignment, trying to match the DCDs of each expression pair.

The traversal procedure is described by the two term rewriting rules on the bottom of Fig. 4.8. The symbols and functions used in the rules are defined on the top. The configuration of evaluation consists of the common sub-sequence $\mathbb{C}$ containing a sequence of label instance pairs, the DCD mappings $\mathbb{I}$ used to facilitate well-formedness checks, and the instance mappings $\mathbb{V}$.

The traversal is driven by $\mathbb{C}$. Rule [UNMATCHED-EXPR] is to handle a pair of expressions in $\mathbb{C}$ that were matched by sequence alignment but violate the well-formedness constraints. Function $wellformed()$ determines if matching two label instance sequences L and T (denoting DCDs) causes any well-formedness violations. According to its definition in Fig. 4.8, it detects three kinds of violations. In the first case, if there is already a mapping $L_x \leftrightarrow T_y$ admitted to $\mathbb{I}$ in the past, and L_x is a prefix of L and T a prefix of T_y (or vice versa), there must be a cycle similar to Fig. 4.7 (a). The $\subset$ operator means prefix here. In Fig. 4.7 (a), there must be some matching between a statement instance in $L \ominus L_x$ (i.e., in L but not L_x) and some statement instance in T (e.g. `if(C3)` $\leftrightarrow$ `if(D1)`). Similarly, there must be some matching between an instance in $T_y \ominus T$ and some instance in L_x (e.g. `if(C1)` $\leftrightarrow$ `if(D3)`). Also because $L \ominus L_x$ must be control dependent on L_x (all these are valid control dependencies) and $T_y \ominus T$ control dependent on T. A cycle of dependence is formed, violating [WF-X] (Section 4.3).

As illustrated in Fig. 4.7 (b), the second case describes that there is an existing expression matching $\ell_i \leftrightarrow t_j \in \mathbb{V}$ (whose DCD matching is hence in $\mathbb{I}$ which we will explain later), and the dynamic control dependence of ℓ_i, $DCD(\ell_i)$, is a prefix of L, and T is a prefix of $DCD(t_j)$. If $L \leftrightarrow T$, the last entry of L (i.e. `if(C2)`) must match with some entry in T (e.g. `if(D2)`). However, the matching becomes illegal if some predicate in $L \ominus DCD(\ell_i)$ (e.g. `if(x)`) is dependent on ℓ_i (i.e. `x=...`), because a cycle ℓ_i-`if(x)`-`if(C2)`-`if(D2)`... -t_j-ℓ_i is formed. The third case is symmetric.

Rule [MATCHED-EXPR] handles the case that the matching of the DCDs of an expression pair is well-formed. The control dependence mapping set $\mathbb{I}$ is updated by adding the control dependencies of the matched expressions. The instance mapping set $\mathbb{V}$ is also updated so that some previous matched statement sets can be broken down to smaller (matched) subsets. Note that smaller matched sets mean finer granularity in matching. This is done by a *cross-product* operation between the control dependence mapping (of the symbolic expressions) and the current instance mapping set. The expression matching is also added to the result set.

The cross-product operation $\mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T$ may introduce new mappings and split an existing mapping into multiple mappings. If L and T do not share any common prefixes with any existing control dependence mapping, $L \leftrightarrow T$ is added to $\mathbb{V}$ (Rule [C-NEW]). If they do share common prefixes with some existing mappings, which suggests that the existing mappings are too coarse grained, the existing mappings are hence refined. The mapping with the maximum common prefixes is identified through the $maxpref()$ primitive. Assume the maximum common prefixes are L_1 and T_1, the existing mappings are split if they include the mapping $\text{LAST}(L_1) \leftrightarrow \text{LAST}(T_1)$ through the *split*() primitive ([C-SPLIT]). For example, assume a new mapping $E\text{-}3_1\text{-}6_1 \leftrightarrow E$ shares common prefix with an existing mapping $E\text{-}3_1\text{-}4_1 \leftrightarrow E\text{-}5_1$. The existing mapping is split by the last entries of the common prefixes $3_1 \leftrightarrow E$, resulting in two smaller mappings $E\text{-}3_1 \leftrightarrow E$ and $4_1 \leftrightarrow 5_1$. The suffices of L and T are also added to $\mathbb{V}$ as a new mapping. Rules [C-TAILA] and [C-TAILB] handle the corner cases that the maximum common prefix is one of L and T, in which the non-empty suffix is matched with the last entry of the prefix. This is the only legal mapping without introducing cycles.

□ *Example.* Table. 4.1 shows how the algorithm works on the traces in Fig. 4.3. The sequence alignment generates the initial $\mathbb{C}$ that identifies the longest sequence of equivalent symbolic expression pairs, as shown in the first row. Each row of the table represents one step of the algorithm that processes and removes a pair from $\mathbb{C}$. Columns 3 and 4 show the DCD mappings and instance mappings, after the rules

Table 4.1.: Applying the algorithm in Fig. 4.8 to traces in Fig. 4.3

#	$\mathbb{C}$	$\mathbb{I}$	$\mathbb{V}$	rules applied
1	$\{\langle 5_1, 4_1\rangle, \langle 5_2, 4_2\rangle, \langle 5_3, 4_3\rangle, \langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E$	E-3_1-$4_1 \leftrightarrow E, 5_1 \leftrightarrow 4_1$	[M-EXPR,C-NEW]
2	$\{\langle 5_2, 4_2\rangle, \langle 5_3, 4_3\rangle, \langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E$, E-3_1-4_1-8_1-$4_2 \leftrightarrow E$-5_1	E-3_1-$4_1 \leftrightarrow E$, 8_1-$4_2 \leftrightarrow 5_1, 5_1 \leftrightarrow 4_1, 5_2 \leftrightarrow 4_2$	[M-EXPR,C-SPLIT]
3	$\{\langle 5_3, 4_3\rangle, \langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E$, E-3_1-4_1-8_1-$4_2 \leftrightarrow E$-5_1, E-3_1-4_1-8_1-4_2-8_2-$4_3 \leftrightarrow E$-5_1-5_2	E-3_1-$4_1 \leftrightarrow E$, 8_1-$4_2 \leftrightarrow 5_1, 8_2$-$4_3 \leftrightarrow 5_2$, $5_1 \leftrightarrow 4_1, 5_2 \leftrightarrow 4_2, 5_3 \leftrightarrow 4_3$	[M-EXPR,C-SPLIT] [M-EXPR,C-SPLIT]
4	$\{\langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E$, E-3_1-4_1-8_1-$4_2 \leftrightarrow E$-5_1, E-3_1-4_1-8_1-4_2-8_2-$4_3 \leftrightarrow E$-5_1-5_2, E-3_1-$13_1 \leftrightarrow E$-5_1-5_2-7_3	E-$3_1 \leftrightarrow E$, $4_1 \leftrightarrow E$, 8_1-$4_2 \leftrightarrow 5_1, 8_2$-$4_3 \leftrightarrow 5_2$, $5_1 \leftrightarrow 4_1, 5_2 \leftrightarrow 4_2, 5_3 \leftrightarrow 4_3, 13_1 \leftrightarrow 5_1$-$5_2$-$7_3, 14_1 \leftrightarrow 8_1$	[M-EXPR,C-SPLIT]

specified in the last column are applied. At step one, matching the DCDs of 5_1 and 4_1 is well-formed. As such, the DCDs are added to both $\mathbb{I}$ and $\mathbb{V}$, and $5_1 \leftrightarrow 4_1$ is added to $\mathbb{V}$. At step two, matching the DCDs of 5_2 and 4_2 is also well-formed. Since $DCD(5_2) = E\text{-}3_1\text{-}4_1\text{-}8_1\text{-}4_2$ and $DCD(4_2) = E\text{-}5_1$, the cross product of their matching with $\mathbb{V}$ identifies that an existing mapping $E\text{-}3_1\text{-}4_1 \leftrightarrow E$ has the maximum common prefix with the new mapping. Hence the suffix mapping $8_1\text{-}4_2 \leftrightarrow 5_1$ is added. Step three is similar. At step four, the DCD matching of 14_1 and 8_1 is well-formed. The cross product of their DCD matching $E\text{-}3_1\text{-}13_1 \leftrightarrow E\text{-}5_1\text{-}5_2\text{-}7_3$ with $\mathbb{V}$ not only induces the addition of $13_1 \leftrightarrow 5_1\text{-}5_2\text{-}7_3$ to $\mathbb{V}$, but also splits $E\text{-}3_1\text{-}4_1 \leftrightarrow E$ to $E\text{-}3_1 \leftrightarrow E$ and $4_1 \leftrightarrow E$ by the *split*() primitive. □

To handle implementation differences such as statement reordering (e.g. Fig. 4.2), APEX applies the aforementioned procedure iteratively until no more matchings can be found. In particular, after each round, the trace entries corresponding to the matched symbolic expressions that pass the well-formedness checks (i.e., those admitted by Rule [MATCHED-EXPR]) are removed from the traces. Note that the predicate instances representing control dependencies are never removed even they are matched. This is to support well-formedness checks for the matchings in the following rounds. The same matching algorithm is then applied to the remaining traces. For example in Fig. 4.2 (a), all the entries corresponding to `S1` are removed after the first round but the loop predicate instances are retained, which allows us to perform well-formed matching of `S2` entries in the next round. As a result, the loop predicate on the left is correctly matched with the two loop predicates on the right.

Finally, the results in $\mathbb{V}$ denote the matchings between statement instances in the two versions. They correspond to the common bug-free behavior.

4.4.2 Phase (2): Residue Alignment

There are statement instances that cannot be matched, which are likely bug related. They may belong to statements unique to an implementation, or statements with some

but not all their instances matched. For the latter case, it is highly desirable to *align* the unmatched instances of those statements such that it becomes clear why they do not match while they should have. This is very important for bug explanation. APEX further aligns these unmatched instances. It does so by generating a statement level mapping $\mathbb{M}$ between the two versions, from the matching results in the previous phase. Particularly, relation $\mathbb{M} : \mathcal{P}(\mathcal{P}(Label_a) \times \mathcal{P}(Label_b))$ indicates a set of statements in program a matches with a set of statements in b. It is generated by the following equation.

$$\langle L, T\rangle \in \mathbb{V} \cup \mathbb{I} \implies \langle \textsc{set}(L), \textsc{set}(T)\rangle \in \mathbb{M}$$

Function SET() turns a sequence of label instances to a set of labels (e.g. $\textsc{set}(E\text{-}3_1\text{-}4_1) = \{E, 3, 4\}$).

For the example in Fig. 4.1, $\mathbb{M} = \{\langle\{3, 4, 8\}, \{5\}\rangle, \langle\{5\}, \{4\}\rangle, \langle\{14\}, \{8\}\rangle ...\}$. It essentially denotes a common sub-program of the two as shown in Fig. 4.1.

Then APEX traverses the residue traces that contain the remaining unmatched symbolic expressions and all the predicate instances, and aligns trace entries based on the common sub-program $\mathbb{M}$ and well-formedness.

The alignment algorithm takes as input the two residue traces $\mathbb{T}_a$ and $\mathbb{T}_b$. Each trace entry is a triple consisting of the label instance, the symbolic expression se and the concrete value v. It traverses the buggy trace and looks for alignment for each instance. Basically, two instances are aligned if such alignment is compatible with the statement mappings $\mathbb{M}$ and aligning their dynamic control dependencies is well-formed. Note that they are aligned but not matched. They have different (symbolic) values. The green nodes in Fig. 4.4 are such examples.

The rules are presented in Fig. 4.9. In Rule [UNALIGN-PRED], a predicate instance ℓ_i is discarded if there is no alignment. Note that since ℓ_i and t_j are predicates, they are concatenated to the dynamic control dependencies (e.g. $DCD(\ell_i)$) for the well-formedness check. If multiple well-formed alignments exist, the first one is selected and

$$\frac{ispred(\ell_i) \quad \neg\exists\langle t_j, se1, v1\rangle \in \mathbb{T}_b,\ \langle \textsc{set}(DCD(\ell_i)\text{-}\ell_i), \textsc{set}(DCD(t_j)\text{-}t_j)\rangle \in \mathbb{M} \wedge \textsc{wellformed}(DCD(\ell_i)\text{-}\ell_i, DCD(t_j)\text{-}t_j, \mathbb{I})}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V}} \quad \text{[UNALIGN-PRED]}$$

$$\frac{\begin{array}{c} ispred(\ell_i) \quad \mathbb{T}_b = \mathbb{T}'_b \cdot \langle t_j, se1, v1\rangle \cdot \mathbb{T}''_b \quad \langle \textsc{set}(DCD(\ell_i)\text{-}\ell_i), \textsc{set}(DCD(t_j)\text{-}t_j)\rangle \in \mathbb{M} \\ \textsc{wellformed}(DCD(\ell_i)\text{-}\ell_i, DCD(t_j)\text{-}t_j, \mathbb{I}) \neg\exists\langle t'_k, se2, v2\rangle \in \mathbb{T}'_b, \\ \langle \textsc{set}(DCD(\ell_i)\text{-}\ell_i), \textsc{set}(DCD(t'_k)\text{-}t'_k)\rangle \in \mathbb{M} \wedge \\ \textsc{wellformed}(DCD(\ell_i)\text{-}\ell_i, DCD(t'_k)\text{-}t'_k, \mathbb{I}) \\ \mathbb{I}' = \mathbb{I} \cup DCD(\ell_i)\text{-}\ell_i \leftrightarrow DCD(t_j)\text{-}t_j \quad \mathbb{V}' = \mathbb{V} \otimes_{\mathbb{I}} DCD(\ell_i)\text{-}\ell_i \leftrightarrow DCD(t_j)\text{-}t_j \end{array}}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}''_b, \mathbb{I}', \mathbb{V}'} \quad \text{[ALIGN-PRED]}$$

$$\frac{\neg ispred(\ell_i) \quad \neg\exists\langle t_j, se1, v1\rangle \in \mathbb{T}_b,\ \langle \ell, t\rangle \in \mathbb{M} \wedge \textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I})}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V}} \quad \text{[UNALIGN-ASSIGN]}$$

$$\frac{\begin{array}{c} \neg ispred(\ell_i) \quad \mathbb{T}_b = \mathbb{T}'_b \cdot \langle t_j, se1, v1\rangle \cdot \mathbb{T}''_b \quad \langle \ell, t\rangle \in \mathbb{M} \\ \textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I}) \\ \neg\exists\langle t'_k, se2, v2\rangle \in \mathbb{T}'_b,\ \langle \ell, t'\rangle \in \mathbb{M} \wedge \textsc{wellformed}(DCD(\ell_i), DCD(t'_k), \mathbb{I}) \\ \mathbb{I}' = \mathbb{I} \cup DCD(\ell_i) \leftrightarrow DCD(t_j) \quad \mathbb{V}' = (\mathbb{V} \otimes_{\mathbb{I}} DCD(\ell_i) \leftrightarrow DCD(t_j)) \cup \ell_i \leftrightarrow t_j \end{array}}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}''_b, \mathbb{I}', \mathbb{V}'} \quad \text{[ALIGN-ASSIGN]}$$

Figure 4.9.: Residue alignment

$\mathbb{I}$ and $\mathbb{V}$ are updated accordingly (Rule [ALIGN-PRED]). The alignment of assignment instances is similar (Rules [UNALIGN-ASSIGN] and [ALIGN-ASSIGN]).

