AUTOMATED SUPPORT FOR REPRODUCING AND DEBUGGING FIELD FAILURES

A Thesis
Presented to
The Academic Faculty

by

Wei Jin

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August, 2015

Copyright © 2015 by Wei Jin

AUTOMATED SUPPORT FOR REPRODUCING AND DEBUGGING FIELD FAILURES

Approved by:

Dr. Alessandro Orso, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Mayur Naik
School of Computer Science
Georgia Institute of Technology

Dr. Taesoo Kim
School of Computer Science
Georgia Institute of Technology

Dr. Satish Chandra

Samsung Electronics

Date Approved: May 1, 2015

I dedicate my dissertation work to my wife, Chenjie Zeng. I would not be able to finish my work and pursue a doctorate without her support.

I also dedicate my dissertation to my parents. They supported and encouraged me throughout my study.

ACKNOWLEDGEMENTS

First, I would like to thank my committee members who gave me a lot of great insights and suggestions during my proposal and thesis writing. The feedbacks from the committee in the proposal help me shape my thesis.

I also want to deeply appreciate my PhD advisor, Dr. Alessandro Orso. He guided me through this challenging process and gave me endless precious advices on my research and study. His guidance helped me build up my research projects and publish papers that lead to this thesis. I could not achieve this goal without his advices.

I also want to thank my colleges and collaborators during my PhD study. Without their hard work and discussion, I would not be able to develop these techniques in my thesis and publish my papers.

I would also like to thank Georgia Tech and NSF for providing financial support during my study.

TABLE OF CONTENTS

DEDICATION . . . . **iii**

ACKNOWLEDGEMENTS . . . . **iv**

LIST OF TABLES . . . . **vii**

LIST OF FIGURES . . . . **viii**

SUMMARY . . . . **ix**

I INTRODUCTION AND MOTIVATION **1**

1.1 Thesis Statement 2

1.2 Approaches 3

1.3 Contributions 7

II BACKGROUND **9**

2.1 Symbolic Execution 9

2.2 Statistical Debugging 10

2.2.1 Ochiai 11

2.2.2 OBM (Observation-Based Model) 12

2.3 MAX-SAT Problems 13

2.4 SSA Form 14

2.5 Terminology 14

III OVERALL VISION **17**

IV AUTOMATED FIELD FAILURE REPRODUCTION **20**

4.1 My Technique for Reproducing Field Failures 20

4.1.1 Overview 20

4.1.2 Instrumenter and Analyzer Components 21

4.1.3 Execution Data 26

4.2 Empirical Investigation 27

4.2.1 BugRedux Implementation 27

4.2.2 Programs and Faults Considered 27

4.2.3 Experimental Setup 29

4.2.4 Results . . . 30
4.2.5 Discussion . . . 34
4.2.6 Limitations and Threats to Validity . . . 36
4.3 Conclusion . . . 37
V AUTOMATED FAULT LOCALIZATION FOR FIELD FAILURES . . . 38
5.1 F^3 Technique . . . 38
5.1.1 Execution Generator . . . 39
5.1.2 Fault Localizer . . . 41
5.2 Empirical Evaluation . . . 47
5.2.1 Implementation . . . 48
5.2.2 Benchmark of Study . . . 49
5.2.3 Experiment Protocol . . . 50
5.2.4 Results and Discussion . . . 51
5.3 Conclusion . . . 63
VI FURTHER IMPROVEMENT ON FAULT LOCALIZATION . . . 64
6.1 Motivation . . . 64
6.2 Improving Formula-based Debugging . . . 66
6.2.1 Clause Weighting (CW) . . . 67
6.2.2 On-demand Formula Computation (OFC) . . . 67
6.3 Preliminary Evaluation . . . 80
6.3.1 Evaluation Setup . . . 80
6.3.2 Results and Discussion . . . 82
6.3.3 Threats to Validity . . . 92
6.4 Conclusion . . . 93
VII RELATED WORK . . . 94
7.1 Techniques for Reproducing Field Failures . . . 94
7.2 Fault Localization Techniques . . . 96
7.3 Formula-based Fault Localization Techniques . . . 98
VIII CONCLUSION AND OPEN PROBLEMS . . . 100
REFERENCES . . . 103

LIST OF TABLES

1 Benchmark programs used in my study 28

2 Time (%) and space (KB) overhead imposed by BugRedux 31

3 Effectiveness and efficiency of BugRedux in synthesizing executions starting from collected execution data 32

4 Minimal number of entries in call sequences that are needed to reproduce observed failures 35

5 Programs from SIR and exploit-db used in my study 50

6 Number of failing and passing executions generated by F^3 51

7 Path Similarity among executions generated by my approach 54

8 Ranks of the faulty entity using F^3 56

9 Total number of branches exercised by the synthesized passing and failing executions and size of the corresponding filtering sets 59

10 Positions of the faulty entity in the ranked list produced by traditional Ochiai, Ochiai with three filters, traditional OBM, and OBM with three filters . . . 60

11 Positions of the actual faults in the ranked list produced by Ochiai with and without profiling information 61

12 Positions of the actual faults in the ranked list produced by the fault localizer when using the original Ochiai and OBM techniques with and without grouping 62

13 Results for BA and BA+CW when run on tcas 83

14 Performance results for BA and OFC on tcas 85

15 Average time for processing tcas faults 86

16 Ranking results of OFC+CW on tot_info 88

17 Results for OFC+CW when run on Redis's bug 88

18 Percentage of CPU time spent in the solvers when running OFC on tcas . . 91

LIST OF FIGURES

1 Simple code example to illustrate symbolic execution 10

2 Overall vision of my techniques 18

3 High-level overview of BugRedux 21

4 The analysis component of BugRedux 22

5 High-level overview of F^3 38

6 A code snippet containing a buffer-overflow bug 45

7 Overview of on-demand formula computation 68

8 Example code in normal (left) and SSA (right) form 74

9 Control flow graph of P (a) and partial P considered during the first (b) and second (c) iteration of OFC 74

10 Excerpt code of the bug in Redis 89

11 Correlation between the percentage of CPU time spent in the solvers and the total CPU time for our OFC technique 92

SUMMARY

Software testing activities are generally insufficient in house due to time and resource limitations. As a result, deployed software is bound to contain bugs and will eventually misbehave in the field, which will result in field failures—failures that occur on user machines after the deployment of software. As confirmed by a recent survey conducted among developers of the Apache, Eclipse, and Mozilla projects, two extremely challenging tasks during maintenance are *reproducing* and *debugging* field failures. Unfortunately, the information typically contained in traditional bug reports, such as memory dumps or crash call stacks is usually insufficient for reproducing the observed field failure, which seriously hinders developers' ability to debug such failures.

To address and mitigate the problems of reproducing and debugging field failures, I first present an overall approach that comprises two techniques, BugRedux and F^3, in this dissertation. BugRedux is a general technique for reproducing field failures that collects dynamic data about failing executions in the field and uses this data to synthesize executions that mimic the observed field failures. F^3 leverages the executions generated by BugRedux to perform automated debugging using a set of suitably optimized fault-localization techniques. To assess the usefulness of my overall approach, I performed two empirical studies of my approach on a set of *real-world programs and field failures.* The results of the evaluation are promising in that, for all the failures considered, my approach was able to (1) synthesize failing executions that mimicked the observed field failures, (2) synthesize passing executions similar to the failing ones, and (3) use the synthesized executions successfully to perform fault localization with accurate results.

The results of my studies and the observations that I made in BugRedux and F^3 lead to another goal of my dissertation—providing a principled and efficient way to identify potentially faulty statements together with information that can help fixing such statements. To achieve this goal, I propose two techniques to improve the overall efficiency

of formula-based debugging, a principled debugging technique. In particular, on demand formula computation (OFC) improves by exploring all and only the parts of a program that are relevant to a failure. Clause Weighting (CW) improves the accuracy of formula-based debugging by leveraging statistical fault-localization information that accounts for passing tests. The results of the show that, although my techniques are only a first step towards making formula-based debugging more applicable, both of them are effective and can improve the state of the art.

CHAPTER I

INTRODUCTION AND MOTIVATION

Quality-assurance activities, such as software testing and analysis, are notoriously difficult, expensive, and time-consuming. As a result, software products are typically released with faults or missing functionality. The characteristics of modern software are making the situation even worse. Because of the dynamic nature, configurability, and portability of today's software, deployed applications may behave very differently in house and in the field. In some cases, these different behaviors may be totally legitimate behaviors that simply were not observed during in-house testing. In other cases, however, such behaviors may be anomalous and result in *field failures*, failures of the software that occur after deployment, while the software is running on user machines.

Field failures are both difficult to foresee and difficult, if not impossible, to reproduce outside the time and place in which they occurred. In fact, a recent survey among developers of the Apache, Eclipse, and Mozilla projects revealed that most developers consider information on how to reproduce failures (*e.g.,* stack traces, steps to follow, and ideally even inputs) to be the most valuable and difficult to obtain piece of information in a bug report [88]. This pressing need is demonstrated by the emergence, in the last decade, of several reporting systems that collect information (*e.g.,* stack traces and register dumps) when a program crashes and send it back to the software producer (*e.g.,* [3,6,10]). Although useful, the information collected by these systems is often too limited to allow for reproducing a failure and is typically used to identify correlations among different crash reports or among crash reports and known failures [17].

Researchers have also investigated more sophisticated techniques for capturing data from deployed applications that can help debugging (*e.g.,* [25, 28, 37, 46, 58, 59, 69]). Among these techniques, some collect only limited amounts of information (*e.g.,* sampled branch profiles for CBI [58, 59]). These techniques have the advantage of collecting types of data

that are unlikely to be sensitive, which makes them more likely to be accepted by the user community. Moreover, given the amount of information collected, it is conceivable for users to manually inspect the information before it is sent to developers.

Unfortunately, subsequent research has shown that the usefulness of the information collected for debugging increases when more (and more detailed) data is collected. Researchers have therefore defined novel techniques that gather a wide spectrum of richer data, ranging from path profiles to complete execution recordings (*e.g.,* [4,25,28,50]). Complete execution recordings, in particular, can address the issue of reproducibility of field failures. User executions, however, have the fundamental drawbacks that they (1) can be expensive to collect and (2) are bound to contain sensitive data. While the former issue can be alleviated with suitable engineering (*e.g.,* [4,28]), the latter issue would make the use of these techniques in the field problematic. Given the sheer amount of data collected, users would not be able to manually check the data before they are sent to developers, and would therefore be unlikely to agree on the collection of such data. Although some techniques exist whose goal is to sanitize or anonymize collected data, they are either defined for a different goal, and would thus eliminate sensitive data only by chance (*e.g.,* [5, 85]), or are still in their early phase of development and in need of a more thorough evaluation (*e.g.,* [22, 30]).