□ *Example.* Table 4.2 presents an example. The first two columns show the residue traces for the fibonacci executions. In the buggy run, since the value in 8_5 is incorrect, 9_1 is incorrectly executed and the loop is terminated. Outside the loop, the false branch of 13_2 is taken and the outer loop is also terminated. In the correct run, the predicate at 5_6 takes the false branch and one more round of fibonacci computation is performed until the true branch of 5_7 is taken and the loop is terminated. The next two columns show the dynamic control dependencies of the first trace entries.

In the first step, the alignment $8_5 \leftrightarrow 5_6$ is added to $\mathbb{V}$ as the statement mapping $\langle\{E, 3, 4, 8\}, \{E, 5\}\rangle \in \mathbb{M}$ and the alignment of control dependencies is well-formed. It corresponds to the first green node in Fig. 4.4. Next, no alignment is found for 9_1 (i.e. red node in Fig. 4.4). In the third step, 13_2 and 7_6 are aligned (despite their different branch outcomes), corresponding to the second green node. □

Table 4.2.: Applying the algorithm in Fig. 4.9 to the residue traces in the fibonacci executions. Let $L = E\text{-}3_1\text{-}3_2\text{-}4_4\text{-}8_3\text{-}4_5\text{-}8_4\text{-}4_6$, $T = E\text{-}5_1\text{-}5_2\text{-}5_3\text{-}5_4\text{-}5_5$, $\mathbb{I}_0 = \{...E\text{-}3_1\text{-}13_1 \leftrightarrow E\text{-}5_1\text{-}5_2\text{-}7_3, ..., L \leftrightarrow T\}$,$\mathbb{V}_0 = \{E\text{-}3_1 \leftrightarrow E, 3_2 \leftrightarrow 5_1\text{-}5_2\text{-}5_3, ..., 8_4\text{-}4_6 \leftrightarrow 5_5, 5_5 \leftrightarrow 4_5\}$. [A-P] stands for [ALIGN-PRED]

$\mathbb{T}_a$	$\mathbb{T}_b$	DCD_a	DCD_b	$\mathbb{I}$	$\mathbb{V}$	rules applied
$8_5 \cdot 9_1 \cdot 13_2 \cdot 3_3 \cdot 16_1$	$5_6 \cdot 7_6 \cdot 8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	L	T	$\mathbb{I}_0 \cup \{L\text{-}8_5 \leftrightarrow T\text{-}5_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6\}$	[A-P]
$9_1 \cdot 13_2 \cdot 3_3 \cdot 16_1$	$7_6 \cdot 8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	$L\text{-}8_5$	$T\text{-}5_6$	$\mathbb{I}_0 \cup \{L\text{-}8_5 \leftrightarrow T\text{-}5_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6\}$	[U-A]
$13_2 \cdot 3_3 \cdot 16_1$	$7_6 \cdot 8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	$E\text{-}3_1\text{-}3_2$	$T\text{-}5_6$	$\mathbb{I}_0 \cup \{L\text{-}8_5 \leftrightarrow T\text{-}5_6,$ $E\text{-}3_1\text{-}3_2\text{-}13_2 \leftrightarrow T\text{-}5_6\text{-}7_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6, 13_2 \leftrightarrow 5_4\text{-}5_5\text{-}5_6\text{-}7_6\}$	[A-P]
$3_3 \cdot 16_1$	$8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	$E\text{-}3_1\text{-}3_2$	$T\text{-}5_6\text{-}7_6$	$\mathbb{I}_0 \cup \{L\text{-}8_5 \leftrightarrow T\text{-}5_6,$ $E\text{-}3_1\text{-}3_2\text{-}13_2 \leftrightarrow T\text{-}5_6\text{-}7_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6, 13_2 \leftrightarrow 5_4\text{-}5_5\text{-}5_6\text{-}7_6\}$	[U-A]
...	...	...	...	...	...	

4.4.3 Phase (3): Comparative Dependence Graph Construction, Slicing, and Feedback Generation

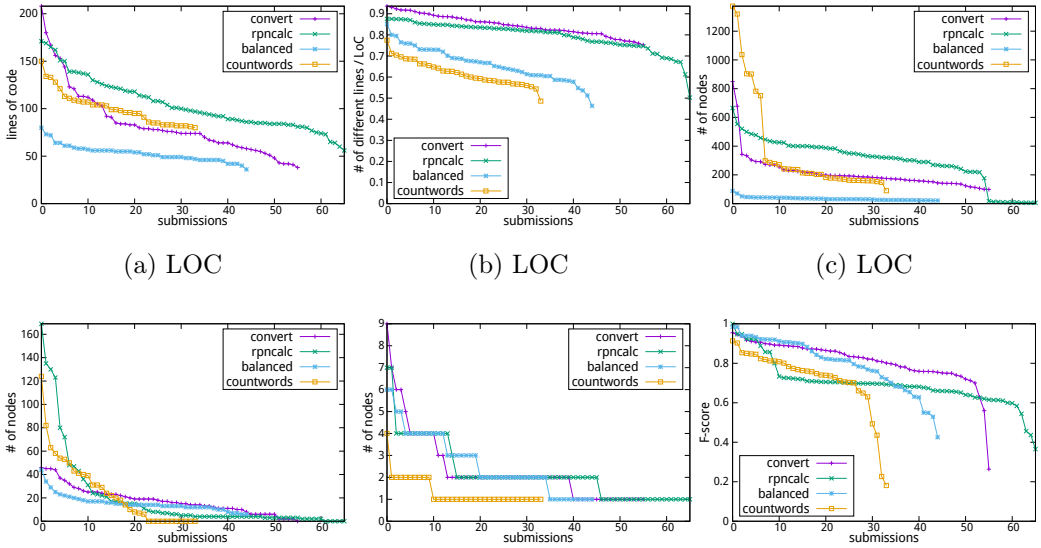
APEX generates the DCDG from the matching and alignment results. Fig. 4.4 represents an example graph. It further computes a *comparative slice* from the graph. The slicing criterion consists of the instances that emit the different outputs. The slice captures the internal differences that caused the output differences. It is computed by graph traversal, which starts from the criterion, going backward along dependence edges. If a plain node (for matched instances) is reached, no traversal is beyond the node. Our tool follows a set of rules to generate the bug report from a slice. For example, a pair of aligned but unmatched assignment instances $\ell_i \leftrightarrow t_j$, is translated to "*the value at* ℓ_i *should have been* $v(t_j)$ *instead of* $v(\ell_i)$". The report for the fibonacci bug can be found at the end of Section 4.2. Details are elided.

4.5 Implementation and Evaluation

The tracing component of APEX that collects symbolic and concrete traces is implemented using LLVM. The SMT solver used is Z3 [92]. The rest is implemented in Python. The experiments were conducted on an Intel Core i7 machine running Arch Linux 3.17.1 with 16GB RAM. All the benchmarks, the failure inducing inputs, and the bug reports by APEX are available on the project site [93].

4.5.1 Experiment with Real Student Submissions

We have acquired 4 programming assignments from a recent programming course at the authors' institute: `convert` turns a number with one radix into another radix; `rpncalc` evaluates a postfix expression using a stack; `balanced` checks if the input string has a valid nesting of parentheses and brackets; `countwords` counts the frequency of words. The number of buggy versions ranges from 33-65 for each submission. The total number of buggy submissions is 205.



(a) LOC (b) LOC (c) LOC

(d) LOC (e) LOC (f) LOC

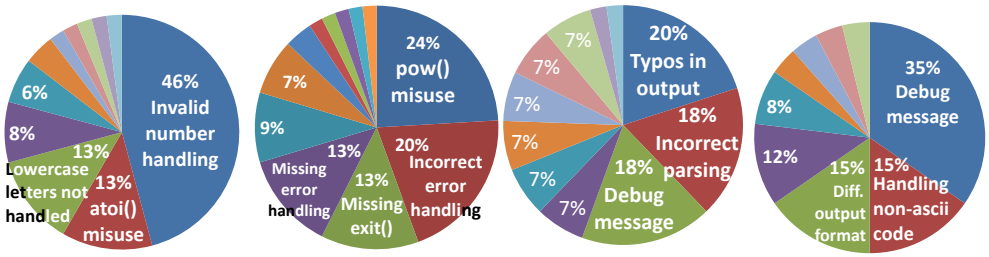
Figure 4.10.: Student submission results. On each figure, the submissions are sorted by the Y-axis values.

For each buggy version, we have the failure inducing input and the patched version (submitted later by the students). For each assignments, we have the instructor's solution. We applied APEX to each failing run. The results are presented in Fig. 4.10. The execution time ranges from 1 to 20 seconds with most finishing in a few seconds.

From Fig. 4.10a, the submission LOC ranges from 60-210. Fig. 4.10b measures the syntactic differences between the submissions and the instructor's version (i.e. edit distance over LOC sum). Observe that they are substantially different. From Fig. 4.10c, the computed DCDG has 10-1300 nodes. Some have a small DCDG because of the simplicity of the test case (e.g., testing input validation). From Fig. 4.10d, the comparative slices have 2-160 nodes. The large slices usually correspond to cases in which the buggy program has substantially different states from the correct program. However, as shown in Fig. 4.10e, the bug reports are very small. According to our experience with students, succinct bug reports without too much low level details are important for usability. We have a number of methods to reduce bug report size,

including coalescing the repetitive instances (from loops), and avoiding presenting detailed causality in large unmatched code regions, which often occur when the correct program terminates quickly due to invalid inputs but the buggy program goes on as normal, or vice versa. We cross-checked the root causes reported by APEX with the patched versions and found that APEX identifies the correct root causes for 195 out of 205 cases. Here, when we say APEX identifies a root cause, we mean that the root cause is reported as the first entry in the causal explanation just like the example in Section 4.2.

Fig. 4.10f shows the F-score [94] of execution matching, including both assignment and predicate matchings. F-score is a weighted average of precision and recall that reflects the percentage of matching. Here, *precision/recall means the percentage of the statement instances in the failing/passing run that have matches in the other party*, and F-score $F = 2 \cdot \frac{\texttt{precision} \cdot \texttt{recall}}{\texttt{precision} + \texttt{recall}}$. Observe that APEX is able to match a lot of instances for many cases. Some have almost everything matched. These bugs are usually due to typos in outputs.



(a) Convert (b) Convert (c) Convert (d) Convert

Figure 4.11.: Student bug classifications

Student Bugs. To better demonstrate the effectiveness of APEX in identifying root causes and matching executions, we further generate a grading report for each assignment by classifying the bugs based on the root causes. However, the root causes in the bug reports by APEX only contain artifacts from the buggy programs, which are very different from each other. It is hence difficult to classify based on bug reports

directly. Fortunately, APEX has the matchings to the correct version, which is stable across all bugs. We hence classify bugs based on the projection of the root cause in the correct version. Intuitively, with APEX we are able to classify by *the part that is not correctly implemented by the student*. The results are shown in Fig. 4.11. We have the following observations. **(1)** Most bugs fall into a few main categories. For example in `rpncalc`, 24% of bugs are due to the incorrect parameter order of the `pow()` function. The reason is that parameters are stored in the stack order so that they need to be flipped before calling `pow()`. In `convert`, almost half of the students forgot to check if an input character is legal for the radix. Such information is very useful for the instructor as they indicate where to improve. **(2)** Typos in final outputs are very common (e.g. `printf("String not balanced.\n")` versus `printf("String is not balanced.\n")` in `balanced`). For these cases, a simple automatic grading policy that counts the number of passing runs by comparing outputs would give 0 credit. In contrast, APEX would allow partial credit by execution matching (e.g., the F-score). In these cases, the students will get almost full credit. **(3)** APEX missed the root cause for 10 out of 205 cases. We further inspected these cases. Most of them are because the buggy run is so wrong that there are very few matched symbolic expressions to begin with. For example in `rpncalc`, the instructor and most students used predicates whereas two students used table look-up to drive execution like in a compiler frontend. However, the table indexing is wrong. As such, almost the entire sequence of (symbolic) values are wrong. **(4)** Although from the reports many bugs have simple root causes, it does not mean they are easier for APEX as identifying them requires matching the substantially different program structures. There are also subtle bugs. But their number is relatively small.

4.5.2 Experiment with `stackoverflow.com` Programs

To better evaluate applicability, we have collected 15 pairs (buggy vs. correct) of implementations from `stackoverflo- w.com`. They were mainly posted in 2014. The

Table 4.3.: Benchmarks and symbolic expression matching

benchmark	url	LOC	# sym expr	time	# matches
knapsack-1	[95]	56 / 78	69 / 210	4.65s	401
matrix-mult	[96]	53 / 53	421 / 421	9.56s	1534
fibonacci-sum	[90]	35 / 29	22 / 28	0.79s	22
kadane	[97]	43 / 29	22 / 26	0.25s	34
euclid	[98]	23 / 22	17 / 10	0.66s	5
dijkstra	[99]	57 / 64	79 / 76	1.85s	219
mergesort	[100]	47 / 70	135 / 231	8.66s	150
span-tree	[101]	71 / 75	153 / 135	7.60s	1499
floyd	[102]	46 / 47	154 / 173	8.51s	1892
dijkstra-2	[103]	61 / 64	121 / 76	2.00s	303
euler	[104]	44 / 27	110 / 81	21.74s	167
gt_product	[105]	27 / 27	315 / 330	164.05s	866
binarysearch	[106]	25 / 27	32 / 37	1.60s	27
euclid-2	[107]	31 / 21	8 / 10	0.52s	5
knapsack-2	[108]	33 / 42	60 / 109	1.29s	41

benchmarks and their urls are presented in the first two columns of Table 4.3. The benchmarks are named after the algorithms they implement. Each row represents two programs. The sizes of each pair are presented in the third column. The programs are by different programmers. The fourth column presents the size of the symbolic trace, i.e. the number of symbolic expressions for assignments, excluding all simple copies and the assignments that are not data dependent on inputs. The `time` column shows the time taken to match all symbolic expression pairs. This is to prepare for the iterative instance matching algorithm. The last column shows the number of equivalent pairs. Observe that the number of pairs may be much larger than the number of expressions in the individual versions because one expression may be symbolically equivalent to many. Also observe that the time taken is not substantial as APEX uses concrete value comparison to prune the candidate pairs. That is, we only compare symbolic equivalence when two expressions have the same concrete value.

Table 4.4 shows the instance matching results. The `size` column shows the number of LLVM IR statement instances in the execution traces. We have excluded all the copy operations and short-cut the corresponding dependencies for brevity. The "Matched" column shows the number of instances that are matched, and the percentage (e.g. for the `knapsack` case, $80/221 = 36\%$ whereas $80/417 = 19\%$). The A&U column shows those aligned but not matched. The U&U column shows those neither aligned nor matched. The next two columns show the graph and the slice sizes (in nodes). The last column shows the root causes reported by APEX. Symbol '-' means that APEX misses the real root cause. Observe that APEX can align and match roughly half of the instances. It can also align part of the unmatched instances. Those instances are usually closely related to bugs. Depending on the semantic differences, the unaligned and unmatched parts may still be large. For example, 81% of the instances in the correct version of `knapsack` cannot be matched or aligned. This is because the correct execution is much longer. Also observe that most of comparative slices are small, much smaller than the graph sizes. More importantly, in most cases, the root of the slice precisely identifies the real root cause as mentioned in the online bug report. Recall

Table 4.4.: Instance matching. “s.h.” stands for “should have”, “s.n.” for “should not”

benchmark	size	time	# Matched (%/%)	# A&U (%/%)	# U&U(%)		G.size	S.size	root cause
					buggy	correct			
knapsack	221 / 417	0.51s	80(36/19)	0(0/0)	141(64)	337(81)	558	51	s.h. `if (weight<w2 && w1>=weight)` ...
matmult	514 / 514	0.62s	247(48/48)	33(6/6)	179(35)	179(35)	638	312	s.n. `FIRST[c*M+k]` s.h. `FIRST[c*N+k]` at line 25
fibonacci-sum	38 / 42	0.34s	21(55/50)	2(5/4)	15(40)	19(46)	59	19	`if (i1+i0>n)` s.h. `false` s.n. `true` at line 26
kadane	40 / 49	0.34s	26(65/53)	4(10/8)	10(25)	19(39)	59	5	s.h. `if (0<=cumulativeSum)` inside line 15.
euclid	21 / 15	0.32s	6(29/40)	1(5/7)	9(43)	5(33)	21	8	s.n. `b*(a/b)` s.h. `a%b` at line 7.
dijkstra	195 / 150	0.40s	68(35/45)	0(0/0)	125(64)	77(51)	270	18	s.n. `visited[vert]=1` at line 43.
mergesort	178 / 294	0.53s	118(66/40)	17(10/6)	21(12)	117(40)	273	31	s.n. `arr[mid]=arr[start]` s.h. `new_array[i] = arr[high]` at line 31.
span-tree	389 / 324	0.51s	210(54/65)	2(1/1)	177(46)	112(35)	501	6	-
floyd	286 / 351	0.51s	210(73/60)	2(1/1)	177(62)	112(32)	501	16	s.h. `if (path[i][k] != INT_MAX...)` at line 30
dijkstra-2	192 / 150	0.42s	39(20/26)	0(0/0)	82(43)	67(45)	188	22	s.n. `for (...;i<nr_airport;i++)` s.h. `while (cur_vertex != END)` at line 24.
euler1	133 / 91	0.38s	71(53/78)	0(0/0)	50(38)	11(12)	132	61	s.h. `return 0` at line 43.
gt_product	546 / 473	0.80s	318(58/67)	5(1/1)	117(21)	29(6)	469	88	s.n. `product = product * ((int) NUM[j] - 48)` at line 16.
binarysearch	46 / 54	0.36s	21(46/39)	2(4/4)	13(28)	18(33)	54	6	s.n. `retval=0` at line 22.
euclid-2	11 / 15	0.33s	2(18/13)	0(0/0)	4(36)	5(33)	11	2	s.h. `q = (r[0] % r[1])` at line 15
knapsack-2	87 / 174	0.44s	11(13/6)	1(1/1)	53(61)	80(46)	145	45	-

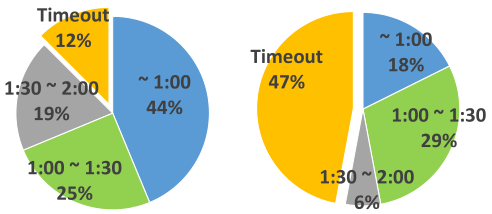
that the root of a slice is the A&U or U&U instances whose parents are matched. Benchmark `matmult` has an exceptionally large slice. That is because the buggy execution has largely corrupted state. An array index computation is wrong such that most values are wrong from the beginning. All such faulty values are part of the slice. Interestingly, APEX can still match and align most of the control structures and part of the computation and it also precisely pinpoints the root cause. Since these unmatched instances belong to a few statements (in loops), the bug report is still very small.

`Mergesort` is an interesting case. The buggy code compares values l from the lower half and h from the higher half of an array and directly swaps the values if h is smaller than l. It uses one loop while the correct code uses four loops. APEX was able to match the control structures and recognize that the buggy code needs an additional array instead of direct swapping. In particular, APEX identifies an unmatched additional array assignment within a matched branch in the correct run. In `dijkstra`, the two implementations are substantially different. They use different values to denote if a node has been visited. Moreover, a loop in the correct version corresponds to two separate loops in the buggy version. APEX was able to match the control structures and correctly explain the bug. In `dijkstra-2`, a nesting loop in one version corresponds to a few consecutive loops in the other. Details can be found at [93].

APEX misses the root cause for two cases: `span-tree` that computes the minimal spanning tree and `knapsack-2`. The reason is that the buggy programs used algorithms different from that used by the correct version. APEX currently does not support matching across algorithms (e.g., bubble sort vs. merge sort). If different algorithms are allowed, we plan to follow the same strategy as in [82], which is to let the instructor provide a set of possible algorithms beforehand. We can also use APEX to match *passing runs*. Algorithmic differences will yield poor matchings in passing runs. We will leave it to future work.

4.5.3 User Study

We have evaluated APEX with 34 undergraduate students who take the C programming course at the authors' institute. We have partitioned the students into 2 groups, one using APEX and the other not. We requested them to implement `convert` in Sec. 4.5.1 in a two-hour lab. Our research questions are: 1. Can APEX help the overall productivity of the students? 2. Can APEX help understand bugs?



(a) w/ APEX (b) wo/ APEX

Figure 4.12.: Time taken by students to finish the task

We have implemented a script that records the students' activities, including each compilation, each test run, each revision, and each invocation of our tool. The completion time is what a student took to pass all the test cases (11 in total). Fig. 4.12 shows the results of the two groups. In group (a) (with APEX), only 12% of the students could not finish the task in time. On the other hand, in group (b), 47% of the students could not finish. This supports that APEX can help the students' productivity in programming assignments with 99% certainty (following the A/B significance test). In group (a), 44% finished within an hour while only 18% in group

	(a) w/ APEX	(b) wo/ APEX
Avg	4417.6s	5742.4s
SD	1670.7	1606

Figure 4.13.: Average and standard deviation of time took by each group in seconds

(b). We have also inspected the suggestions APEX generated. Most of them have only 2-3 lines, which imply that APEX did not disclose too much about the correct version. Fig. 4.13 presents the average and standard deviation of time took by each group considering timeouts as 2 hours. On average students in group (a) took about 67 minutes and students in group (b) took 74 minutes. On average group (a) can complete the task about 23% faster than group (b). We also performed the t-test with our data. The p-value is 0.0316 and with 95% confidence our system can help students' productivity.

A. Suggestions are easy to understand.
B. Suggestions are useful to locate error.
C. Suggestions are useful to understand errors.
D. Suggestions are useful to understand correct algorithm.
E. Suggestions are useful to fix errors.
F. Suggestions are useful for overall productivity.

Figure 4.14.: Questions

Question	Agree	Disagree	Neutral
A	56%	6%	38%
B	61%	11%	28%
C	72%	6%	22%
B+C	78%	11%	11%
D	56%	22%	22%
E	83%	0%	17%
D+E	94%	0%	6%
F	78%	0%	22%

Figure 4.15.: Students' response to the questions

In order to evaluate the quality of our suggestions, we surveyed the participants in group (a). We asked them 6 questions as in Fig. 4.14. We classify the questions

into 4 groups. First, question A asks if the suggestions in pseudo code can be easily apprehended. Second, questions B and C ask if the student can understand their problems more easily with our system. Third, questions D and E ask whether APEX can provide hints on how to fix the problems. Last, question F asks the overall effectiveness of our tool. Fig. 4.15 shows the responses. We have the following observations. First, while 6% of the students complained about difficulties in understanding the suggestion pseudo code, the presentation of the suggestions could be improved. More details are disclosed in Section 4.5.3.

Second, 78% of the students agreed that our tool is useful in either locating or understanding errors. Oral communication with the students discloses that they seem to have very diverse understanding about where the root causes are. Third, 94% of the students agreed that they can get hints on fixing the problems. It was very much appreciated by the students that APEX can present correction suggestions in the context of their code (e.g., using their variables). Last, 78% of the students agreed that our tool can help the overall productivity. This is consistent with the results in Fig. 4.12.

Limitations

One student had a very interesting comment that although she got her bug fixed by copying a constant value in the correction suggestion, she did not understand why she should use the constant. We inspected her case. The buggy code is shown in Fig. 4.16. This code is for converting an integer digit into an alphanumeric digit: converting 10 into A, 11 into B, and so on. The operation at line 4 should be "`digit + 'A' - 10`". APEX precisely reported the root cause and suggested the proper correction. However, the suggestion is simply a line of pseudo code "`digit + 55`". This is because APEX internally operates on the IR level so that letters are all represented as constant values which lack semantic meanings and operations on constants are unfolded.

```
// Convert an integer digit into a character digit.
if ('A' <= digit && digit <= 'Z')
  // SUGGESTIONS
  ??? digit = digit + 'A';  // (BUGGY)
  // Instead,
  +++ digit = digit + 55    // (CORRECT)

  // The constant 55 means 'A' - 10
```

Figure 4.16.: Student's buggy code and the suggestion

We plan to address the problem by adding annotations or textual debugging hints to the instructor's version. In the former example, line 3 could be commented with a debugging hint such as "*It is likely that the constant you use to transform a value to a letter is wrong*". Instead of showing the pseudo code, the instructor can configure the tool to emit the textual hint. Together with the (faulty) variable values emitted by APEX, the student should be able to quickly understand the bug. Note that in order to provide high quality textual hints, internally APEX should capture the precise bug causality and identify the corresponding correct code.

4.5.4 Comparison with PMaxSat

We have also implemented a version of APEX directly based on the PMaxSat formulation. We used Z3 as the PMaxSat solver. We compare the performance and the quality of execution alignment of the two versions. We set the timeout of PMaxSat to 5 minutes and ran it for the stackoverflow cases and the convert cases. The results are shown in Table 4.5. In most of the cases, PMaxSat is much slower than APEX. In 3 out of the 15 stackoverflow cases, PMaxSat could not find the solution in 5 minutes. On average our system can find the alignment in less than 2 seconds, whereas PMaxSat requires more than 90 seconds. The results for the convert cases (the last

row) are similar. Note that the high overhead of PMaxSat may not be acceptable for the students, especially during labs.

In terms of execution alignment, the two versions generate similar results. On average the precision of the approximate version of APEX is more than 79% and the recall is more than 74%, compared to the PMaxSat version. This indicates that most alignments discovered by the approximate version are identical to those by PMaxSat. The most common cases of alignment differences are constant operations that do not depend on the inputs such as initializing variables with 0 and increasing loop variables by 1. In our observation, these operations have very little effect on the generated suggestions.

We compared the performance with WPM3-2015-in [109], a state-of-art incomplete partial maxsat solver. The incomplete solvers can find the solution incrementally and hence they can produce intermediate results as soon as possible. We measured the time took by the WPM3 solver until it finds a solution that can satisfy the same number of the clauses as the solution found by APEX. Table 4.6 presents the comparison result. Though the incomplete solver can reach the similar solution faster than Z3 which is a complete solver, APEX is more than 10 times faster on average.