In addition to the problem of recreating failures observed in the field, many of these techniques do not provide any explicit support for understanding the recreated failures, if they are able to, and their causes. Developers are therefore left with no alternative but to perform traditional manual debugging.

1.1 Thesis Statement

To address some of the existing problems in reproducing and debugging field failures, my thesis is to design several different automated techniques that (1) allow developers to investigate field failures by reproducing them in a faithful way, (2) help developers identify and understand causes of field failures by improving fault localization techniques, and (3) ultimately help developers eliminate these causes by providing more useful explanations of failures and more actionable fault localization reports that can better support program

repair techniques.

1.2 Approaches

To achieve the first two parts of my thesis statement, I developed techniques that aim to reproduce field failures and perform fault localization to address the limitations of existing techniques while only imposing limited overhead on the users and avoiding violating the users' privacy.

More precisely, I first aim to develop a set of general techniques that can synthesize, given a program P, a field execution E of P that results in a failure F, and a set of execution data D for E, multiple failing executions *FAIL* and passing executions *PASS* as follows. *First*, *FAIL* should result in a failure F' that is analogous to F, that is, F' has the same observable behavior of F. If F is the violation of an assertion at a given location in P, for instance, F' should violate the same assertion at the same point. *Second*, *PASS* should not trigger any failure of P or not violate any oracle for P. *Third*, *FAIL* and *PASS* should be actual executions of P, that is, the approach should be sound and generate actual inputs that, when provided to P, results in the synthesized executions. *Fourth*, the approach should be able to generate *FAIL* and *PASS* using only P and D, without the need for any additional information. *Finally*, D should not contain sensitive data and should be collectible with low overhead on E.

As a first step towards my goal, in Chapter 4, I present BugRedux, a general technique for (1) collecting different kinds of execution data and (2) using the collected data to synthesize executions that can reproduce failures observed in the field. Intuitively, BugRedux can be seen as a general framework parameterized along two dimensions: the kind of execution data collected and the technique used for synthesizing a failing execution. I present four variations, or instances, of BugRedux that all share the same synthesis technique (*i.e.,* symbolic execution) but differ in the kind of execution data they use. Specifically, I consider four types of increasingly rich execution data: points of failure, call stacks, call sequences, and complete program traces.

To address the problem of lack of automated debugging support, in Chapter 5, I present

another technique called F^3 (Fault localization for Field Failures), which extends BugRedux by adding to it support for automated debugging.

I devised F^3 based on several observations made in my failure reproduction work and automated debugging in general. The *first observation* is that the most popular statistical fault localization techniques (*e.g.,* [12,53,58]) rely on the existence of numerous passing and failing executions and cannot be applied to individual executions in isolation. This can be a serious limitation in practice because test suites are often of limited size and, especially, rarely contain multiple failing tests for the same failure. I defined F^3 so that it can suitably address this limitation. Specifically, I extended BugRedux so that (1) it generates not one failing execution, but rather a set of executions that "mimic" a failing field execution E and (2) this set includes both passing and failing executions, such that the failing executions fail for the same reasons as E, and the passing executions should be "similar" to E. To do so, I first modified the execution generator in BugRedux so that it tries to synthesize as many executions as possible for a given set of crash data (*i.e.,* list of goals). In case this set of executions does not contain any passing one, the algorithm starts eliminating goals from the list and tries again to synthesize passing executions. Eliminating goals increases the degrees of freedom of the synthesis, thus increasing the chances of generating passing executions, at the cost of reducing the similarity between the synthesized executions and E. The *second observation* I made from many other researchers' previous work is that the output of traditional statistical fault-localization techniques, a long list of program entities ranked based on their likelihood of being faulty, may be of limited usefulness to developers [68]. Developers tend to give up when the faulty program entity is not among the first ones in the list, which is often the case—even when the number of program entities to inspect before finding the fault is less than 5% of the program, which is considered a good result in most fault-localization literature, that percentage could correspond to hundreds of program entities in any non trivial program. To address this second limitation, in F^3, I tailored traditional statistical fault-localization techniques by adding to them several customizations and optimizations well suited to the sets of executions generated by my

approach and the failures I target. More precisely, I selected four well known state-of-the-art fault-localization techniques—Ochiai [11], Observation-Based Model (OBM) [12], Nashi1, and Naish2 [63]—as baseline techniques and defined three optimizations for these techniques, namely, profiling, filtering, and grouping, based on my experience with field failures and my preliminary investigation.

Despite F^3 can effectively localize faults of field failures, it still has some limitations that are inherited from statistical fault localization. First, the granularity of the final reports generated by F^3 is limited to the basic block level because that is the finest level that can be achieved by leveraging dynamic coverage or profiling information without combining other debugging techniques. In other words, program entities that come from the same basic block normally share the same suspiciousness values in the final report generated by F^3. Second, similar to traditional statistical fault localization techniques, the final report generated by F^3 provides limited contextual information of the failure as it only provides a list of program entities in decreasing order of "suspiciousness" (likelihood of being related to the failure). As a result, developers have to examine single entities to identify the causes of the failure without any context, which is very difficult in most cases. Therefore, to mitigate these limitations in statistical fault localization, there has been a considerable interest in techniques that can perform fault localization in a more principled way recently (*e.g.,* [26, 39, 54, 76]). These techniques, collectively called *formula-based debugging*, model faulty programs and failing executions as formulas and perform fault localization by manipulating and solving these formulas. As a result, they can provide developers with the possible location of the fault, together with a mathematical explanation of the failure (*e.g.,* the fact that an expression should have produced a different value or that a different branch should have been taken at a conditional statement).

Besides presenting these approaches, I also performed several empirical studies to assess my approaches. For BugRedux, I performed an empirical study in which I assessed the trade-offs that characterize the variations of BugRedux with respect to (1) the cost of the data collection, in terms of space and time overhead (and, indirectly, likelihood to contain sensitive data), and (2) the ease of synthesizing a failing execution starting from such data.

In the evaluation, I used an implementation of BugRedux developed for the C language and applied it to 16 failures of 14 real-world programs. For each failure, I collected the four different types of execution data, measured the overhead of the collection, and tried to synthesize an execution that reproduced the failure using such data. Interestingly, my results show that the richest data, beside being the most expensive to collect and the most problematic in terms of potential privacy violation, is not necessarily the most useful when used for synthesizing executions. My results also confirm that, at least for the cases I considered, information that is traditionally collected by crash-report systems, such as the call stack at the point of failure, is typically not enough for recreating field failures.

For the current incarnation of BugRedux, I found that the best option in terms of cost-benefit ratio is the use of call sequences. As the study in BugRedux shows, using partial call-sequence data BugRedux was able to recreate all of the 16 failures considered, while imposing an acceptable time and space overhead. This result actually led us to my second technique F^3, in which I directly used the identified partial call-sequence data as relevant crash data.

To validate F^3, I implemented it in a prototype tool and used the tool to perform another empirical study on a set of programs and failures that I selected from the first study. In my second study, I assessed (1) whether F^3 is actually able to synthesize multiple passing and failing executions for a given set of crash data, (2) whether these synthesized executions can be used for fault localization, (3) the degree of similarity between the synthesized passing and failing executions and how similarity affects fault localization, and (4) whether my optimizations actually improve the effectiveness of fault localization and to what extent. The results of my study on F^3 are also promising in that, for all the cases considered, F^3 was able to synthesize sets of similar passing and failing executions and use these synthesized executions to perform effective fault localization. The results also show that my optimized fault-localization techniques can mitigate the limitations of their traditional counterparts, at least when applied to the particular set of executions synthesized by F^3.

To assess the effectiveness of OFC and CW, I selected BugAssist as a baseline and considered four different formula-based debugging techniques: the original BugAssist, BugAssist+CW, OFC, and OFC+CW. We implemented all four techniques in a tool that works on C programs and used the tool to perform an empirical study. In the study, I first applied the four techniques to 52 versions of two small programs to assess several tradeoffs involved in the use of CW and OFC and compare with related work. Our results are encouraging, as they show that CW and OFC can improve the performance of BugAssist in several respects. First, the use of CW resulted in more accurate results—in terms of position of the actual fault in the ranked list of statements reported to developers—in the majority of the cases considered. Second, CW and OFC were able to reduce the computational cost of BugAssist by 27% and 75% on average, respectively, with maximum speedups of over 70X for OFC. To further demonstrate the practicality of CW and OFC, I also performed a case study on a real-world bug in Redis, a popular open source project. Overall, the results show that CW and OFC are promising, albeit initial, steps towards more practically applicable formula-based debugging techniques and motivate further research in this direction.

1.3 Contributions

This thesis dissertation provides the following novel contributions:

- Two general approaches for collecting execution data in the field and leveraging the data to synthesize executions that reproduce field failures and perform customized statistical fault localization on the synthesized executions.

- The definition of clause weighting and on-demand formula computation, two approaches for improving the accuracy and efficiency of formula-based debugging.

- The implementation of all techniques in three prototype tools.

- An empirical study that performs a cost-benefit analysis of BugRedux and its variations in terms of data collection costs and ability to synthesize failing executions.

- A set of empirical studies that show that F^3 can be effectively used on real-world failures and faults.

- Initial empirical evidence that CW and OFC are as effective and more efficient than existing approaches.

The rest of the thesis is organized as follows. Chapter 2 provides some necessary background information and defines some relevant terminology. Chapter 3 introduces my overall vision of achieving my overarching goals. Chapter 4 describes the details of my field-failure reproduction technique, BugRedux, and presents the first empirical evaluation. Chapter 5 describes the details of my fault-localization approach for field failures, F^3, and presents the second empirical evaluation. Chapter 6 discusses the details of the two techniques to improve formula-based debugging, CW and OFC, and the initial evaluation of the techniques. Chapter 7 puts my research in context by discussing related work. Finally, Chapter 8 concludes the thesis and discusses some future opening problems related to the thesis.

CHAPTER II

BACKGROUND

Before discussing my approaches, in this chapter, I briefly provide some necessary background information on symbolic execution, statistical fault localization, MAX-SAT, and the SSA form and define some terms that I use in the rest of my thesis proposal.

2.1 Symbolic Execution

In its most general formulation, symbolic execution is a technique that executes a program using symbolic instead of concrete inputs. At any point in the computation, the program state consists of a *symbolic state* expressed as a function of the inputs; and the conditions on the inputs that result in the execution to reach that point are expressed as a set of constraints in conjunctive form called the *path condition (PC)* [55]. More formally, the symbolic state can be seen as a map $\mathcal{S} : \mathcal{M} \mapsto \mathcal{E}$, where $\mathcal{M}$ is the set of memory addresses for the program, and $\mathcal{E}$ is the set of possible symbolic values, that is, expressions in some theory $\mathcal{T}$ such that all free variables are input symbolic values.