4.5.5 Experiment with IntroClass Benchmarks

We have also evaluated APEX with the IntroClass Benchmarks [89]. The benchmark is designed for evaluation of automated software repair techniques. Out of the 6 projects, we have selected 5 projects with 710 buggy implementations: `checksum` computes the checksum of input string; `digits` prints each digit of the input number; `grade` computes a letter grade for the input score; `median` finds a median number among the 3 input numbers; `syllables` counts the frequencies of vowels in the input string. We did not select `smallest` because it is too small (usually a few lines).

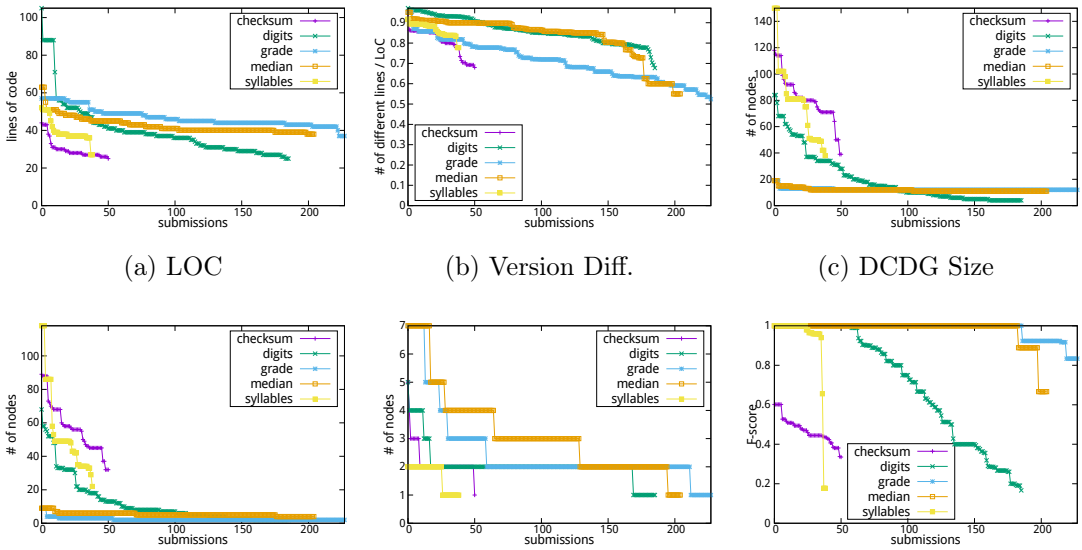
The results are shown in Fig. 4.17. From Fig. 4.17a, the program sizes are mostly 40-60 LOC. Fig. 4.17b suggests that the programs are very different from the solution

Table 4.5.: Comparison between Apex and PMaxSat

Program	Run time (s)		Precison	Recall	F-score
	Apex	PMaxSat			
binarysearch	1.00	14.19	0.91	0.99	0.95
dijkstra	1.58	30.46	0.77	0.77	0.77
dijkstra-2	1.18	6.64	0.70	0.74	0.72
euclid	0.30	.16	0.80	0.80	0.80
euclid-2	0.25	.14	0.71	0.83	0.77
euler1	0.93	1.34	0.97	0.94	0.95
fibonacci-sum	1.06	1.44	0.96	0.88	0.92
floyd	5.04	> 300	-	-	-
gt_product	1.72	> 300	-	-	-
kadane	1.33	4.25	0.60	0.64	0.62
knapsack	1.52	14.28	0.69	0.80	0.74
knapsack-2	1.32	162.58	0.67	0.72	0.69
matmult	0.93	53.75	0.88	1.00	0.93
mergesort	1.48	7.42	0.77	0.91	0.84
span-tree	1.37	> 300	-	-	-
average	1.40	90.21	0.79	0.84	0.81
Convert (average)	2.30	46.02	0.88	0.74	0.79

Table 4.6.: Comparison between APEX and incomplete solver

Program	Run time (s)	
	APEX	WPM3
binarysearch	1.00	2.54
dijkstra	1.58	5.50
dijkstra-2	1.18	1.77
euclid	0.30	0.14
euclid-2	0.25	0.11
euler1	0.93	0.57
fibonacci-sum	1.06	0.74
floyd	5.04	71.68
gt_product	1.72	103.62
kadane	1.33	2.31
knapsack	1.52	2.09
knapsack-2	1.32	2.26
matmult	0.93	2.62
mergesort	1.48	2.78
span-tree	1.37	19.1
average	1.40	14.52
Convert (average)	2.30	20.63
Rpncalc (average)	1.53	52.86
Balanced (average)	0.88	24.15



(a) LOC (b) Version Diff. (c) DCDG Size

(d) Slice Size (e) Bug Report Size (f) F-Score

Figure 4.17.: IntroClass benchmark results. On each figure, the submissions are sorted by the Y-axis values

version syntactically. Fig. 4.17c shows that the DCDGs have 10-150 nodes and some projects such as `median` and `grade` have mostly less than 20 nodes. This is because these programs have no loop and their executions are very short. Fig. 4.17e presents that our suggestions are very small.

Regarding the bugs and the quality of suggestions, we have the following observations. **(1)** For 57 out of 710 cases, APEX missed the root cause. This happens mostly in `median`. The programs in this project have neither loops nor arithmetic operations. They have at most 6 comparisons. In buggy executions, there are usually insufficient evidence for APEX to achieve good alignment. **(2)** Most bugs are due to missing/incorrect conditions, missing computation, or typos in output messages. For example, in `checksum`, 76% of the submissions failed because of missing a modulo operation. Detailed breakdown for all projects can be found at [93]. Note that such information is very useful for the instructors.

4.6 Related Work

In [81], program synthesis was used to automatically correct a buggy program according to the correct version. The technique requires the instructor to provide a set of correction rules. In [82], a technique was proposed to detect algorithms used in functionally correct student submissions by comparing the value sequences. Then it provides feedback prepared by the instructor for each type of algorithm. It is complementary to ours as we could use it to detect cases in which the students are using a different algorithm. Equivalence checking [83, 110] determines if two programs are equivalent. If not, it generates counter-examples. In [85], equivalence checking is extended to look for a single value replacement that can partially fix the faulty state. The value replacement is reported as the root cause. In [86], a technique was proposed to identify the weakest precondition that triggers the behavioral differences between two program versions. These techniques do not align/match the intermediate states, which is critical to understanding failure causality.

There have been works on debugging using the passing and failing executions of *the same program*, by mutating states [3, 87, 111] or slicing two executions [2]. In contrast, Apex assumes different programs. There are also works on comparing traces from program versions to understand regression bugs [5, 56]. They perform sequence alignment in traces without using symbolic analysis. There are satisfiability based techniques that also strive to explain failures within a single program [112–115], and even fix the bugs through templates [116]. They do not leverage the correct version. Fault localization [117–119] leverages a large number of passing and failing runs to identify root causes. In our experience, many buggy student submissions fail on all inputs. Besides, they usually do not explain causality or provide fix suggestions. LAURA [120] statically matches the buggy and correct implementations and reports mismatches. TALUS [121] recognizes the algorithm from the students' submissions and projects the correct implementation to the submissions to generate feedback.

5 ENHANCING APEX WITH BELIEF PROPAGATION

5.1 Introduction

There are many web sites and services to provide self-studies of programming languages and There have been several techniques to either automatically fix or explain bugs in students' codes.

There is a prior work to automatically explain the bug by comparing it with the golden version. However, the technique utilizes symbolic expressions and there are difficulties in finding correct alignments and in producing the correct explanation for programs where statements computing similar values.

In this paper, we will present a new method to find alignments between two program versions using probabilistic inference. We leverage the alignments from the previous technique as prior probabilities of alignments. The previous technique relies on equivalence checking and sequence alignments to compute the alignments. However, the lack of unique states and reordering of program statements can induce false alignments. Thus we model both data and control dependence relations with the prior probabilities. As a result, the false alignments that do not fit to the dependency relations are discarded. The resulting alignments can be used with the unaligned parts of the previous technique and improve the quality of the explanations.

Our contributions are:

- We develop a new alignment algorithm with probability.
- Our approach is evaluated with a large number of students' codes.

```
for (i = 1 to N)
  read(D[i].score);
  D[i].chef = i;

for (i = 1 to Q)
  read(query);
  if (query == C)
    read(x, y);
    if D[x].chef == D[y].chef
      print("invalid query");
    else if D[x].score > D[y].score
      D[y].chef = D[x].chef;
    else
      D[x].chef = D[y].chef;
  else if (query == Q)
    read(x);
    print(D[x].chef)
```

(a) Buggy implementation

```
for (i = 1 to N)
  chef[i] = i;

for (i = 1 to N)
  read(score[i]);

for (i = 1 to Q)
  read(query);
  if (query == C)
    read(x, y);
    a = FindChef(x);
    b = FindChef(y);
    if a == b
      print("invalid query");
    else if score[a] > score[b]
      chef[b] = a;
    else
      chef[a] = b;
  else if (query == Q)
    read(x);
    print(FindChef(x));

function FindChef(c)
  if c != chef[c]
    chef[c] = FindChef(chef[c]);
  return chef[c];
```

(b) Correct implementation

Figure 5.1.: Buggy and correct implementations

5.2 Motivation

Fig. 5.1 are the two implementations of DISHOWN problem in Codechef. The implementations simulate competition between chefs. As an input, the number of chefs and the score of each chef's dish are provided. Then the program reads Q queries. There are two types of queries. The one is given as 'C x y', where x and y mean the indexes of dishes. Upon this query, the program should compute the result of the competition between the two chefs: one who owns the x-th dish and the other who owns the y-th dish. If the owners are the same the program should print "invalid query". Otherwise the program finds who is the winner and let the winner takes all of the loser's dishes. The other type of query is given as 'Q x', where x is the index of a dish. The program should find the owner of the x-th dish and print the owner's index. Initially x-th dish is owned by x-th chef.

Lines 12 and 14 in the buggy implementation update the owner of the loser's dishes to the winner. However they fail to update all of the loser's dishes, and line 9 cannot acquire the correct owner for the dish x and y. On the other hand, lines 11 and 12 in the correct implementation find the correct owner by recursively searching through the winner-loser relationships.

Fig. 5.2 shows parts of inputs, corresponding outputs and internal states of the correct implementation. Input #1 specifies the initial scores for each dishes. Upon reading the input both implementations initialize internal scores with the values. Initially C1(chef #1) owns D1(dish #1), C2 owns D2 and C3 owns D2. Input #2 competes D1 and D2. D1 has a score of 1 and D2 has a score of 2. Since D1 has higher score than D1, C2 who is the owner of the D2 wins and C2 owns both D1 and D2. Input #3 queries the owner of D1. In the previous step, C2 took D2. Hence the program prints "2" as an output. Input #4 competes D1 and D2. Since both D1 and D2 are owned by C2, the program prints "invalid query". Input #5 compares D1 and D3. D1 is owned by C2 and the maximum score of C2 is 2. D3 is owned by C3 whose maximum score is 3. Since C3 has a higher score than C2, C3 wins and C3 owns all

#	Input	Output	State
1	1 2 3		initialize scores
2	C 1 2		C2 wins and now C2 owns D1
3	Q 1	"2"	C2 owns D1
4	C 1 2	"Invalid query"	C2 owns both D1 and D2
5	C 1 3		C3 wins and now C3 owns D1
6	C 2 3	no error	C2 owns D2 and C3 owns D3

(a) Buggy output

#	Input	Output	State
1	1 2 3		initialize scores
2	C 1 2		C2 wins and now C2 owns D1
3	Q 1	"2"	C2 owns D1
4	C 1 2	"invalid query"	C2 owns both D1 and D2
5	C 1 3		C3 wins and now C3 owns D1 and D2
6	C 2 3	"invalid query"	C3 owns both D2 and D3

(b) Expected output

Figure 5.2.: Feedback generated by Apex and ApexBP

three dishes: D1, D2 and D3. Input #5 compares D2 and D3. Since C3 owns both D2 and D3, the program prints "invalid query".

Fig. 5.2 shows the outputs and internal states of the buggy implementation with the same inputs. Up to input #4 the buggy program behaves the same way as the correct program. At input #5, C3 wins and the program changes the owner of D1 to C3. However it does not update the owner of D2 which is also owned by C2. Hence at input #6, the program does not know the owner of D2 and D3 are the same and it does not print "invalid query".

The existing technique, Apex, can be used to automatically generate explanation of the bug. Since Apex relies on symbolic equivalences, it performs well on a program that computes a result through arithmetic operations but it doesn't perform well on a program with less arithmetic operations such as the motivating example.

Apex compares two program executions in 3 steps. In the first step, Apex compares the symbolic expression of every statement in two programs. Then Apex builds the initial alignment with the sequence alignment algorithm and the previous symbolic equivalence information. Last using the initial alignments as truth, Apex finds alignment between remained items.

Fig. 5.3 shows the initial alignments from Apex. Each line in the figure is a statement instance in the program. The left hand side shows statement instances from the buggy program and the right hand side shows the correct program. The first and the fourth column are the labels of statement instances. The second and the fifth column are values of variables at each statement instance. The bold text represents the output values which are used in the sequence alignment algorithm to compare. The gray boxes show the aligned statement instances.