Both the symbolic state and the PC are built incrementally during symbolic execution, with PC initialized to `true`, each input expressed as a symbolic variable, and $\mathcal{S}$ initialized according to the semantics of the language. (In C, for instance, memory addresses not yet initialized could be mapped to $\perp$ to indicate that they are undefined.) Every time a statement *stmt* that modifies the value of a memory location m is executed, the new symbolic value e' of m is computed according to *stmt*'s semantics, and $\mathcal{S}$ is updated by replacing the old expression for m with e' ($\mathcal{S}' = \mathcal{S} \oplus [m \mapsto e']$, where $\oplus$ indicates an update). Conversely, when a predicate statement *pred* that modifies the flow of control is executed, symbolic execution forks and follows both branches. Along each branch, the PC is augmented with an additional conjunct that represents the input condition, expressed in terms of symbolic state, that makes the predicate in *pred* `true` or `false` (depending on the branch).

```
    function foo(int a, int b, int c) {
 1. int d = a + 4
 2. if (d < b)
 3.    //do something
 4. if (b > 5)
 5.    //do something
 6. else if (a < 5)
 7.   if (d < c)
 8.     //do something
 9.   else
10.     //do something
11. else
12.   //do something
13. return
}
```

Figure 1: Simple code example to illustrate symbolic execution

Symbolic execution, when successful, can be used to compute an input that would cause a given path to be executed or a given statement to be reached. To do so, at program exit or at a point of interest in the code, the PC for that point would be fed to an SMT solver, which would try to find a solution for PC. Such a solution would consist of an assignment to the free variables in PC (*i.e.,* the inputs) that satisfies PC. If such a solution is found, the corresponding concrete input is exactly the input that causes the path to be executed. To illustrate symbolic execution with an example, consider the code snippet in Figure 1. I indicate the symbolic inputs for the parameters a, b, and c with a_0, b_0, and c_0. When symbolic execution follows path $\langle 1, 2, 3, 4, 6, 7, 8, 13\rangle$, for instance, the symbolic state at statement 13 is $\{[a \mapsto a_0], [b \mapsto b_0], [c \mapsto c_0], [d \mapsto a_0 + 4]\}$, and the corresponding PC would be $(a_0 + 4 < b_0) \wedge (b_0 <= 5) \wedge (a_0 < 5) \wedge (a_0 + 4 < c_0)$, which corresponds to the conjunction of the predicates for branches $2T$, $4F$, $6T$, and $7T$. A possible solution for this PC is the set of assignments $a_0 = 0$, $b_0 = 5$, and $c_0 = 5$, which correspond to an input $i = \{0, 5, 5\}$ that causes path $\langle 1, 2, 3, 4, 6, 7, 8, 13\rangle$ to be followed.

2.2 Statistical Debugging

Given the high cost of manual debugging, researchers have investigated and proposed a countless number of automated debugging techniques that can support developers in their

debugging tasks. Fault localization in particular, the task of locating the faulty code entities responsible for a failure in a program, has received a great deal of attention in the last decade (*e.g.,* [12, 19, 31, 53, 58]). In F^3, I am interested in using statistical approaches to perform fault localization, also called statistical debugging techniques. At a high level, these approaches work by observing the behavior of a (ideally) large number of passing and failing executions, performing statistical inference based on the observed behavior, and using the results of such inference to rank program entities in terms of their likelihood to be related to the failure (*i.e.,* their suspiciousness). The program entities would then be presented to the developers in decreasing order of suspiciousness, and the developers would examine every entity in that order (again, ideally) and assess whether that entity is faulty.

In F^3, I considered various statistical fault-localization approaches. Among the numerous approaches presented in the literature, I chose four representative and well-known fault localization techniques: Ochiai [11] and OBM [12] by Abreu and colleagues and two optimal models by Naish and colleagues [63]. (Note that I do not try to consider as many fault-localization techniques as possible, as my goal is to show that fault localization can successfully be used on and optimized for the synthesized executions generated by my approach.) I choose Ochiai because previous research showed, both empirically and analytically, that it is quite effective compared to other traditional techniques that use similar metrics [11]. OBM, conversely, is a good representative of a different family of techniques that focus on models of the program behavior. In addition to these two techniques, I also consider two other techniques by Naish and colleagues that have been shown to work better than Ochiai in many cases [63, 79]. Since these statistical fault localization techniques focus on different properties of dynamic executions, they may potentially generate different ranked lists of suspicious statements for the same failure. I will quickly summarize these four approaches to make the thesis self contained.

2.2.1 Ochiai

Ochiai [11] is a spectra-based fault-localization technique that leverages coverage information in passing and failing runs to compute the suspiciousness of program entities and

rank them accordingly. The specific formula used by Ochiai was previously used for computing genetic similarity in molecular biology. Like most spectra-based fault-localization techniques, Ochiai assigns a suspiciousness value to each program entity based on coverage information and on the following intuition: the higher the number of failing runs that executed a program entity, the higher its suspiciousness; conversely, the higher the number of passing runs that executed a program entity (or the higher the number of failing runs that did not execute a program entity), the lower its suspiciousness. Accordingly, the formula used by Ochiai to compute suspiciousness for a program entity *en* is the following (using the same notation used in Reference [11]):

$$suspiciousness(en) = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \times (a_{11} + a_{10})}} \qquad (1)$$

In the formula, a_{11} indicates the number of failing executions that exercised *en*, a_{01} indicates the number of failing executions that did not exercise *en*, and a_{10} indicates the number of passing executions that exercised *en*. In theory, the approach can be instantiated for any type of program entity (*e.g.,* statements, functions, branches, or predicates).

2.2.2 OBM (Observation-Based Model)

OBM [12] models the executions in a way that is different from that used in traditional spectra-based fault-localization techniques. Specifically, OBM considers sets of one or more entities in the program as independent diagnoses d_k for a failure f (*i.e.,* the entities in a d_k set represent possible causes of f). Each execution is then treated as an observation *obs* of the system that can be used to confirm or refuse a set of diagnoses. To do so, OBM uses the following Bayesian formula, which computes the conditional probability of each diagnosis to be a possible cause of the failure:

$$Pr(d_k|obs) = \frac{Pr(obs|d_k)}{Pr(obs)} Pr(d_k). \qquad (2)$$

Since OBM assumes that program entities fail independently, the prior probability that a diagnosis is correct, $Pr(d_k)$, and the prior probability that an execution is observed,

$Pr(obs)$, are constants and identical for all d_ks. The posterior probability for each observation conditional to a diagnosis, $Pr(obs|d_k)$, is computed based on whether *obs* is a passing or a failing run. The conditional probability of each individual diagnosis d_k is the product of $Pr(d_k|obs)$ over all observations, as observations are considered to be independent. These conditional probabilities represent the likelihood of a diagnosis to contain the causes of the failure and can be used to rank diagnoses.

2.2.2.1 Naish Models

In a recent in-depth investigation of a wide range of statistical fault localization techniques [63], Naish, Lee and Ramamohanarao proposed two novel statistical fault localization techniques. I refer to them as Naish1 and Naish2 hereafter. Similar to Ochiai, Naish1 and Naish2 also compute suspiciousness scores based on the same intuition and leverage similar dynamic coverage information. In particular, Naish1 uses the following metric to compute suspiciousness:

$$suspiciousness_{Naish1}(en) = \begin{cases} 0 & if\ a_{nf} > 0 \\ a_{np} & otherwise \end{cases}$$

Naish2 uses another optimal metric:

$$suspiciousness_{Naish2}(en) = a_{ef} - \frac{a_{ep}}{a_{ep} + a_{np} + 1}$$

In these two formulas, I use the original notation used in Reference [63]. Here, a_{nf} stands for the number of failing executions that did not exercise *en* (*i.e.,* a_{01} in Ochiai), a_{np} stands for the number of passing executions that did not exercise *en*, a_{ef} stands for the number of failing executions that exercised *en* (*i.e.,* a_{11} in Ochiai), and a_{ep} stands for the number of passing executions that exercised *en* (*i.e.,* a_{10} in Ochiai).

2.3 MAX-SAT Problems

MAX-SAT is the problem of determining the maximum number of clauses of a given unsatisfiable Boolean formula that can be satisfied by some assignment [20]. An extension of MAX-SAT is pMAX-SAT, in which clauses are marked as either hard (*i.e.,* clauses that

cannot be dropped) or soft (*i.e.,* clauses that can be dropped). wpMAX-SAT extends pMAX-SAT by assigning weights to soft clauses, such that clauses with higher weights are less likely to be dropped. A solution to a wpMAX-SAT problem is a maximal satisfiable subset of clauses (MSS) with maximum weight in which all hard clauses are satisfied. The complement of MSS is called CoMSS. MSS is defined as a maximal set of clauses, in the sense that adding any of the other clauses in CoMSS would make the set unsatisfiable. The maximal property of MSS and the minimal property of CoMSS essentially imply that clauses in CoMSS are responsible for making the formula unsatisfiable. There may be several different maximal satisfiable subsets and complementary sets for a given MAX-SAT problem, and each of these sets can contain multiple clauses.

2.4 SSA Form

Given a program P, the static single assignment (SSA) form of P is a program semantically equivalent to P in which each variable is assigned exactly once [33]. Because multiple definitions can reach a join point, for each conditional statement *cs*, the SSA form contains one ϕ function *phi* for each definition d in the original program that is control dependent on *cs* and can reach *cs*'s join point. *phi* is located at the join point and selects the correct definition to use at that point depending on which branch of *cs* was executed. I refer to conditional statement *cs* as *phi*'s *conditional*.

2.5 Terminology

In this section, I will define some terms that I am going to use in the rest of my thesis proposal.

A *control flow graph (CFG)* for a function f is a directed graph $G = \langle N, E, entry, exit \rangle$ where N is a set of nodes that represent statements in f and $E \subseteq N \times N$ is a set of edges that represent the flow of control between nodes, and $entry \in N$ and $exit \in N$ are the unique entry and exit points, respectively, for the CFG.

An *interprocedural control flow graph (ICFG)* is a graph built by composing a set CFGs. To build an ICFG, CFGs are connected based on call relationships between the functions they represent. If a function f_1 calls a function f_2, the two CFGs for f_1 and f_2, G_1 and G_2,

are connected as follows: the node n in f_1 representing the call site to f_2 is replaced by two nodes n_c (call node) and n_r (return node), such that all predecessors of n are connected to n_c, and n_r is connected to all successors of n. Then, n_c is connected to G_2's entry node, and G_2's exit node is connected to n_r. This process is repeated for every call site in the program.