Note that for the simplicity, the concrete values instead of symbolic values are presented in figure. However Apex compares symbolic values if available. For example, the concrete values of 2_1^ℓ is 1. However in symbolic execution, the value is replaced with a symbolic variable. Therefore the sequence alignment algorithm align 2_1^ℓ with

2_1	read(D[i].score)	i=1,**D[i].score=1**	2_1	chef[i]=i	i=1,**chef[i]=1**
3_1	D[i].chef=i	i=1,**D[i].chef=1**	2_2	chef[i]=i	i=2,**chef[i]=2**
2_2	read(D[i].score)	i=2,**D[i].score=2**	2_3	chef[i]=i	i=3,**chef[i]=3**
3_2	D[i].chef=i	i=2,**D[i].chef=2**	5_1	read(score[i])	i=1,**score[i]=1**
2_3	read(D[i].score)	i=3,**D[i].score=3**	5_2	read(score[i])	i=2,**score[i]=2**
3_3	D[i].chef=i	i=3,**D[i].chef=3**	5_3	read(score[i])	i=3,**score[i]=3**
6_1	read(query)	**query=C**	8_1	read(query)	**query=C**
7_1	if query==C	**BR=T**	9_1	if query==C	**BR=T**
8_1	read(x,y)	**x=1,y=2**	10_1	read(x,y)	**x=1,y=2**
9_1	if D[x].chef==D[y].chef	D[x]...=1,D[y]...=2, **BR=F**	11_1	a=FindChef(x)	**a=1**
11_1	else if D[x].score>D[y].score	D[x]...=1,D[y]...=2, **BR=F**	12_1	b=FindChef(y)	**b=2**
14_1	D[x].chef=D[y].chef	**D[x].chef=2**	13_1	if a==b	**BR=F**
			15_1	if score[a]>score[b]	score[a]=1,score[b]=2, **BR=F**
			18_1	chef[a]=b	**chef[a]=2**
6_2	read(query)	**query=Q**	8_2	read(query)	**query=Q**
7_2	if query==C	**BR=F**	9_2	if query==C	**BR=F**
15_1	if query==Q	**BR=T**	19_1	if query==Q	**BR=T**
16_1	read(x)	**x=1**	20_1	read(x)	**x=1**
17_1	print(D[x].chef)	**"2"**	21_1	print(FindChef(x))	**"2"**
6_3	read(query)	**query=C**	8_3	read(query)	**query=C**
7_3	if query==C	**BR=T**	9_3	if query==C	**BR=T**
8_2	read(x,y)	**x=1,y=2**	10_2	read(x,y)	**x=1,y=2**
9_2	if D[x].chef==D[y].chef	D[x]...=2,D[y]...=2, **BR=T**	11_2	a=FindChef(x)	**a=2**
10_1	print("invalid query")	**"invalid query"**	12_2	b=FindChef(y)	**b=2**
			13_2	if a==b	**BR=T**
			14_1	print("invalid query")	**"invalid query"**
6_4	read(query)	**query=C**	8_4	read(query)	**query=C**
7_4	if query==C	**BR=T**	9_4	if query==C	**BR=T**
8_3	read(x,y)	**x=1,y=3**	10_3	read(x,y)	**x=1,y=3**
9_3	if D[x].chef==D[y].chef	D[x]...=2,D[y]...=3, **BR=F**	11_3	a=FindChef(x)	**a=2**
11_2	else if D[x].score>D[y].score	D[x]...=2,D[y]...=3, **BR=F**	12_3	b=FindChef(y)	**b=3**
14_2	D[x].chef=D[y].chef	**D[x].chef=3**	13_3	if a==b	**BR=F**
			15_2	if score[a]>score[b]	score[a]=2,score[b]=3, **BR=F**
			18_2	chef[a]=b	**chef[a]=3**
6_5	read(query)	**query=C**	8_5	read(query)	**query=C**
7_5	if query==C	**BR=T**	9_5	if query==C	**BR=T**
8_4	read(x,y)	**x=2,y=3**	10_4	read(x,y)	**x=2,y=3**
9_4	if D[x].chef==D[y].chef	D[x]...=2,D[y]...=3, **BR=F**	11_4	a=FindChef(x)	**a=3**
11_3	else if D[x].score>D[y].score	D[x]...=2,D[y]...=3, **BR=F**	12_4	b=FindChef(y)	**b=3**
14_3	D[x].chef=D[y].chef	**D[x].chef=3**	13_4	if a==b	**BR=T**
			14_2	print("invalid query")	**"invalid query"**

Figure 5.3.: Initial alignment with sequence alignment algorithm

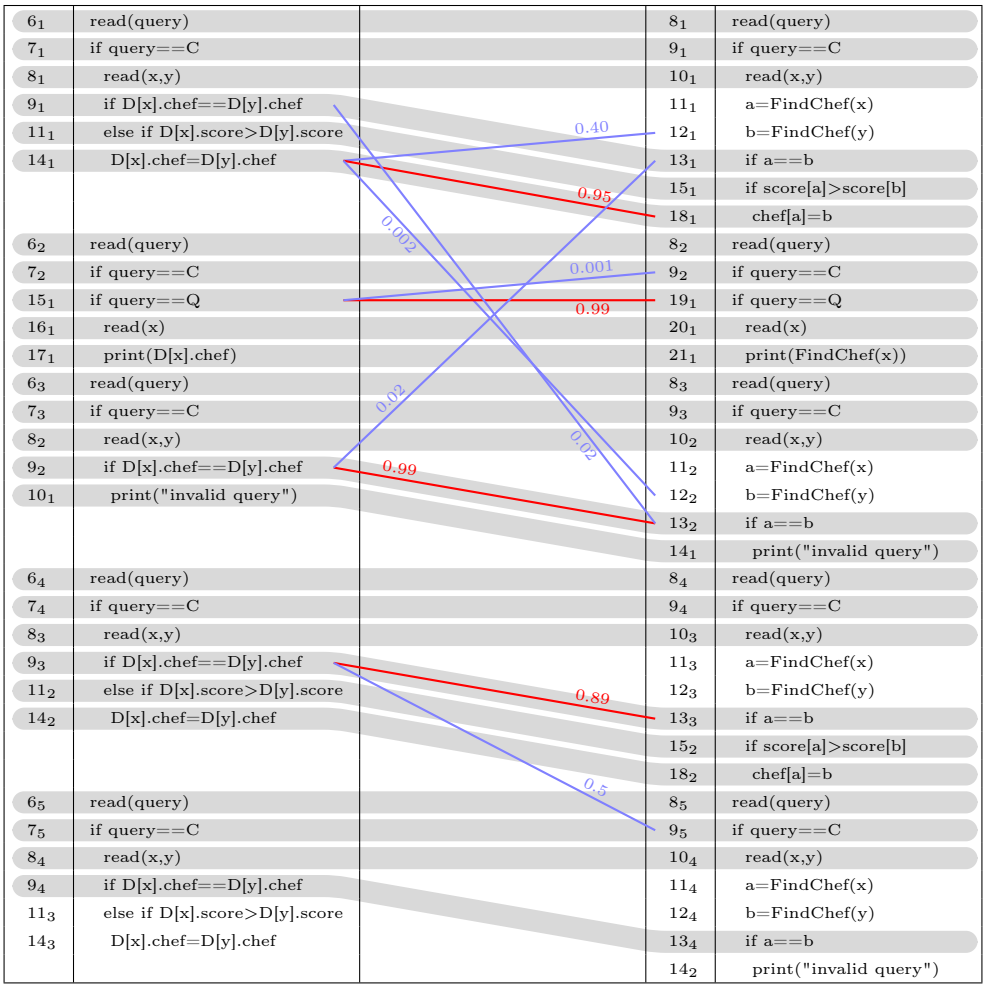


Figure 5.4.: Final alignments generated by ApexBP

5_1^t, which also represented with the same symbolic variable, and it does not align 2_1^ℓ with 2_1^t since its concrete value is the same but its symbolic value is different.

D[i].score and score[i] which represent the score of i-th dish are provided as inputs and have symbolic values. However D[i].chef and chef[i] which represent the owner of the i-th dish are initialized in loops iterating from 1 to N. Thus they have concrete integer values. During the execution both implementations change the value of D[x].chef and chef[x] using the value of D[y].chef and chef[y] respectively. Therefore the values of 14_1^ℓ, 12_1^t and 18_1^t are the concrete value of 2. Therefore both 12_1^t and 18_1^t are equal candidates for 14_1^ℓ for the sequence alignment algorithm.

Because of the fore-mentioned issue, the previous technique aligned the statement instance 14_1^ℓ with 12_1^t. However this is not accurate since the 14_1^ℓ updates the owner of x-th dish with the owner of y-th dish, while 12_1^t finds the owner of y-th dish. Also the previous technique uses these alignments as a baseline for branch alignments. Therefore it cannot detect that the branch 9_1^ℓ should be aligned with 13_1^t since 14_1^ℓ depends on the branch 9_1^ℓ but the branch 13_1^t depends on 12_1^t.

Similarly our technique starts from the initial alignment computed by the sequence alignment algorithm. However we do not consider the initial alignment as the truth, instead we consider it as probabilities. We use our initial probabilities with the belief propagation, and we gradually computes the probabilities of other alignments through control and data dependence relations. Also our initial probabilities can be changed through the propagation. Hence even though there are errors in our initial belief, our system can fix the issues.

Fig. 5.4 shows the result of our alignment algorithm. The red line shows the possible alignments considered in our system and its probabilities. Grey boxes shows the final alignments. For clarity, we do know present the third and sixth column in the Fig. 5.3. For 14_1^ℓ, our system finds 12_1^t, 18_1^t and 12_2^t as candidates for alignment. Among them, 18_1^t has the highest probability of 0.95 and our algorithm aligns 14_1^ℓ with 18_1^t. Through the belief propagation, the alignment between 14_1^ℓ and 18_1^t is propagated to other pairs such as 11_1^ℓ and 15_1^t.

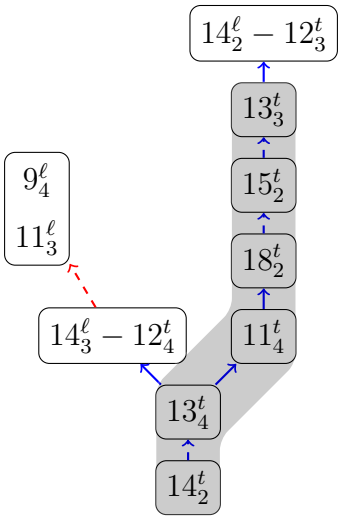


Figure 5.5.: Comparative dependency graphs generated for Apex

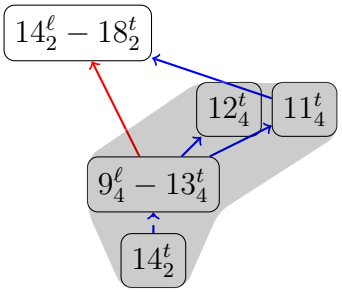


Figure 5.6.: Comparative dependency graphs generated for ApexBP

From the discovered alignment Apex automatically generates a feedback by generating a comparative dependence graph representing the differences in control flow and data flow. The feedback generation method can be used with our alignment algorithm.

Fig. 5.5 and Fig. 5.6 present the part of the comparative dependence graph using Apex's alignment and our alignment respectively. In the graph, each node is either an unaligned statement instance or a pair of aligned statement instance. An edge represents a data or control dependence relations between statement instances. In the graph, a solid line represents a data dependence and a dashed line represents a control dependence. A red line denotes dependence relations from the buggy program and a blue line denotes a dependence from the correct program. The gray region in the graph shows the nodes used to generate a feedback.

Apex starts a traversal of the graph from the print statement instances, which is 14_2^t in this example. It follows dependence edges until it reaches an aligned node. In Fig. 5.5 the feedback generation system first finds 13_4^t since 14_2^t control depends on 13_4^t. Apex does not find any alignment for the instance and follows its dependence edges. By repeating this, the system finds all nodes up to 13_3^t and generates feedback as in Fig. 5.7. The explanation says that some statements are missing in the buggy program and to fix the problem one should implement FindChef and updating owner of dishes. However the latter is already implemented in the buggy program and thus it is redundant.

On the other hand, ApexBP detects the correct alignment between 9_4^ℓ and 13_4^t and between 14_2^ℓ and 18_2^t. In Fig. 5.6, the system first finds 13_4^t similar to the previous scenario. Our alignment discovers that the 13_4^t should be aligned with 9_4^ℓ. However since their branch outcome is different, the system further traverses its dependencies.

```
When query == C at 7^ℓ_4, you should have done
11^t_3 : a = FindChef(x)
12^t_3 : b = FindChef(y)
13^t_3 : if a != b
  15^t_2 : if score[a] > score[b]
    18^t_2 : chef[a] = b
11^t_4 : a = FindChef(x)
12^t_4 : b = FindChef(y)
13^t_4 : if a == b
  14^t_2 : print("invalid query");
```

Figure 5.7.: Feedback generated by Apex

```
When query == C at 7^ℓ_5, you should have done
11^t_4 : a = FindChef(x);
12^t_4 : b = FindChef(y)
13^t_4 : if a == b
  14^t_2 : print("invalid query");
```

Figure 5.8.: Feedback generated by ApexBP

5.3 Apex

In this section we briefly describe the algorithm of Apex, an automated feedback generation system. The algorithm of Apex consists of two parts: sequence alignment and well-formedness constraints.

First given two execution traces, Apex compare symbolic expressions between labels of the two programs. Then Apex runs a sequence alignment algorithm and finds the initial alignment.

Next Apex tries to find more alignments from the initial alignment while preserving the well-formedness constraint, which prohibits cyclic dependencies. The well-formedness constraint can be summarized as following: if two labels ℓ_i and t_j are aligned there cannot be additional alignment between ℓ_p and t_q where ℓ_i depends on ℓ_p and t_q depends on t_j.

In order to find additional alignments while preserving the well-formedness constraint, Apex applies the sequence alignment algorithm iteratively. In the first run, Apex runs the alignment procedure over the entire execution. Then Apex runs the procedure again over the remaining sequence of labels by removing ones in the previous alignment. For each newly discovered alignment Apex checks if the new alignment violates the constraint or not. If not, Apex add the alignment to the aligned labels and repeat the process.