A *conditional statement* is a statement whose node has two successors (*branches*) and contains a Boolean *predicate*, whose value determines which branch the execution will follow. (For ease of presentation, I assume that conditional statements with more than two branches, such as switch statements, are suitably transformed into several conditional statements with just two branches.) Given a conditional statement and a branch for that statement, I refer to the other branch for the same statement as the *alternative branch*.

Given a program P, a failing execution E of P for a given input I, and the resulting failure F, I define the following terms. I call F a *field failure* if it occurred on a user machine, after P has been deployed. A *point of failure (POF)* is the statement in P where F manifests itself. For the sake of the discussion, and without loss of generality, I assume that a failure corresponds to a failing assertion, and that POF is the statement in which the assertion fails (all failure conditions can be expressed in the form of assertions in the code). A *failure call stack* for F is the ordered list of functions that were on the call stack when F occurred. Each entry in the list consists of a function and a location in the function (*i.e.,* either the location of the call to the next function in the list or, for the last entry, the location of the failure). In the rest of the thesis proposal, I refer to the failure call stack for F as F's call stack or simply call stack, except for cases where the term may be ambiguous. A *call sequence* for E is the sequence of calls executed (*i.e.,* call sites traversed) during E. A *complete trace* for E is the sequence of all branches (*i.e.,* program predicates and their outcomes) exercised during E. Obviously, complete traces subsume call sequences. I use the term *execution data* for E to refer to any dynamic information collected during E. Therefore, call sequences and complete traces are examples of execution data. Finally, a *crash report* for F is a record that is produced when F occurs and can be later sent to P's developers. Although crash reports can have different formats and contents, I assume

that a crash report contains at least a POF and a call stack, and possibly some additional execution data.

CHAPTER III

OVERALL VISION

In this chapter, I will provide the high-level overall vision of achieving the overarching goals in my thesis statement and addressing the need of automated support for reproducing and debugging field failures in a faithful way.

Figure 2 provides the high-level vision to achieve the goals. As the figure shows, there are three main stages in this overall vision. The stages and generated artifacts on the left side of the dashed line are generally handled in house, while the ones on the right side of the line normally happen in the field.

The first stage is *Instrumentation*, in which my approach takes a software application built by software developers as input and generates an instrumented version of the application as the outcome artifact of the first stage before releasing this application into the field. During the instrumentation, probes will be inserted into certain places of the application so that when the released application runs in the field, these probes can collect execution data and add such execution data into crash reports only when field failures occur. The instrumentation stage can be instantiated in several different ways based on different types of dynamic execution data to be collected in the field. I will investigated the use of several instances of execution data and evaluated their effectiveness in terms of reproducing and debugging field failures in Chapter 4 and Chapter 5. Since instrumentation is a well assessed and straight forward technique, in my overall approach, I decide to directly leverage traditional instrumentation techniques in order to only pose limited overhead into the released application. The detailed implementation of the instrumentation stage can be found in the implementation section of Chapter 4.

The other two stages are more important and challenging than Instrumentation and they are the core components of my research in this thesis dissertation. The second stage is *Field Failure Reproduction*, in which my approach takes the generated crash report as

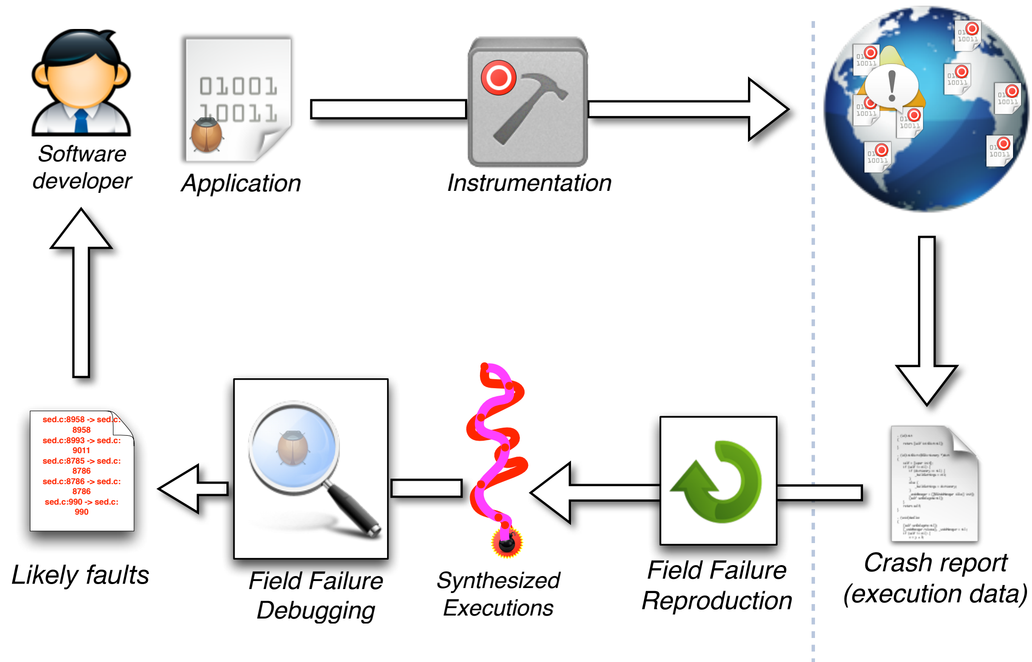

Figure 2: This is the overall vision of my techniques that contains three main steps.

input and tries to synthesize a set of failing executions of the original application that "mimic" the original field failures observed in the field and a set of passing executions that are "similar" to the original field execution. I will give the exact definitions of "mimic" and "similar" in detailed techniques in Chapter 4 and Chapter 5. These generated executions are real executions of the application, and thus, the outcomes of this stage are a set of passing and failing inputs that can result in the generated executions when provided to the application. There are multiple possible types of techniques to generate these executions in this framework. In particular, I chose a guided symbolic execution algorithm and I will discuss my general field failure reproduction technique that is based on this customized algorithm in Chapter 4. This general technique can be also instantiated in different ways, like the instrumentation stage, based on which types of execution data are available after the first stage—Instrumentation.

The third stage is *Field Failure Debugging*, in which my approach takes the generated executions from the previous step as input, performs fault localization to compute a list of likely faults and reports this list to software developers so as to help them understand and identify the causes of the failure. In this framework, I defined two different fault localization

techniques to proceed this step. In particular, I first present a customized spectra-based statistical fault localization technique for field failures in Chapter 5. By leveraging the generated "similar" passing and failing executions, I defined a customized statistical fault localization techniques with several optimizations that are based on previous observations. In this statistical fault localization technique, the output are a list of program entries with suspiciousness values. To further improve the effectiveness of fault localization, I also present two novel techniques to improve formula-based debugging in Chapter 6. I define these two techniques that can address several effectiveness and efficiency issues of previous techniques by performing the formula-based fault localization in an on-demand manner. The extended formula-based debugging technique can provide not only suspicious locations in the program but also some mathematical explanations of the failure. Therefore, these results and explanations may later help automated patch generation generate valid bug fixes.

Since the last two stages are designed to be completely automated without any help of users or developers, they can be performed on either users' machines or in house depending on the free computing resources available on users' machines. If these stages are completed on users' side, my approach can directly send the generated list of likely faults back into house as the outcome of these two stages. Otherwise, the generated crash report will be sent back in house as the input of these two stages and these two stages can be carried out in house in this case.

In summary, in this chapter, I proposed an overall vision to address the need of automated support for reproducing and debugging field failures and mitigate problems introduced by previous techniques that tried to solve these two problems. I will present all techniques mentioned in this overall vision (*i.e.,* a field failure reproduction technique, a statistical based field failure debugging technique, and two techniques for improving formula-based debugging) and empirical studies that assess these techniques in the following chapters.

CHAPTER IV

AUTOMATED FIELD FAILURE REPRODUCTION

4.1 My Technique for Reproducing Field Failures

As stated in the Chapter 3, the first two processes of my overall approach are to recreate field failures faithfully (*i.e.,* in a way that allows for investigating and debugging them) by using execution data collected in the field that can be gathered without imposing too much space and time overhead on field executions. A first important step towards defining a technique for reproducing field failures is to understand the usefulness of different kinds of execution data in this context. To this end, I defined a general technique for synthesizing executions that (1) mimic executions that resulted in field failures and (2) faithfully] reproduce such failures. I instantiated several variants of my technique that differ in the kind of execution data they use, and studied the effectiveness of these different variants. In the rest of this chapter, I will discuss my field failure reproduction technique and present an empirical investigation on this technique.

4.1.1 Overview

My general technique for reproducing field failures is called BugRedux. Intuitively, BugRedux operates by (1) collecting different kinds of execution data and (2) using the collected data to synthesize real executions that reproduce failures observed in the field. Figure 3 provides a high-level overview of BugRedux and of the scenario I target.

As the figure shows, BugRedux consists of two main components. The first one is the *instrumenter*, which takes as input an *application* provided by a *software developer* and generates an *instrumented application* that can collect execution data and add such execution data to *crash reports* from the field. The second component is the *analyzer*, which takes as input a *crash report* and tries to generate a *test input* that, when provided to the *application*, results in the same failure that was observed in the field. A *software tester* can then use the generated input to recreate and investigate the field failure through some

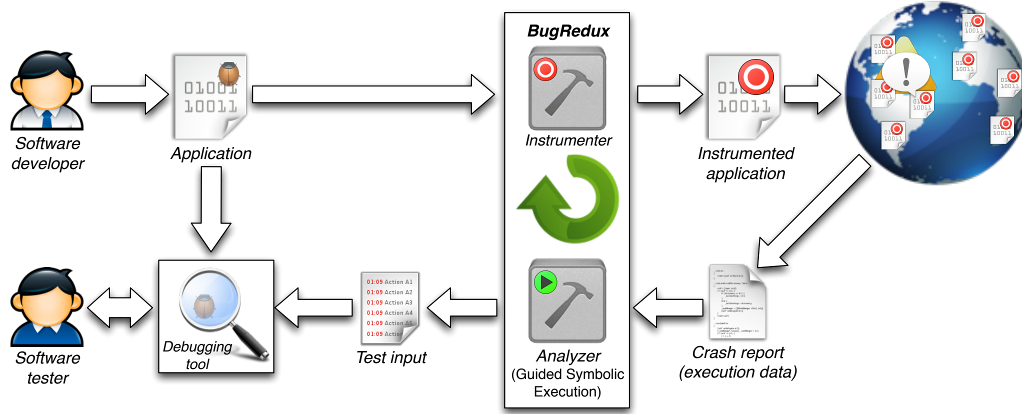

Figure 3: This is the intuitive high-level overview of BugRedux.

debugging approaches. This general technique can be defined in different ways depending on the kind of execution data collected and on the technique used for synthesizing execution.

4.1.2 Instrumenter and Analyzer Components

Instrumentation is a well assessed technology, so I do not discuss this part of the technique further. It suffices to say that BugRedux adds probes to the original program that, when triggered at runtime, generate the execution data of interest. Conversely, the analyzer is the core part of the technique and the most challenging to develop. Figure 4, which provides a more detailed view of the analysis component of BugRedux, puts the problem in context and lets me discuss how I addressed this challenge. As the figure shows, the inputs to the analyzer are an application program P, whose execution E produces failure F that I want to reproduce, and a crash report C for F. The goal of the analyzer is to generate a test input that would result in an execution E' that "mimics" E and would fail in the same way.

Given crash report C, the *input generator* would analyze program P and try to generate such test input. The exact definition of *mimicking* depends on the amount of information about the failing execution E that is available. If only the POF were available, for instance, E' would mimic E if it reaches the POF. Conversely, if a complete trace were to be used, E' would have not only to reach the POF but also to follow the same path as E. This concept of mimicking is defined within the *input generator*, which receives the execution data in the form of a sequence of goals (or locations in the program) to be reached and tries to generate

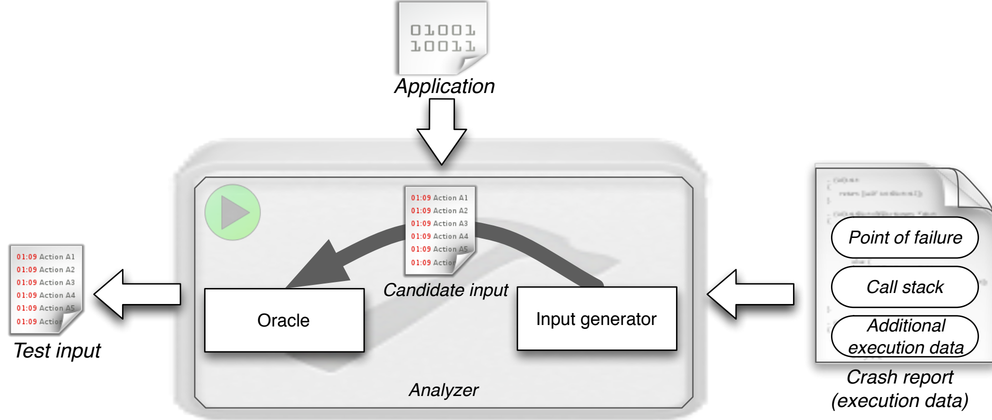

Figure 4: This is the detailed view of analysis component of BugRedux.

executions that reach such goals in the right order. If successful, the *input generator* would generate a candidate input, and the *oracle* would check whether that input actually fails in the same way as E.

In theory, any automated input generation technique (*e.g.,* symbolic execution [55], weakest precondition analysis [35], or genetic algorithms [47]) could be used in this context, as long as it can be guided towards a goal (*e.g.,* the point of failure, the entry point of a function on the failure's call stack, or a branch within the program). In this part of my overall technique, I decided to use an approach based on symbolic execution [55]. Specifically, I use a symbolic execution algorithm customized with an ad-hoc search strategy that leverages the execution data available expressed as a set of goals. My algorithm, GenerateInputs, is shown in Algorithm 1. GenerateInputs takes as input $icfg$, the Interprocedural Control Flow Graph (ICFG) [13] for program P, and $goals_list$, an ordered list of statements to be reached during the execution. (The exact content of $goals_list$ is discussed in Section 4.1.3.)

Initially, GenerateInputs performs some initializations (lines 2–4). First, it initializes sym_state_0 with the initial symbolic state, where all inputs are marked as symbolic. Then, it initializes $states_set$, a set that will be used to store search states during the execution, with the initial search state. Entries in $states_set$ are quadruples $\langle cl, pc, ss, goal \rangle$, where cl is a code location, pc the Path Condition (PC) for the path followed to reach location cl, ss the symbolic state right before cl, and $goal$ the current target for this state (used to enforce

```
   Input  : icfg : ICFG for program P
            goals_list : an ordered list of statements G_0, ...G_n
   Output: input_f: candidate input for synthesized run
 1 begin
 2     sym_state_0 ← initial symbolic values of program inputs
 3     states_set ← (icfg.entry, true, sym_state_0, G_0)
 4     curr_goal ← G_0
 5     while true do
 6         curr_state ← null
 7         while curr_state == null do
 8             curr_state ← SelNextState(icfg, states_set, curr_goal)
 9             if curr_state == null then
10                 if curr_goal ≠ G_0 then
11                     curr_goal ← previous goal in goals_list
12                     continue
13                 else
14                     return null
15                 end
16             end
17         end
18         if curr_state.cl == curr_goal then
19             if curr_goal == G_n then
20                 input_f ← solver.getSol(curr_state.pc)
21                 if input_f is found then
22                     return input_f
23                 else
24                     remove(curr_state, states_set)
25                     continue
26                 end
27             else
28                 curr_goal ← next target in goals_list
29                 curr_state.goal ← curr_goal
30             end
31         else
32             if curr_state.cl ∈ goal_list then
33                 remove(curr_state, states_set)
34                 continue
35             end
36         end
37         if curr_state.cl is a conditional statement then
38             curr_state.pc ← addConstr(curr_state.pc, pred, true)
39             curr_state.cl ← getSucc(curr_state.cl, true)
40             if solver.checkSat(curr_state.pc) == false then
41                 remove(curr_state, states_set)
42             end
43             false_pc ← addConstr(curr_state.pc, pred, false)
44             false_cl ← getSucc(curr_state.cl, false)
45             if solver.checkSat(curr_state.pc) ≠ false then
46                 new_state ← (false_cl, false_pc, curr_state.ss, curr_state.goal)
47                 insert(new_state, state_set)
48             end
49         else
50             curr_state.ss ← symEval(curr_state.ss, curr_state.cl)
51             curr_state.cl ← getSucc(curr_state.cl)
52         end
53     end
54 end
```

Algorithm 1: GenerateInputs

the order in which goals are reached). The initial search state consists of the entry of the program for *cl*, *PC* `true`, symbolic state sym_state_0, and goal G_0. Next, the algorithm assigns to *curr_goal* the first goal from *goals_list*.

The algorithm then enters its main loop. At the beginning of each loop iteration, GenerateInputs invokes algorithm SelNextState, shown in Algorithm 2. SelNextState looks for the most promising state to explore in *states_set*. (At the first invocation of SelNextState, only the initial state is in the *states_set*. The number of states will increase in subsequent invocations, when more of the program has been explored symbolically.) SelNextState selects

states based on the minimum distance *mindis*, computed in terms of number of statements in the ICFG, between each state's *cl* and *curr_goal*. To avoid selecting states that have not reached goals that precede *curr_goal* in *goals_list*, SelNextState only considers states whose target is *curr_goal* (line 5 in Algorithm 2). If none of these states can reach *curr_goal* (*i.e.,* there's no path between the state's *cl* and *curr_goal* in the ICFG), SelNextState returns `null` to GenerateInputs. Otherwise, the selected state is returned (line 15 in Algorithm 2).

When GenerateInputs receives the candidate state from SelNextState, it first checks whether the returned state is `null`, which means that no state in *states_set* with target *curr_goal* can actually reach such target. If so, GenerateInputs backtracks by updating *curr_goal* to the previous goal in the *goals_list* and looking for another path that can reach that goal (line 11). Conversely, if *curr_state* is not null, GenerateInputs continues the execution of its main loop.

If *curr_state*'s code location corresponds to *curr_goal*, GenerateInputs updates both global goal *curr_goal* and local goal *curr_state.goal* to the next goal in *goals_list* (lines 28–29). It then continues the symbolic execution. If the last goal G_n is reached, the algorithm stops the symbolic execution, feeds the current PC to the SMT solver, and asks the solver to find a solution for the PC (line 20). If a solution is not found, the generation of the candidate input is deemed unsuccessful. If *curr_state*'s code location is not *curr_goal* but another goal in *goal_list*, the algorithm removes *curr_state* from *state_set* and goes back to the beginning of the main loop (lines 32–34). It does so to avoid that the execution reaches the goals in the goal list in a different order from the one observed in the failing execution. If *curr_state*'s code location is a conditional statement *pred* that involves symbolic values, the algorithm performs one execution step along both branches, that is, it updates states' current location and path condition, checks the feasibility of both branches using the SMT solver, and removes (or does not add) infeasible states from *states_set* (lines 38–47). (If the SMT solver did not provide an answer for PC, the algorithm would consider the corresponding state feasible and continue.) Finally, if *curr_state*'s code location is any statement other than a conditional, the algorithm suitably updates the symbolic state and the current location of *states_set* (lines 50–51).

Input : $icfg$: ICFG for program P
$states_set$: set of symbolic states
$curr_goal$: the current goal
Output: ret_state: candidate state for exploration

```
1  begin
2      mindis ← +∞
3      ret_state ← null
4      foreach State_i ∈ states_set do
5          if State_i.goal == curr_goal then
6              if State_i.loc can reach curr_goal in ICFG then
7                  nd ← shortest distance from State_i.loc to curr_goal in ICFG
8                  if nd < mindis then
9                      mindis ← nd
10                     ret_state ← State_i
11                 end
12             end
13         end
14     end
15     return ret_state
16 end
```

Algorithm 2: SelNextState

The algorithm terminates when either there are no more states to explore (*i.e.,* it tries to backtrace from G_0 (line 14)) or a candidate input is successfully generated (line 22). In the former case, the algorithm fails to find an input that can mimic the observed execution. In the latter case, conversely, the algorithm successfully produces such input.

In summary, my guided symbolic execution technique has two key aspects. First, it uses the execution data from the field to identify a set of intermediate goals that can guide the exploration of the solution space. Second, it uses a heuristic based on distance to select which states to consider first when trying to reach an intermediate goal during the exploration. In theory, the more data (*i.e.,* number of intermediate goals) available, the more directed the search, and the higher the likelihood of synthesizing a suitable execution. On the other hand, collecting too much data can have negative consequences in terms of overhead and introduce privacy issues. To study this tradeoff, I define several variants of my field failure reproduction approach that differ on the kind of execution data they consider. The next section describes these variants.

4.1.3 Execution Data

In selecting the execution data to consider, I aimed to cover a broad spectrum of possibilities. To this end, I selected four kinds of data characterizing a failure: point of failure (POF), call stack, call sequence, and complete program trace. These types of data are representatives of scenarios that go from knowing as little as possible about the failing execution to knowing almost everything about it. In addition, POFs and call stacks are types of data that are very commonly available for crashes, as they are normally included in crash reports. Call sequences, and program traces, on the other hand, are not normally available and represent data that, if they were shown to be useful, would require changes in the way programs are monitored and crash reports are generated.