From the alignments, Apex generates the comparative dependency graph by merging the aligned labels in two dependency graph of each version. Then it traverses the graph from the node where output differences are observed.

The key characteristic of the Apex is that in Apex, all the alignments are assumed to be true. The initial alignments are always considered as true alignments and Apex uses them as baseline to find more alignments. This works well in a program where labels have unique symbolic expressions. For example, in a program implementing the euclidean algorithm of the GCD, the key statements have a unique symbolic expressions. Thus if the symbolic expressions are equivalent, it strongly suggests

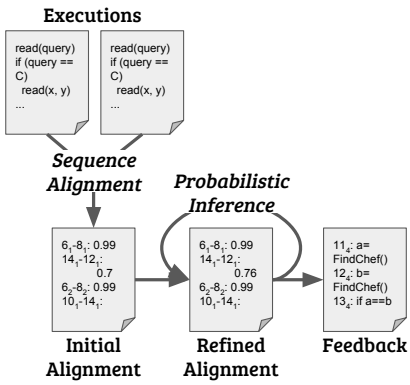


Figure 5.9.: System overview

that two labels must be semantically equivalent in two programs. However in a program where data are copied around like the motivating example, labels have similar expressions and values. In this program, the initial sequence alignment program does not work well.

5.4 System Approach

In this section, we present the details of our algorithm. Fig. 5.9 presents the overview. First the system collects execution traces from two programs. Then the system computes the initial alignment between two executions using the sequence alignment algorithm. Next the initial alignment is refined with probabilistic inference. The refinement process can be repeated with the previous alignments. The refined alignment is fed to the existing feedback generation algorithm.

Alg. 7 describes the overall process of our feedback generation system. Line 2 computes the initial alignment using the sequence alignment algorithm with two program execution traces, E_1 and E_2. Line 3 assigns the probabilities to the initial alignment. Line 5-7 iteratively refine the alignment and probability gradually. Line 8 generate feedback from the refined alignment.

Algorithm 7 Probabilistic alignment

```
1: procedure BP(E_1, E_2)
2:     seqAlign ← SEQUENCEALIGN(E_1, E_2)
3:     initialAlign ← INITIALPROBABILITY(seqAlign)
4:     alignment ← initialAlign
5:     for i = 1 to n do
6:         alignment ← REFINE(alignment, i)
7:     GENERATEFEEDBACK(alignment)
```

5.4.1 Sequence alignment

Algorithm 8 Sequence alignment

```
1: procedure SEQUENCEALIGN(E_1, E_2)
2:     for i_1 = 1 to E_1.length do
3:         for i_2 = 1 to E_2.length do
4:             Align[i_1][i_2] ← max(
5:                 Align[i_1 − 1][i_2 − 1] + SCORE(E_1[i_1], E_2[i_2]),
6:                 Align[i_1 − 1][i_2], Align[i_1][i_2 − 1])
7:     alignments ← {}
8:     i_1 ← E_1.length
9:     i_2 ← E_2.length
10:    while i_1 ≥ 1 ∩ i_2 ≥ 1 do
11:        if Align[i_1][i_2] == Align[i_1 − 1][i_2 − 1] + cost(E_1[i_1], E_2[i_2]) then
12:            alignments ← alignments ∪ (E_1[i_1], E_2[i_2])
13:            i_1 ← i_1 − 1
14:            i_2 ← i_2 − 1
15:        else if Align[i_1][i_2] == Align[i_1 − 1][i_2] then
16:            i_1 ← i_1 − 1
17:        else if Align[i_1][i_2] == Align[i_1][i_2 − 1] then
18:            i_2 ← i_2 − 1
19: procedure SCORE(ℓ_i, t_j)
20:    if symb(ℓ_i) ≡ symb(t_j) then
21:        return1
22:    else
23:        return0
```

Alg. 8 describes the Needleman-Wunsch sequence alignment algorithm used in our system. We use the score of 1 when two statement instances have the equivalent

$$p(I_{\ell_i,t_j}) = \begin{cases} 0.99 & \text{if } symb(\ell_i) \equiv symb(t_j) \wedge \\ & \quad \text{both have symbolic values} \\ 0.99 & \text{if } t_j and t_j are identical print statements \\ 0.70 & \text{otherwise} \end{cases}$$

Figure 5.10.: Initial probability

symbolic expressions and use the score of 0 otherwise. This algorithm finds the maximal alignment between two sequences of instances. Note that, all alignments found by this algorithm always have equivalent symbolic expressions.

Then, we assign probabilities to the alignments according to the following rule.

For each pair of statement instances in the initial alignment, if symbolic expressions of two instances have symbolic variables, we assign a high probability, 0.99 to the pair. Otherwise if their symbolic expressions are concrete values, we assign a relatively low probability of 0.7 to the pair.

Symbolic values in the expressions denotes input values. For example, we assign a symbolic value I1 for the first input value and so on. Therefore if the value of a statement instance does not data depend on any input values, its symbolic expression can be a concrete value. This can happen frequently for loop variables since in many cases loop variables starts from a concrete value such as 0 and increases by 1.

The intuition is that if two symbolic expressions having symbolic values are equivalent, it strongly suggests that two instances are indeed semantically equivalent. However symbolic expressions of concrete values are common. For example loop variables are likely to have the same sequence of values since they normally starts from 0 and increases by 1. Therefore we give a lower probability to represent that this alignment has a possibility of being wrong.

5.4.2 Refinement

Once the initial alignments are discovered by the sequence alignment algorithm. We apply probabilistic inference and refine the existing alignments. This process can be applied multiple times to increase the search space of alignments.

Algorithm 9 Alignment algorithm

```
1: procedure REFINE(oldAlignments, n)
2:     bp ← PROPAGATE(oldAlignments, n)
3:     bpResult ← solve(bp)
4:     newAlignments ← {}
5:     for align ∈ bpResult do
6:         if align.probability > θ then
7:             newAlignments ← newAlignments ∪ {align}
8:     return newAlignments
```

Alg. 9 describes the refining process. Line 2 propagates and builds a probabilistic model with old alignments. Line 3 solves the model and computes the new probabilities. Among the new probabilities, probabilities higher than a threshold are added to the new alignments set. This process can be repeated multiple times.

There are 2 reasons to make the process iterative. One is to prioritize the closer pair. Our propagation rule considers all transitive dependencies of a pair. In our observation, two statement instances are align, in close distance there are aligned dependencies of the pair. Since our rule does not consider distance, we start building our factor graphs by considering only close distance, and then expand the graph iteratively.

The second reason is to reduce the size of the factor graph. In our factor graph models, we create a random variable for every pair of statement instances. To limit the number of random variables, we first considers only a small distance. Once we can find strong candidates within a small distance, we can start building our graph

from the pair. With this approach we can reduce the number of candidates and thus can solve our factor graph faster.

Algorithm 10 Belief propagation

```
1: procedure PROPAGATE(alignments, n)
2:     worklist ← alignments
3:     while worklist is not empty do
4:         I_{ℓ_i,t_j}, distance ← pop(worklist)
5:         ApplyRule(I_{ℓ_i,t_j})
6:         if distance <= threshold then
7:             for I_{ℓ_p,t_q} ∈ dep(ℓ_i, n) × dep(t_j, n) do
8:                 worklist ← worklist ∪
9:                     {(I_{ℓ_p,t_q}, distance + 1)}
```

Alg. 10 shows the process of building a probabilistic model from a set of alignments. This alignments can be either the initial alignments or the refined alignments.

This process uses a worklist algorithm. Line 2 initializes the worklist with the provided alignments. Line 3-8 repeats until the worklist is empty. Line 4 and 5 pops an item from the worklist and apply the fore-mentioned propagation rules to the pair. This function limits the worklist by distance from the previous aligned pair. Line 6-8 add more candidates from the current aligned pair if the current pair is within a certain distance from the previous aligned pair. Also for the candidates from the current aligned pair, the algorithm only considers dependencies from the current instance up to distance up to distance n. This can prevent rapid growth of the probabilities model.

5.4.3 Probabilistic inference

After we acquire the initial alignments and their probabilities, we refine the alignments with probabilistic inference. First we describe the intuition of our inference rules and the procedure to build a probabilistic model from execution traces.

DEFINITIONS:

ℓ, t : statement label

ℓ_i, t_j : the i/j-th instance of label ℓ and t

I_{ℓ_i,t_j} : true if ℓ_i and t_j are aligned

$S_{\ell,t}$: true if ℓ and t are aligned

$symb(\ell_i)$: symbolic expression of ℓ_i

$cdep(\ell_i, n)$: set of direct control dependencies of ℓ_i upto distance n

$ddep(\ell_i, n)$: set of direct data dependencies of ℓ_i upto distance n

RULES:

1. $symb(\ell_i) \equiv symb(t_j) \overset{p}{\Rightarrow} I_{\ell_i,t_j}$
2. $I_{\ell_i,t_j} \overset{p}{\Rightarrow} \forall \ell_p \in cdep(\ell_i, n),\ \exists t_q \in cdep(t_j, n),\ I_{\ell_p,t_q}$
3. $I_{\ell_i,t_j} \overset{p}{\Rightarrow} \forall \ell_p \in ddep(\ell_i, n),\ \exists t_q \in ddep(t_j, n),\ I_{\ell_p,t_q}$
4. $I_{\ell_i,t_j} \overset{p}{\Leftrightarrow} S_{\ell,t}$

Figure 5.11.: Basic idea

Fig. 5.11 shows the basic idea of our probabilistic inference. Rule 1 suggests that if two labels have equivalent symbolic expressions, it is likely that two labels are aligned. Rule 2 describes that if two labels ℓ_i and t_j are aligned, dependencies of ℓ_i are likely to be aligned with dependencies of t_j. Rule 3 means that if ℓ_i and t_j are aligned, it is likely that their corresponding statements ℓ and t are aligned and vice versa. Due to an error in different location, two labels that should be aligned may have different symbolic expressions and system may fail to align them. Rule 3 suggests that once two statements instances are aligned, it is likely that their other instances are aligned.

Note that the dependencies in rule 2 mean both direct and transitive dependencies. Due to different implementations, two equivalent dependencies may appear as a direct dependency in one version and as a transitive one in another version.

```
mid = (low + high) / 2;
```

```
diff = high - low;
mid = low + diff / 2;
```

The above code snippets show two different way to compute the middle point of low and high. The left hand side code computes the middle point directly from low and high. The mid in that code directly depends on the variable low and high. On the other hand, the right hand side code first computes differences between high and low. Then it computes the mid by adding half of the differences to the low. In this code, mid directly depends on low but not on high. Instead it transitively depends on high. Therefore we consider both direct and transitive dependencies.

But we cannot consider all transitive dependencies in rule 2. If we consider, any pair of instances that originate from the same value can have high probabilities. To prevent this issue, we limit the distance of dependencies and iteratively increase the maximum distance. First we search direct dependencies and if we can find the aligned dependencies, the pair will have a high probability. In the future iteration, we increase the bound and search for longer distance. Therefor we consider prioritize the closest dependencies.

Factor Graph

With the previous basic idea, we build a factor graph from the executions and their dependence relations. Factor graph is a graphical representation of probabilistic model. It is a bipartite graph and consists of two types of nodes: factors and random variables. In factor graph, probability of the entire system is computed from multiplications of factors. A factor is a probability function on a set of random variables.

$$p(\mathbb{X}) = \prod_{i=1}^{n} f_i(\mathbb{X}_i) \quad , \mathbb{X}_i \subset \mathbb{X}$$

In our model, we use two types of binary random variables. The first random variable, I_{ℓ_i,t_j} is true if ℓ_i and t_j are aligned and false otherwise. Another random variable $S_{\ell,t}$ is true if ℓ and t are aligned.

Fig. 5.12 presents the factor functions used in our model. $f_P(I_{\ell_i,t_j})$ models rule 1. It assigns high probability if symbolic expressions of ℓ_i and t_j are equivalent. It assigns low probability if the values are different. Otherwise it assigns probability of 0.5 which means there is no preference. $f_H(I_{\ell_i,t_j}, S_{\ell,t})$ models rule 3. It assigns a high probability if two instances are aligned and two statements are aligned. If two statements are aligned and two instances are not aligned, it gives a low probability.