For each of these four types of execution data, I instantiated a variation of BugRedux that collected and used that type of data. As far as data collection is concerned, the first two types of execution data do not require any modification of the program being monitored, as they can be extracted them from existing reports. The other two types of data can be collected using well-understood program instrumentation techniques. To collect call sequences, BugRedux instruments all call sites and entry points (these latter to account for the possible presence of function pointers), whereas to collect program traces it instrument all branches.

Customizing BugRedux so that it uses the different data is also relatively straightforward, as it amounts to suitably generating the *goals_list* set to be passed to BugRedux's input generator. For POF, *goals_list* would contain a single entry—the POF itself. For a failure's call stack, there would be an entry in the set for each function on the stack, corresponding to the first statement of the function, plus an additional entry for the POF. Call sequences would result in a *goals_list* that contains an entry for each call, corresponding to the call statement. Also in this case, there would be an additional, final entry for the POF. Finally, the *goals_list* for a program trace would consist of an entry per branch, corresponding to the statements that is the destination of the branch, and the usual entry for the POF.

In the next section, I will discuss how I used these four variants of BugRedux to study

the tradeoffs involved with the use of different kinds of information and assess the general usefulness of BugRedux.

4.2 *Empirical Investigation*

I investigated the following two research questions for BugRedux:

- **RQ1:** Can BugRedux synthesize executions that are able to reproduce field failures?
- **RQ2:** If so, which types of execution data provide the best tradeoffs in terms of cost benefit?

To address these questions, I implemented the four variants of BugRedux discussed in the previous section and applied them to a set of real-world programs.

4.2.1 BugRedux Implementation

My implementation of BugRedux works on C programs and consists of three modules that correspond to the three components shown in my high-level view of BugRedux (see Figures 3 and 4): instrumenter, input generator, and oracle. BugRedux's instrumenter performs static instrumentation (*i.e.,* probes are added to the code at compile time) by leveraging the LLVM compiler infrastructure (`http://llvm.org/`). The input generator in BugRedux is built on top of KLEE [21], a symbolic execution engine for C programs. I implemented Algorithms 1 and 2 as a custom search strategy for KLEE and also made a few modifications to KLEE's code. Finally, BugRedux's oracle module is implemented as a Perl script that operates as follows: (1) it takes as input program P, an input I for P, and a crash report C corresponding to failure F; (2) it runs P against I and collects any crash report generated as a result of the execution; and (3) if either no report is generated or the call stack at the moment of the crash and POF in the generated report do not match those in C, it reports that the approach failed, whereas it reports a success otherwise.

4.2.2 Programs and Faults Considered

To investigate my research questions in a realistic setting, I used a set of real, non-trivial programs that contained one or more faults and had test cases that could reveal such faults.

Table 1: Benchmark programs used in my study

Name	Repository	Description	Size (kLOC)	# Faults
sed	SIR	stream editor	14	2
grep	SIR	pattern-matching utility	10	1
gzip	SIR	compression utility	5	2
ncompress	BugBench	(de)compression utility	2	1
polymorph	BugBench	file system "unixier"	1	1
aeon	exploit-db	mail relay agent	3	1
glftpd	exploit-db	FTP server	6	1
htget	exploit-db	file grabber	3	1
socat	exploit-db	multipurpose relay	35	1
tipxd	exploit-db	IPX tunneling daemon	7	1
aspell	exploit-db	spell checker	0.5	1
exim	exploit-db	message transfer agent	241	1
rsync	exploit-db	file synchronizer	67	1
xmail	exploit-db	email server	1	1

I considered programs from three public repositories that have been used extensively in previous research: SIR [2], BugBench [61], and exploit-db [18]. Specifically, I selected three programs from SIR, two from BugBench, and nine from exploit-db. Table 1 shows the relevant information about each program: name, repository from which it was downloaded, size, and number of faults it contains. As the table shows, the program sizes range from 0.5 kLOC to 241 kLOC, and each program contains one or two faults. The faults in the BugBench and exploit-db programs are real field , whereas the ones in the programs from SIR are seeded.

I selected these programs because they have been used in previous research [18,30] and because of the representativeness of their faults. The faults in exploit-db and BugBench are *real field failures mostly discovered by users in the field*, whereas the faults in SIR are seeded by researchers but are carefully designed to simulate real faults.

I excluded from my study three programs from SIR and four from BugBench because the version of KLEE I used could not handle some of the constructs in these programs (*e.g.,* complex interactions with the environment and network inputs). As far as faults are concerned, I selected faults that caused a program crash, rather than just generating an

incorrect result. This choice was made for convenience and to minimize experimental bias—with crashes, failures can be objectively identified and do not require the manual encoding of the failure condition as an assertion.

I also performed a preliminary check on the programs and faults that I selected by feeding them to an unmodified version of KLEE and letting it run for 72 hours. The goal of this check was to assess whether these faults could have been discovered by a technique that blindly tries to explore as much of the programs as possible. If so, this would be an indication that the faults are too easy to reveal to be good candidates for my study. The unmodified KLEE was unable to reveal the faults in the programs except for one case: iwconfig. I therefore removed iwconfig from my set of benchmark. It is worth noting that I decided not to use the benchmarks used in Reference [82] for the same reason—all of those failures could be recreated through plain, unguided symbolic execution, as also shown in Reference [21]. (Moreover, the benchmarks I selected are more representative, as programs are larger and 9 out of 16 faults are real faults reported by users, rather than faults found in-house by KLEE.)

4.2.3 Experimental Setup

In order to collect the data needed for my investigation, I proceeded as follows. To simulate the occurrence of field failures, I used the test cases distributed with my benchmark programs as proxies for real users. For each fault f considered, I ran the test cases until a test case t_f failed and generated a program crash; I associated t_f to f as its failing input. I then reran all the failing inputs on all the corresponding faulty programs three times. The first time, I ran them on the unmodified programs, the second time on the programs instrumented by BugRedux to collect call sequences, and the third time on the programs instrumented by BugRedux to collect complete program traces. For each such execution, I measured the duration of the execution and the size of the execution data generated.

With this information available, I used the four variants of BugRedux to synthesize a failing execution starting from a suitable set of goals (*i.e.,* POF, call stack at the time of failure, call sequence, and complete program trace). For each run of BugRedux, I recorded

whether the generation was successful (*i.e.,* whether a candidate input was generated at all) and how long it took. I set a timeout of 72 hours for the generation, after which I marked the run as unsuccessful. I also recorded whether the candidate input, if one was generated, could reproduce the original failure according to BugRedux's oracle.

4.2.4 Results

This section presents the results of my empirical study and discusses the implication of the results in terms of my two research questions. I present the results using two tables, where the first table contains the data related to the cost of the approach (*i.e.,* the time and space overhead imposed by BugRedux), and the second table shows the data about the effectiveness of the approach (*i.e.,* whether BugRedux was able to synthesize an execution and whether such execution could be used to reproduce an observed failure). These two tables present the results for each of the 16 failing executions considered, identified by the name of the failing program possibly followed by a fault ID, and for each of the variants of BugRedux, identified by the kind of execution data on which it operates.

Table 2 shows the time and space overhead imposed by BugRedux on the benchmark programs for each of the four types of execution data collected. Time overhead is measured as the percentage increase of the running time due to instrumentation, whereas space overhead is measured as the size of the different kinds of execution data collected by BugRedux. I discuss the two types of overheads separately.

Time overhead. Because POFs and call stacks are collected by the runtime system at the moment of the failure, and do not require any additional instrumentation, collecting them incurs no overhead. The situation is different for call sequences and complete traces, which both require BugRedux to instrument the programs (see Section 4.1.3). As expected, the overhead imposed by complete-trace collection is almost an order of magnitude higher than that for call sequences. I also observe that the overhead for collecting call sequences depends on program size and execution length. To correctly interpret these results, it is important to consider that this data was collected with a naive instrumentation that writes events to the log as soon as they occur; the use of caching techniques could decrease the

Table 2: Time (%) and space (KB) overhead imposed by BugRedux

Name	POF		Call stack		Call sequence		Complete trace	
	time	space	time	space	time	space	time	space
sed.fault1	0%	0.8	0%	0.8	4.5%	5.8	27.2%	54.4
sed.fault2	0%	0.9	0%	0.9	12.5%	10.2	87.5%	261.9
grep	0%	0.7	0%	0.7	47%	3.4	182%	716.1
gzip.fault1	0%	0.8	0%	0.8	10.3%	2	72%	176
gzip.fault2	0%	0.8	0%	0.8	12%	2.5	308%	1784.6
ncompress	0%	0.7	0%	0.7	2%	0.9	16%	33.1
polymorph	0%	0.5	0%	0.5	1%	0.7	8%	1.5
aeon	0%	1	0%	1	50%	1.1	1066%	3
glftpd	0%	1.5	0%	1.5	9%	3.2	45%	130
htget	0%	0.7	0%	0.7	9%	2.7	287%	2814
socat	0%	0.8	0%	0.8	21%	9.6	110%	451
tipxd	0%	0.6	0%	0.6	2%	0.7	36%	19
aspell	0%	0.6	0%	0.6	18.8%	30.5	143%	566
rsync	0%	1	0%	1	3%	11.4	66%	521
xmail	0%	0.8	0%	0.8	22.6%	84.8	290%	2361
exim	0%	0.9	0%	0.9	17.4%	100.7	389%	14897

overhead dramatically. Because the goal of this initial investigation was more exploratory, and the numbers are acceptable, I left optimizations for future work. Moreover, record-replay techniques (*e.g.,* [4, 28]) could be used to (1) efficiently record field executions and (2) collect execution data while replaying offline and when free cycles are available on the user machines.

Space overhead. The data size for POFs and call stacks is the same because my current implementation of BugRedux extracts both of them from the crash reports generated by the runtime system. I therefore decide to report the size of the crash reports for these two types of data. Also in this case, the size of the complete-trace data is at least an order of magnitude larger than that of the call-sequence data, and in some cases the difference is even more extreme. For instance, in the case of gzip.fault2, the reason for the large gap is that the number of function calls is low but there is a large number of loop iterations within functions. Overall, however, for the executions in this study, the size of the execution data is fairly contained, and it would be practical to collect them.

Table 3 addresses the core question of the effectiveness of my approach. The table shows,

Table 3: Effectiveness and efficiency of BugRedux in synthesizing executions starting from collected execution data

Name	POF		Call stack		Call sequence		Complete trace	
sed.fault1	N/A		N/A		98s	Y	N/A	
sed.fault2	N/A		N/A		17349s	Y	N/A	
grep	N/A		16s	N	48s	Y	N/A	
gzip.fault1	3s	Y	18s	Y	11s	Y	N/A	
gzip.fault2	20s	N	28s	N	25s	Y	N/A	
ncompress	155s	Y	158s	Y	158s	Y	N/A	
polymorph	65s	Y	66s	Y	66s	Y	N/A	
aeon	1s	Y	1s	Y	1s	Y	1s	Y
rysnc	N/A		N/A		88s	Y	N/A	
glftpd	5s	Y	5s	Y	4s	Y	N/A	
htget	53s	N	53s	N	9s	Y	N/A	
socat	N/A		N/A		876s	Y	N/A	
tipxd	27s	Y	27s	Y	5s	Y	N/A	
aspell	5s	N	5s	N	12s	Y	N/A	
xmail	N/A		N/A		154s	Y	N/A	
exim	N/A		N/A		269s	Y	5624s	Y

for each failing execution fe considered and each type of execution data ed, the time it took BugRedux to generate inputs that mimicked fe using ed (or "N/A" if BugRedux was unable to generate such inputs in the allotted time) and whether the mimicked execution reproduced the observed failure ("Y" or "N").

As expected, symbolic execution guided only by the POF was unsuccessful for most programs. A manual examination of the programs for which POFs are enough to reproduce failures showed that all such failures have two common characteristics: (1) the POFs are close to the entry of the programs and are easy to reach; (2) the failures can be triggered by simply reaching the POFs. For these failures, developers could easily identify the corresponding faults if provided with traditional crash reports. As also expected, the larger the amount of data available (in the form of intermediate goals) to guide the exploration, the better the performance of the approach. Using call stacks, BugRedux could mimic 10 out of the 16 failing executions, and using call sequences, it was able to mimic all failing executions.

I observe that, in some cases (*e.g.,* htget, tipxd), the time needed to synthesize an execution using call stacks was larger than the time needed when using call sequences (when

they are both successful). The reason for this behavior is that the additional information provided by call sequences can better guide symbolic execution and avoid the exploration of many irrelevant paths. One surprising finding, however, is that this trend is not confirmed when complete traces are used. I further analyzed this result and found that this happened for two reasons. One reason is that, intuitively, following complete traces can result in conditions that the SMT solver is unable to handle. The second reason is a mostly practical one: KLEE uses a simplified implementation of the system libraries when symbolically executing a program, which makes it impossible in some cases to follow exactly the same path that was followed in the original execution. Conversely, a looser, yet informative guidance, such as a call sequence, leaves more degrees of freedom to the input generator and increases its chances of success. For example, paths that result in constraints that are beyond the capabilities of the SMT solver could be dropped in favor of simpler paths that may still reach the targeted goals. In a sense, among the execution data considered, call sequences represent a sweet spot between providing too little and too much information to the search.

It is important to stress that the executions synthesized by BugRedux are executions that reach all of the intermediate goals extracted from the execution data and provided to the input generator, but they are not guaranteed to reproduce the observed failure. This is especially true when considering more limited types of execution data, such as POFs and call stacks, which provide little guidance to the search. The results in Table 3 clearly illustrate this issue. As shown in the table, for three of the failures in my set, reaching the POF is not enough to trigger the original failure. Similarly, for the four failures in grep, gzip, htget, and aspell, BugRedux was able to synthesize executions that generated the same call stacks as the failing executions, but such synthetic executions did not reproduce the considered failures. Conversely, all of the 16 synthetic executions successfully generated from call sequences were able to reproduce the original failures.

4.2.5 Discussion

The results of my investigation, albeit preliminary, let me address the two research questions and make some observations. For RQ1, my results show that, for the programs and failures considered, BugRedux can reproduce observed failures starting from a set of execution data. For RQ2, the results provide initial but clear evidence that call sequences represent the best choice, among the ones considered, in terms of cost-benefit tradeoffs: using call sequences, BugRedux was able to reproduce all of the observed failures; even using an unoptimized instrumentation, BugRedux was able to collect call sequences with an acceptable time and space overhead; and I believe that call sequences are unlikely to reveal sensitive or confidential information about an execution. (Although this is just anecdotical evidence, I observed that none of the inputs generated when synthesizing executions from call sequences corresponded to the original input that caused the failure.) Unlike complete traces, which may provide enough information to reverse engineer the execution and identify the inputs that caused such execution, call sequences are a much more abstract model of executions.

An additional observation that can be made on the results is that POFs and call stacks do not seem to be particularly helpful for reproducing failures. Manual examination of the faults considered showed that the points where the failure is observed tend to be distant from the fault. Therefore, most such failures are triggered only when the program executes the faulty code and the incorrect program state propagates to the POF. In these cases, POFs and call stacks are unlikely to help because the faulty code may be nowhere near the POF or the functions on the stack at the moment of the crash. If confirmed, this would be an interesting finding, as these are two types of execution data normally collected in crash reports. Extending crash reports with additional information may make them considerably more useful to developers.

As a further step towards understanding the usefulness of different execution data, I performed an additional exploratory study in which I removed entries in the collected call sequences and checked whether the partial sequences still contained enough information to recreate observed failures. More precisely, I selected from my original list the ten failures

Table 4: Minimal number of entries in call sequences that are needed to reproduce observed failures

Name	Original Length	Minimal Length
sed.fault1	73	12
sed.fault2	146	7
grep	31	2
xmail	1142	363
gzip.fault2	27	2
rysnc	23	2
aspell	516	256
socat	62	3
htget	25	2
exim	1029	326

that could only be reproduced using call sequences. For each failure and corresponding call sequence, I then used a straw-man greedy algorithm that considers one entry in the call sequence at a time, starting from the beginning. If BugRedux can reproduce the failure without that entry in the sequence, the entry is removed. Table 4 shows the result of this study in terms of number of entries in the call sequences before and after reduction. For example, only 2 of the 31 entries in the original call sequence are needed to reproduce the observed failure in grep. The results show that, in most cases, only a small subset of calls in the sequences is actually necessary to suitably guide the exploration. I can further observe that the number of entries needed seems to increase with the complexity of the input needed to trigger the fault, which makes sense intuitively. For instance, xmail's fault can only be triggered by an input file that includes a valid email address, and aspell's fault can only be triggered by an input of a given length. For these two faults, the reduction in the call sequence is less substantial than for the other faults considered. These additional results motivate my fault localization technique for field failures, as discussed in Chapter 5.

4.2.6 Limitations and Threats to Validity

The main limitation of BugRedux is that it relies on symbolic execution, an inherently complex and expensive approach. However, recent results have shown that, if suitably defined, tuned, and engineered, symbolic execution can scale even to large systems [43]. Moreover, as discussed in future work, BugRedux can leverage different input generation techniques. Another limitation is the potential overhead involved in collecting field-execution data. For this reason, as also discussed in future work, I am currently investigating the use of alternative execution data. One final limitation is that BugRedux currently does not explicitly handle concurrency and non-determinism. In this initial phase of the research, I chose to focus on a smaller domain, and get a better understanding of that domain, before considering additional issues.

Like for all studies, there are threats to the validity of the results. To mitigate threats of internal validity related to errors in my implementation, I tested BugRedux on small examples and spot checked most of the results presented in this chapter. In terms of external validity, the results may not generalize to other programs and failures. However, I studied 16 failures and 14 programs from three different software repositories. The benchmark programs I used are real-world programs, several of which are widely used both by real users in the field and by researchers as experimental programs. Another issue with the empirical results is that the ultimate evidence of the usefulness of the technique would require its use in a real setting and with real users. Although such an evaluation would be extremely useful, and I plan to do it in the future as other researchers did for their work [68], I believe that it would be premature at this point. Moreover, when successful, BugRedux would generate an actual execution that reproduces the observed field failure. I expect such an execution to be usable, like any other failing execution, to debug the problem causing the failure.

Overall, I believe that my initial results show that the approach is promising and identify several research directions that it would be worth pursuing; directions that could be investigated by building on the failure failure reproduction work.

4.3 Conclusion

The ability to reproduce an observed failure has been reported as one of the key elements of debugging. Whereas recreating failures that occur during in-house testing is usually easy, doing so for failures that occur in the field, on user machines, is unfortunately an arduous task. To address this problem, I have presented BugRedux, a general technique for supporting in-house debugging of field failures. BugRedux works by (1) collecting execution data about failing program executions in the field as a sequence of intermediate goals (*i.e.,* locations in the program) and (2) using input generation techniques to synthesize, in house, executions that reach such goals and mimic the observed failures.

To better understand the tradeoffs between amount of information collected and effectiveness of the approach, I have performed an empirical investigation and studied the performance of four instances of BugRedux that leverage different kinds of execution data. I have applied these four instances to a set of 16 failures for 14 real-world programs and compared their cost and effectiveness. The results of empirical studies are encouraging and provide evidence that BugRedux can reproduce observed failures starting from a suitable set of execution data. In addition, the analysis of the results led to several findings, some of which unexpected (*e.g.,* more information is not always better). Finally, my results provide insight that can guide future work in this area. In fact, I observed that the synthesized executions by BugRedux would be an ideal input for fault localization techniques, which leads to my work for fault localization for field failures in Chapter 5.

CHAPTER V

AUTOMATED FAULT LOCALIZATION FOR FIELD FAILURES

5.1 F^3 *Technique*

As discussed in Chapter 1, one of my overarching goals is to help developers locate the likely cause of observed field failures by applying fault localization techniques. Since BugRedux can only reproduce observed field failures and it does not provide any explicit debugging support, I need to extend it to debug such field failures. In this chapter, I present F^3 (Fault localization for Field Failures) to achieve this goal and accomplish the third step of my overall vision presented in Chapter 3. I devised F^3 by extending BugRedux such that it generates more than one failing executions and a set of passing executions that are "similar" to these failing executions. In addition to extending BugRedux, I also apply customized fault localization with three optimizations on these generated executions. Figure 5 provides a high-level overview of F^3.

As the figure shows, given (1) a set of minimized crash data produced by BugRedux for a field failure f (*i.e.*, a list of intermediate goals leading to f) and (2) the failing application, F^3 produces a report of *likely faults*: a list of program entities ordered in terms of likelihood of being faulty and being responsible for f. The report can then help developers investigate and understand the causes of field failure f.

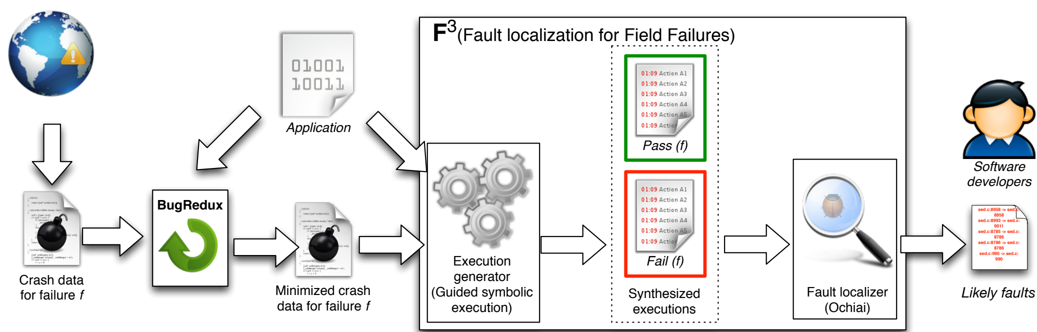

Figure 5: This is the high-level intuitive overview of F^3.