$f_D(I_{\ell_i,t_j}, I_{\ell_{d_1},t_{d_1}}, \cdots, I_{\ell_{d_p},t_{d_q}})$ models rule 2. It assigns a probability of 0.99 if ℓ_i and t_j are aligned and ℓ_{d_i}, one of the dependencies of ℓ_i, and t_{d_j}, one of the dependencies of t_j are aligned. Also ℓ_{d_i} and t_{d_j} should be the same type of dependencies, which means that both ℓ_{d_i} and t_{d_j} are both data dependencies of ℓ_i and t_j or both control dependencies of them. Since it is highly unlikely that a data dependency of ℓ_i and a control dependency of t_j are aligned, we do not consider them as candidates in the first place.

$$f_P(I_{\ell_i,t_j})$$
$$= \begin{cases} 0.9 & \text{if } \; symb(\ell_i) \equiv symb(t_j) \\ 0.2 & \text{if } \; value(\ell_i) \neq value(t_j) \\ 0.5 & \text{otherwise} \end{cases}$$

$$f_{CD}\left(I_{\ell_i,t_j}, I_{\ell_{d_1},t_{d_1}}, \cdots, I_{\ell_{d_p},t_{d_q}}\right)$$
$$= \begin{cases} 0.99 & \text{if } \; I_{\ell_i,t_j} \wedge \left(I_{\ell_{d_1},t_{d_1}} \cup \cdots \cup I_{\ell_{d_p},t_{d_q}}\right) \\ & \quad \wedge \, (\ell_{d_i}, t_{d_i}) \in cdep(\ell_i) \times cdep(t_j) \\ 0.01 & \text{otherwise} \end{cases}$$

$$f_{DD}\left(I_{\ell_i,t_j}, I_{\ell_{d_1},t_{d_1}}, \cdots, I_{\ell_{d_p},t_{d_q}}\right)$$
$$= \begin{cases} 0.99 & \text{if } \; I_{\ell_i,t_j} \wedge \left(I_{\ell_{d_1},t_{d_1}} \cup \cdots \cup I_{\ell_{d_p},t_{d_q}}\right) \\ & \quad \wedge \, (\ell_{d_i}, t_{d_i}) \in ddep(\ell_i) \times ddep(t_j) \\ 0.01 & \text{otherwise} \end{cases}$$

$$f_H(I_{\ell_i,t_j}, S_{\ell,t})$$
$$= \begin{cases} 0.1 & \text{if } \; \neg I_{\ell_i,t_j} \wedge S_{\ell,t} \\ 0.9 & \text{if } \; I_{\ell_i,t_j} \wedge S_{\ell,t} \\ 0.5 & \text{otherwise} \end{cases}$$

Figure 5.12.: Factors

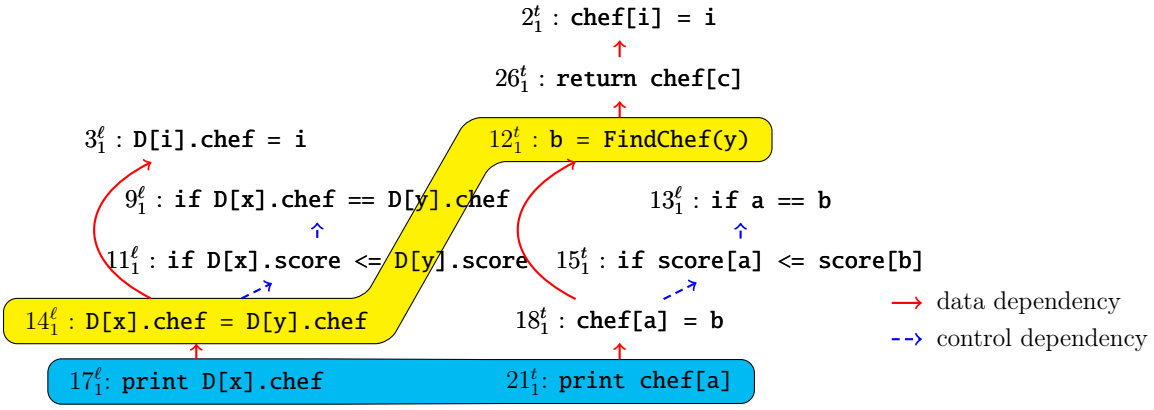


Figure 5.13.: Dependency graph

5.4.4 Example

Fig. 5.3 shows the result of the sequence alignment algorithm in the motivating example. In this section, we will show that how the refinement process can improve the initial alignments.

Fig. 5.13 shows the dependency graph of the motivating example. In the figure, red arrows denote data dependencies and blue dashed arrows denote control dependencies. A box represents a pair of aligned labels in the initial alignment. A blue box shows an alignment considered as truth and having a high probability of 0.99. A yellow box shows an alignment with relatively low probability of 0.7, which means that the alignment is in the initial alignments but it can be wrong.

In our initial probability rules, we consider that two print statements are aligned with a high probability if they the same value in the same order. In the motivating example, both 17^ℓ_1 and 21^t_1 are the first print statement in their executions and they both print the same correct output. Hence the pair is considered as an truth and our system assign a high probability of 0.99 to the pair.

Fig. 5.14 shows the factors generated from the dependency graph in Fig. 5.13 during the first iteration of refinement process. In the first iteration, we limit the

$$f_P(I_{17^\ell_1,21^t_1}) = 0.99$$

$$f_H(I_{17^\ell_1,21^t_1}, S_{17^\ell,21^t}) = \begin{cases} 0.99 & I_{17^\ell_1,21^t_1} = T, S_{17^\ell,21^t} = T \\ 0.01 & I_{17^\ell_1,21^t_1} = T, S_{17^\ell,21^t} = F \\ 0.01 & I_{17^\ell_1,21^t_1} = F, S_{17^\ell,21^t} = T \\ 0.99 & I_{17^\ell_1,21^t_1} = F, S_{17^\ell,21^t} = F \end{cases}$$

$$f_{DD}(I_{17^\ell_1,21^t_1}, I_{14^\ell_1,18^t_1}) = \begin{cases} 0.99 & I_{17^\ell_1,21^t_1} = T, I_{14^\ell_1,18^t_1} = T \\ 0.01 & I_{17^\ell_1,21^t_1} = T, I_{14^\ell_1,18^t_1} = F \\ 0.01 & I_{17^\ell_1,21^t_1} = F, I_{14^\ell_1,18^t_1} = T \\ 0.99 & I_{17^\ell_1,21^t_1} = F, I_{14^\ell_1,18^t_1} = F \end{cases}$$

$$f_P(I_{14^\ell_1,18^t_1}) = 0.5$$

$$f_H(I_{14^\ell_1,18^t_1}, S_{14^\ell,18^t}) = \cdots$$

$$f_{DD}(I_{14^\ell_1,18^t_1}, I_{3^\ell_1,12^t_1}) = \begin{cases} 0.99 & I_{14^\ell_1,18^t_1} = T, I_{3^\ell_1,12^t_1} = T \\ 0.01 & I_{14^\ell_1,18^t_1} = T, I_{3^\ell_1,12^t_1} = F \\ 0.01 & I_{14^\ell_1,18^t_1} = F, I_{3^\ell_1,12^t_1} = T \\ 0.99 & I_{14^\ell_1,18^t_1} = F, I_{3^\ell_1,12^t_1} = F \end{cases}$$

$$f_{CD}(I_{14^\ell_1,18^t_1}, I_{11^\ell_1,15^t_1}) = \begin{cases} 0.99 & I_{14^\ell_1,18^t_1} = T, I_{11^\ell_1,15^t_1} = T \\ 0.01 & I_{14^\ell_1,18^t_1} = T, I_{11^\ell_1,15^t_1} = F \\ 0.01 & I_{14^\ell_1,18^t_1} = F, I_{11^\ell_1,15^t_1} = T \\ 0.99 & I_{14^\ell_1,18^t_1} = F, I_{11^\ell_1,15^t_1} = F \end{cases}$$

$$f_P(I_{3^\ell_1,12^t_1}) = 0.5$$

$$f_P(I_{11^\ell_1,15^t_1}) = 0.5$$

$$f_{CD}(I_{11^\ell_1,15^t_1}, I_{9^\ell_1,13^t_1}) = \cdots$$

Figure 5.14.: Factors generated for Fig. 5.13

15_1^ℓ - 19_1^t 0.99
7_1^ℓ - 9_1^t 0.76
7_3^ℓ - 9_3^t 0.90
7_5^ℓ - 9_5^t 0.84
9_2^ℓ - 13_2^t 0.90
14_1^ℓ - 18_1^t 0.76
⋮

Figure 5.15.: Probabilities after the first iteration

length of the dependency chain in generating factors to 1. In other words, we only consider the direct dependencies of the aligned labels in factors.

From the aligned labels 17_1^ℓ-21_1^t, we generate the factors $f_P(I_{17_1^\ell,21_1^t})$, which represents the initial probability and $f_H(I_{17_1^\ell,21_1^t}, S_{17^\ell,21^t})$, which denotes that two statements are aligned if their instances are aligned. Then we generate the factor $f_{DD}(I_{17_1^\ell,21_1^t}, I_{14_1^\ell,18_1^t})$, which shows the relation between the aligned labels and their direct dependencies.

Then we add the pair 14_1^ℓ-18_1^t to the worklist and further generates factors from the pair. Since the symbolic expressions of 14_1^ℓ and 18_1^t are equivalent but they do not include any symbolic value, the prior probability of the pair is assigned with 0.5.

Belief propagation algorithms can solve the generated factor graph and find the probabilities of each random variables such that the probability of the entire system is maximized. Fig. 5.15 shows the parts of aligned labels and their probabilities. Among them, if the largest probability exceeds a certain threshold, we treat that as a new truth. In this case, we use 0.7 as threshold and 14_1^ℓ-18_1^t becomes a truth.

In the next refinement process, we set the probability of 14_1^ℓ-18_1^t with a higher probability and repeat the process once again. Fig. 5.16 shows the parts of label pairs and their probabilities of alignment. Note that the pair 14_1^ℓ-18_1^t has a higher probability than the pair 14_1^ℓ-12_1^t, which was in the initial alignment of Apex.

$$
\begin{array}{cc}
16_1^\ell \text{ - } 20_1^t & 0.99 \\
3_3^\ell \text{ - } 2_3^t & 0.86 \\
3_2^\ell \text{ - } 2_2^t & 0.79 \\
14_1^\ell \text{ - } 18_1^t & 0.96 \\
14_1^\ell \text{ - } 12_1^t & 0.35 \\
\vdots &
\end{array}
$$

Figure 5.16.: Probabilities after the second iteration

5.5 Evaluation

We implement our system with LLVM and Python. We use an LLVM module to statically instrument the source code of both buggy and correct programs and record the executions. We record labels, values and dependencies of statement instances at runtime. We build alignment, refinement and feedback generation modules with Python. Symbolic expressions of labels are compared with Z3 [92] solver. The factor graph is constructed and solved with libdai.

We evaluate our system in Arch Linux 5.0.6 with i7 2600k CPU and 16GB memory. Our system is evaluated with DISHOWN implementations from Codechef and CoREBench [52].

5.5.1 Codechef DISHOWN case

We evaluate our system with the DISHOWN implementations described in the motivating example. We compared the result with Apex.

Table 5.1.: Evaluation from Codechef DISHOWN

ID	A[1]	B[2]	B-A	C[3]	D[4]	ID	A	B	B-A	C	D
10645832	8	10	+2	8	23	6022804	38	12	-26	2	24
10915727	7	10	+3	14	72	6022856	38	12	-26	3	25
10915734	7	10	+3	15	76	6066091	10	10	=	14	35
11184971	6	3	-3	11	34	6066125	8	10	+2	14	29
11817758	17	10	-7	9	36	6078133	8	10	+2	14	29
11817839	17	10	-7	8	39	6078171	8	10	+2	13	34
11830415	10	10	=	5	34	6082523	8	10	+2	13	36
11830450	10	10	=	5	35	6381515	4	2	-2	20	72

[1]Apex
[2]ApexBP
[3]Only Apex
[4]Only ApexBP

11830534	10	10	=	5	34	6523830	30	5	-25	10	77
11831061	10	10	=	4	23	6523931	27	15	-12	10	79
11898078	10	5	-5	13	48	6523958	27	15	-12	10	79
11898120	10	5	-5	13	48	6525533	30	5	-25	10	77
11898548	44	7	-37	13	47	6525602	27	15	-12	10	79
11987411	3	3	=	8	46	6549951	8	10	+2	13	28
11993995	11	4	-7	13	60	6550441	8	10	+2	13	28
12114817	10	10	=	10	32	6550457	8	10	+2	13	27
12114829	10	10	=	10	32	6551449	10	10	=	3	31
4330122	10	10	=	13	34	6583818	29	13	-16	4	58
4330127	10	10	=	13	34	6613096	10	10	=	8	42
4331036	9	8	-1	10	75	6613182	10	10	=	7	43
4332136	10	10	=	4	26	6778319	8	10	+2	10	38
4332188	8	10	+2	7	26	6940409	3	3	=	10	55
4332462	24	15	-9	5	57	6940568	3	3	=	10	58
4333710	8	8	=	11	33	7010396	10	10	=	4	32
4334609	27	15	-12	12	72	7022503	10	10	=	6	38
4336274	3	3	=	11	25	7032552	9	8	-1	6	6
4336298	3	3	=	11	25	7032556	9	8	-1	6	6
4336333	0	3	+3	11	29	7063317	8	10	+2	4	30
4336341	8	10	+2	11	36	7221510	8	10	+2	12	28
4336369	8	10	+2	11	36	7221525	8	10	+2	12	27
4336380	8	10	+2	11	36	7221560	8	10	+2	12	27
4336384	8	10	+2	11	36	7221579	8	10	+2	12	27
4336582	8	8	=	8	42	7255348	10	10	=	4	36
4337965	7	10	+3	10	25	7255355	3	3	=	10	54
4343567	11	11	=	6	27	7255360	3	3	=	10	55
4343578	10	10	=	6	23	7433673	6	10	+4	11	29
4343645	11	12	+1	8	32	7433751	8	10	+2	15	31

4343649	26	20	-6	10	30	7596507	3	3	=	6	57
4343890	27	15	-12	6	69	7622659	13	3	-10	5	64
4343909	27	15	-12	6	72	7659042	10	10	=	5	26
4344055	9	10	+1	6	74	7659300	10	10	=	6	26
4345734	27	15	-12	11	49	7717049	26	10	-16	6	51
4362515	3	3	=	13	64	7718862	37	10	-27	4	51
4401847	10	10	=	5	23	7726227	21	15	-6	3	26
4401960	8	10	+2	8	34	7823122	3	3	=	6	22
4469962	10	10	=	8	29	7823243	26	20	-6	9	26
4469983	10	10	=	8	33	7823272	26	20	-6	10	26
4473718	3	3	=	4	29	7828759	10	10	=	7	33
4473737	10	10	=	7	37	7828825	10	10	=	7	36
4473771	8	10	+2	10	37	7828942	10	10	=	6	36
4473783	10	10	=	10	35	7829163	9	10	+1	6	73
4475304	15	11	-4	13	43	7873858	3	3	=	3	85
4475521	6	10	+4	17	41	7873888	3	3	=	11	73
4475537	6	10	+4	17	41	7874037	3	10	+7	9	88
4475618	6	10	+4	17	41	7973881	10	10	=	4	31
4506943	3	3	=	13	68	7973936	10	10	=	4	29
4560878	10	10	=	7	32	7974011	10	10	=	5	29
4567937	18	11	-7	6	37	7974090	10	10	=	4	35
4634792	3	3	=	6	30	7974157	10	10	=	6	32
4634797	3	3	=	6	30	8864668	10	10	=	12	22
4634841	10	10	=	6	28	9011087	10	10	=	6	31
4658289	10	10	=	7	28	9013821	10	10	=	7	26
4698512	8	10	+2	10	27	9086815	8	10	+2	10	34
4698762	8	10	+2	10	34	9087615	8	10	+2	10	34
4998899	18	12	-6	9	38	9087887	8	10	+2	10	39
4999289	17	10	-7	13	54	9088506	26	20	-6	9	69

5216473	6	10	+4	12	31	9100384	17	8	-9	6	54
5216502	6	10	+4	7	33	9100863	27	15	-12	5	54
5216511	6	10	+4	12	30	9395795	8	11	+3	10	31
5228540	8	10	+2	10	16	9672735	3	3	=	5	60
5228542	10	10	=	8	26	9673184	31	8	-23	13	67
5228543	8	10	+2	8	28	9673205	31	8	-23	13	67
5264689	8	10	+2	7	31	9722378	10	10	=	7	28
5264910	8	10	+2	9	31	9722540	10	10	=	5	26
5269552	6	9	+3	14	44	9722543	10	10	=	5	26
5604706	10	10	=	4	28	9722587	10	10	=	5	26
5604742	10	10	=	4	32	9925702	8	8	=	12	42
5604749	10	10	=	4	32	9925709	8	8	=	12	42
5604768	10	10	=	4	28	9925713	8	8	=	12	42

We collect 214 buggy submissions of DISHOWN problem from Codechef. We evaluate our system against 158 implementations which are not compile errors nor crash errors and for which we can find the bug inducing inputs. The evaluation results with the 158 programs are presented in Table. 5.1.

ID column represents the id of each buggy submission in the Codechef. Apex column shows the size of the feedback generated by Apex and bp shows the size of the feedback from our technique. Next column shows the difference between the size of feedback generated by Apex and our technique. Positive value denotes that our technique generate longer feedback and negative value means the opposite. = represents that Apex and our technique generate the same feedback. AA and AB shows the number of alignments only in Apex and our technique respectively.

In 70 cases Apex and our technique produces the same feedback. In 48 cases, our technique produces shorter feedback than Apex's. In 40 cases, Apex produces shorter feedback.

5.5.2 CoREBench

Table 5.2.: Size of feedback from CoREBench cases

Bug Id	Prog	Execution Size		Feedback Size		
		Buggy	Correct	Apex	ApexBP	Apex - ApexBP
core.06aeeecb	cut	4792	4416	11	5	+6
core.2e636af1	cut	1813	3198	6	5	+1
core.5ee7d8f5	rm	1997	3567	23	23	=
core.6124a384	ls	5270	5131	20	4	+16
core.61de57cd	tail	3605	3063	5	5	=
core.62543570	cp	6901	6662	12	5	+7
core.6fc0ccf7	expr	2283	2251	26	2	+24
core.a04ddb8d	ls	70156	70134	2	2	=
core.a6a447fc	cut	1595	1041	19	19	=
core.a860ca32	seq	1527	1391	3	2	+1
core.be7932e8	cut	3128	4819	23	5	+17
core.f7f398a1	du	6535	7075	21	7	+14
find.091557f6	find	32351	34899	17	18	-1
find.24bf33c0	find	36825	36411	10	7	+3
find.24e2271e	find	10157	9876	106	2	+104
find.66c536bb	find	40442	41035	2	1	+1
find.93623752	find	4387	4382	19	15	+4
find.b445af98	find	36843	37503	2	2	=
find.dbcb10e9	find	4889	38416	1	3	-2
find.e1d0a991	find	16763	15657	20	10	+10
grep.2be0c659	grep	131953	133407	90	65	+25
grep.3c3bdace	grep	32184	49485	5	6	-1
grep.54d55bba	grep	61685	67544	11	4	+7

grep.55cf7b6a	grep	15925	17377	18	4	+14
grep.7aa698d3	grep	138313	134045	90	65	+25
grep.8f08d8e2	grep	199452	199008	14	14	=
grep.c1cb19fe	grep	81287	350049	8	3	+5
grep.c96b0f2c	grep	74345	73403	20	6	+14
average				21.6	11.0	

Table. 5.2 shows the evaluation results with CoREBench. CoREBench is a benchmark for regression testing. It consists of various versions of coreutils, findtools, grep and make with regression bugs. We evaluate our system with the real world regression bugs to show that our technique can be applied to understand the real world regression bugs as well as programming assignments.

We evaluated with 28 cases from CoREBench. The makefile cases generate huge amount of execution traces because the makefile performs parsing. Due to the size of the execution traces, we cannot evaluate our system with makefile. Also a few cases that uses multi-thread, that does not have obvious difference in outcome, and that we couldn't reproduce are excluded. We use the entire execution traces of the target programs. The size of the execution trace is 42547 statement instances on average.

On average Apex generates feedback with 21.6 statements while our technique does with 11 statements. We examined the feedback manually and there are no significant different in the quality of the result.

REFERENCES

[1] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 253–267, London, UK, UK, 1999. Springer-Verlag.

[2] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 253–264, New York, NY, USA, 2010. ACM.

[3] William N. Sumner and Xiangyu Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 272–281, Piscataway, NJ, USA, 2013. IEEE Press.

[4] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 15:1–15:16, Berkeley, CA, USA, 2007. USENIX Association.

[5] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 241–252, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 347–362, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Babak Salamat. *Multi-variant Execution: Run-time Defense Against Malicious Code Injection Attacks*. PhD thesis, University of California at Irvine, Irvine, CA, USA, 2009. AAI3359500.

[8] Doug Mcilroy. Mass-produced Software Components. *Software Engineering Concepts and Techniques*, pages 138–155, January 1969.

[9] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, April 1997.

[10] William B. Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, July 2005.

[11] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. Decomposing legacy programs: a first step towards migrating to client-server platforms. In *Proceedings of the 6th IEEE International Workshop on Program Comprehension (IWPC)*, 1998.

[12] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 1992.

[13] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, January 1995.

[14] F. Cutillo, F. Lanubile, and G. Visaggio. Extracting application domain functions from old code: a real experience. In *Proceedings of the IEEE Second Workshop on Program Comprehension (WPC)*, 1993.

[15] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.

[16] Maksym Petrenko and VáClav Rajlich. Concept location using program dependencies and information retrieval (depir). *Information and Software Technology*, 55(4):651–659, April 2013.

[17] Kunrong Chen and Vaclav Rajlich. Ripples: Tool for change in legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2001.

[18] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013.

[19] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[20] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[21] G. Antoniol and Y.-G. Gueheneuc. Feature identification: a novel approach and a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[22] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, January 2006.

[23] A.D. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[24] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC)*, 2003.

[25] Ran Ettinger. Program sliding. In *Proceedings of the 26th European conference on Object-Oriented Programming (ECOOP)*, 2012.

[26] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[27] Katsuhisa Maruyama. Automated method-extraction refactoring by using block-based slicing. In *Proceedings of the 2001 Symposium on Software Reusability: putting software reuse in context (SSR)*, 2001.

[28] Arun Lakhotia and Jean-Christophe Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11-12):677–689, 1998.

[29] Research announcement: Binary executable transforms. Defense Advanced Research Projects Agency, 2011.

[30] Chris Puttick. Preserving legacy files with ECMA Office Open XML (MSOOXML). ODF Europe Action Group, 2007.

[31] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, 2010.

[32] Michael N. Gagnon, Stephen Taylor, and Anup K. Ghosh. Software protection through anti-debugging. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, 2007.

[33] W. Eric Wong, Joseph R. Horgan, Swapna S. Gokhale, and Kishor S. Trivedi. Locating program features using execution slices. In *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET)*, 1999.

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

[35] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[36] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[37] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[38] Dasarath Weeratunge and X Zhang. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[39] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. BISTRO: Binary Component Extraction and Embedding for Software Security Applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[40] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[41] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, pages 707–710, 1966. English translation in Soviet Physics Doklady, 10(8).

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[43] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szyd-lowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[44] P. Porras, Hassen Saïdi, and V. Yegneswaran. A Foray into Conficker's Logic and Rendezvous Points. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (LEET)*, 2009.

[45] Demo of murofet domain flux extraction. `https://www.youtube.com/watch?v=7GU4b68oWD4`, 2013.

[46] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1128–1139, New York, NY, USA, 2014. ACM.

[47] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 69–76, New York, NY, USA, 2005. ACM.

[48] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[49] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the Eighteenth*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 127–138, New York, NY, USA, 2013. ACM.

[50] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM.

[51] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, November 2007.

[52] Marcel Böhme and Abhik Roychoudhury. Corebench: studying complexity of regression errors. In *ISSTA*, pages 105–115, 2014.

[53] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM.

[54] David Abramson, Ian Foster, John Michalakes, and Rok Sosič. Relative debugging: A new methodology for debugging scientific applications. *Commun. ACM*, 39(11):69–77, November 1996.

[55] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[56] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 453–464, New York, NY, USA, 2009. ACM.

[57] Kenneth P. Birman. Replication and fault-tolerance in the isis system. *SIGOPS Oper. Syst. Rev.*, 19(5):79–86, December 1985.

[58] M. Chereque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in delta-4. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 28–37, July 1992.

[59] A Tulley and S.K. Shrivastava. Preventing state divergence in replicated distributed programs. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, pages 104–113, Oct 1990.

[60] Dave Black, C. Low, and Santosh K. Shrivastava. The voltan application programming environment for fail-silent processes. *Distributed Systems Engineering*, 5(2):66–77, 1998.

[61] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3):236–269, August 2003.

[62] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[63] Liming Chen and A Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 113–, Jun 1995.

[64] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Samuel Madden. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *ACM SOSP*, Stevenson, WA, October 2007.

[65] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association.

[66] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[67] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicæ for defeating memory error exploits. *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 0:434–441, 2007.

[68] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 115–124, New York, NY, USA, 2008. ACM.

[69] J. McDermott, R. Gelinas, and S. Ornstein. Doc, wyatt, and virgil: prototyping storage jamming defenses. In *Computer Security Applications Conference, 1997. Proceedings., 13th Annual*, pages 265–273, Dec 1997.

[70] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 12–12, Berkeley, CA, USA, 2007. USENIX Association.

[71] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.

[72] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM.

[73] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. *SIGPLAN Not.*, 41(11):229–240, October 2006.

[74] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[75] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM.

[76] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30(1):3:1–3:24, February 2012.

[77] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 80–95, Berlin, Heidelberg, 2006. Springer-Verlag.

[78] Subrata Mitra, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. Accurate application progress analysis for large-scale parallel debugging. In *PLDI*, page 23, 2014.

[79] William N. Sumner and Xiangyu Zhang. Memory indexing: Canonicalizing addresses across executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 217–226, New York, NY, USA, 2010. ACM.

[80] Analysis: The exploding demand for computer science education, and why america needs to keep up. `http://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/`, 2014.

[81] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.

[82] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 41–51, New York, NY, USA, 2014. ACM.

[83] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[84] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[85] Shuvendu Lahiri, Rohit Sinha, and Chris Hawblitzel. Automatic rootcausing for program equivalence failures in binaries. In *Proceedings of the 27th International Conference on Computer Aided Verification*, CAV'15, pages 362–379, Berlin, Heidelberg, July 2015. Springer-Verlag.

[86] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM.

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

[88] Stackoverflow. `http://www.stackoverflow.com`.

[89] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015.

[90] What is wrong with this algorithm? `http://stackoverflow.com/questions/18794190`.

[91] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of ACM*, 24(4):664–675, October 1977.

[92] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[93] Apex benchmarks. `http://apexpub.altervista.org/`.

[94] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.

[95] Knapsack algorithm for two bags. `http://stackoverflow.com/questions/20255319`.

[96] Incorrect result in matrix multiplication in c. `http://stackoverflow.com/questions/15512963`.

[97] Is this an incorrect implementation of kadane's algorithm? `http://stackoverflow.com/questions/22927720`.

[98] Euclid algorithm incorrect results. `http://stackoverflow.com/questions/16567505`.

[99] Dijkstra's algorithm not working. `http://stackoverflow.com/questions/14135999`.

[100] Merge sort implementation. `http://stackoverflow.com/questions/18141065`.

[101] Prims alghoritm. http://stackoverflow.com/questions/24145687.

[102] Bug in my floyd-warshall c++ implementation. http://stackoverflow.com/questions/3027216.

[103] Logical error in my implementation of dijkstra's algorithm. http://stackoverflow.com/questions/10432682.

[104] Project euler problem 4. http://stackoverflow.com/questions/7000168.

[105] Project euler 8, i don't understand where i'm going wrong. http://stackoverflow.com/questions/23824570.

[106] What is wrong with my binary search implementation? http://stackoverflow.com/questions/21709124.

[107] Inverse function works properly, but if works after while loops it produces wrong answers. http://stackoverflow.com/questions/22921661.

[108] Is there something wrong with my knapsack. http://stackoverflow.com/questions/21360767.

[109] Carlos Ansótegui, Frederic Didier, and Joel Gabàs. Exploiting the structure of unsatisfiable cores in maxsat. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI '15, pages 283–289. AAAI Press, 2015.

[110] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[111] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[112] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

[113] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, June 2006.

[114] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 266–276, New York, NY, USA, 2014. ACM.

[115] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[116] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 91–100, Austin, TX, 2011. FMCAD Inc.

[117] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI'03*, 2003.

[118] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 49–60, New York, NY, USA, 2010. ACM.

[119] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. ACM.

[120] Anne Adam and Jean-Pierre Laurent. Laura, a system to debug student programs. *Artificial Intelligence*, 15(1):75–122, 1980.

[121] William R. Murray. Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3(1):1–16, 1987.