F^3 consists of two main parts: the *Execution generator* and the *Fault localizer*. The execution generator component extends BugRedux so that, given a set of crash data for an execution e that fails with failure f, it synthesizes two sets of executions: *FAIL*, a set of failing executions that "mimic" e and generate a failure analogous to f (*i.e.,* the failing executions that are generated by BugRedux as discussed in Chapter 4), and *PASS*, a set of passing executions that are "as similar as possible" to e but do not fail (see Section 5.1.1 for more details on this concept of similarity among executions). The fault localizer component performs my optimized statistical fault localization techniques on these two sets of executions to identify a set of program entities that are likely to be responsible for failure f. In the rest of this section, I will discuss in detail these two components.

5.1.1 Execution Generator

One major issue with traditional statistical fault localization is that the ranking results highly depend on the quality of the test cases (or, more generally, executions) available for the statistical analysis. In general, statistical fault-localization approaches require a large number of passing and failing executions to be effective. Moreover, the quality of the executions is also important. If, for instance, the passing executions exercise completely different parts of the code than the failing executions, they are of limited usefulness for the analysis. Unfortunately, as mentioned in Chapter 1, it is rarely the case that such a high-quality set of executions is available in practice, and identifying inputs that can generate suitable executions is a non-trivial task [70]. In the case of field failures, in particular, the state-of-the-art failure reproduction techniques (*e.g.,* [51, 82]) generate at most one failing execution, preventing any kind of statistical analysis.

My intuition is that this problem can actually be seen as a missed opportunity. Existing execution synthesis approaches can be extended to generate a suitable set of both passing and failing executions to be used in fault localization. Furthermore, because of the way they are generated, these executions may be even more amenable to statistical fault localization than existing test suites.

As explained above, the execution generator takes as input an ordered list of program

locations l. In this context, l is a minimized list of goals that contains enough entries to guide the generator and to allow it to synthesize an execution that mimics the observed field execution e and reproduces the observed field failure f [51]. In the original BugRedux approach, l was used to generate a single failing execution. In F^3, I extend BugRedux so that it continues to synthesize executions that mimic the one observed in the field until it reaches a given time limit.

By construction, each execution that is successfully synthesized in this way is guaranteed to reach all program locations in l and reach them in the same order of the original field execution. Therefore, all these executions reach the point of failure, share with e a set of intermediate execution points, but are likely to follow different paths than e. It is worth noting that reaching the point of failure does not guarantee that the program would fail at that point, as that also depends on the state of the execution. Therefore, some of the synthesized executions may not result in failure f. Although this is true in general, in practice it is often the case that l provides enough constraints to the execution synthesis algorithm that all generated executions would indeed trigger the failure, and no passing execution would be generated.

In this case, my technique increasingly eliminates entries from l and tries to synthesize executions using this reduced list l' as a guide instead of l. (In my current instance of the approach, I use a straw-man approach that simply eliminates one entry at a time starting from the beginning of the list until the execution generator can generate passing executions.) The rationale for this approach is that eliminating entries in the list augments the degree of freedom of the synthesis algorithm, which in turn increases the chances of generating passing executions. Generating executions using a smaller list, however, is bound to reduce the degree of similarity between the synthesized passing executions and the original execution e. Therefore, the generated passing executions will be increasingly dissimilar from e, and thus less likely to be useful for fault localization, as more entries are eliminated from the list. In extreme cases, in order to generate passing executions, I have to remove all entries from l. Fortunately, as the results of my empirical evaluation in Section 6.3 show, this approach seems to work well in practice: (1) the execution synthesizer did not have to eliminate many

entries to generate passing executions in many cases, and the degree of similarity between the synthesized executions and the original execution was not considerably affected; (2) cases that I had to remove all entries to generate passing executions did happen but only in a minority of cases, and even in these cases, the results of fault localization were still good enough.

This process terminates when a time limit is reached, one or more passing executions have been successfully synthesized for a given sublist, or no more entries can be eliminated from l' (*i.e.,* the list is empty). When successful, the execution generator would therefore produce the two sets of executions *FAIL* and *PASS* mentioned above, which contain failing and passing executions. These executions would be similar, in that they share a set of intermediate execution points, because all of them are derived from l or a subset thereof. Intuitively, using such similar passing and failing runs can help fault localization. In fact, recent research has shown that generating passing executions that are close to failing executions can considerably improve the results of fault localization techniques [70, 75].

5.1.2 Fault Localizer

In this section, I discuss how F^3 uses the sets *FAIL* and *PASS*, synthesized by the execution generator and discussed in the previous section, to perform fault localization and identify the code that may be responsible for the field failure *f*.

There are many ways to perform fault localization given two sets of failing and passing executions, as a variety of statistical fault localization techniques, and variations thereof, can be used to this end. As discussed in Chapter 2, I decided to choose Ochiai, OBM, Naish1, and Naish2 as representative techniques and suitably optimized them. Specifically, based on previous experience and initial results, I customized these techniques along three directions: use of profiling information, aggressive filtering, and grouping of related entities. In the next sections, I describe my optimizations and discuss how my fault localizer uses the customized fault-localization techniques.

5.1.2.1 *Traditional Fault Localization*

One straw-man way to use sets *FAIL* and *PASS* for debugging is to simply apply traditional statistical fault localization techniques to these sets. Both fault-localization techniques I am considering only need two such sets to operate and can be used without any additional modification. However, I defined F^3 based on the intuition, later confirmed by the empirical results presented in Section 6.3, that I can improve the effectiveness of traditional fault localization by suitably tailoring these techniques. To do so, I defined the two main optimizations and one engineering improvement of these techniques discussed in the rest of this section.

5.1.2.2 *Fault Localization with Filtering*

One problem with traditional fault-localization approaches is that most of them rely on a potentially misleading metric for defining and evaluating the approach: they measure effectiveness based not on the number of program entities developers must inspect before identifying the fault, but rather on the percentage of the program they must inspect. Although having to inspect only 5% of the program may appear as a fairly positive result at first glance, considering that this may correspond to hundreds or thousands of program entities gives a quite different, and more practical, perspective. In fact, a human study performed in previous work provides clear evidence that fault localization techniques should focus on improving *absolute rank* rather than percentage rank [68], as developers are likely to stop inspecting the list of suspicious program entities if they could not identify relevant entities among the first few entries. (Some researchers have proposed to address this issue by cutting the ranked list at a given size or at a given suspiciousness threshold, but that introduces the issue of finding the right size or threshold [38, 70].)

This issue is made even worse by the fact that most approaches tend to assume *perfect bug understanding*, that is, they assume that simply examining a faulty statement in isolation without any additional context is always enough for a developer to detect, understand, and correct the corresponding bug. The aforementioned human study [68] also shows that

this is not a realistic assumption, which means that the number of entities developers actually examine is normally larger than the number of entities they have to examine in the ranked list.

Because F^3 can generate many passing and failing executions that are similar and tend to focus on a relatively small part of the program, I believe that I can aggressively filter the information computed using these executions in a way that would not be as effective if used on passing and failing executions that are not specifically created to be similar.

Such an aggressive filtering of the list of suspicious program entities has multiple advantages. First and foremost, it can improve the absolute rank of the faulty program entities, which can greatly benefit fault localization. Second, it can also reduce the total length of the ranked list of program entities to examine in a principled way, that is, not based on the choice of an arbitrary size or suspiciousness threshold [38, 70]. Finally, filtering can reduce the number of entities to be considered in the statistical analysis, which can make the analysis more efficient. (This latter advantage would be generally marginal, however, unless a particularly expensive statistical analysis is used.)

F^3 performs filtering using dynamic information about the program entities (branches, in my case) executed in both passing and failing runs, with the goal of discarding beforehand parts of the program that are likely to be irrelevant for the failure. Specifically, I defined three different types of filters, that I use to exclude from the statistical analysis some of the branches in the program. That is, if I define the set of branches exercised by an execution e_i as

$$BR(e_i) = \{br_{i1}, br_{i2}, ..., br_{im}\},$$

using filter FIL_x corresponds to considering only the following branches for the statistical analysis:

$$BranchesToAnalyze = FIL_x \cap \bigcap_{e_i \in FAIL \cup PASS} BR(e_i)$$

The *first type* of filter I defined considers only branches that are exercised in *all* failing

executions and exclude all other branches from the statistical analysis:

$$FIL_f = \bigcap_{e_i \in FAIL} BR(e_i)$$

Clearly, FIL_f may contain branches that are executed by passing executions too, and in some cases by all passing executions, such as initialization code. Therefore, to further filter the fault localization results, F^3 considers a *second type* of filter, FIL_{fp}, which is more aggressive because it further excludes from the analysis those branches that are exercised in *all* passing executions:

$$FIL_{fp} = FIL_f - \bigcap_{e_i \in PASS} BR(e_i)$$

The rationale for FIL_{fp} is that by removing common branches in both passing and failing executions, I remove the branches that may be visited by all executions of the program and are thus likely (but obviously not guaranteed) to be irrelevant.

The *third type* of filtering in F^3 further reduces the amount of entities considered in the statistical analysis by also ignoring branches on which other branches in FIL_{fp} are control dependent:

$$FIL_{dep} = FIL_{fp} - \{br_k \mid br_k \in FIL_{fp} \land \exists br_j \in FIL_{fp}, j \neq k, br_j \text{ is c.d. on } br_k\}$$

The rationale, in this case, is that branches that control other branches are often (but clearly not always) responsible for reaching those branches and not directly responsible for the failure.

The three filters I defined are obviously increasingly restrictive (*i.e.,* $FIL_{dep} \subseteq FIL_{fp} \subseteq FIL_f$) and can thus be used to perform an increasingly aggressive filtering. Note that, although there is overlapping between the information used to filter and the information computed by statistical fault localization, filtering still provides an independent way of improving the results of fault localization because it can reduce the overall number of suspicious program entities considered, as showed in my empirical study of F^3. Moreover, I expect these filters to be particularly effective when operating on similar passing and

```
   ...
1. int a[10];
2. int i=0;
3. do {
4.     a[i] = getchar();
5.     i++;
6. } while (a[i-1]!='\n');
   ...
```

Figure 6: A code snippet containing a buffer-overflow bug

failing executions, such as the ones generated by F^3. First, F^3 can generate several failing executions for a given fault, rather than just one, which I found to be typically the case in existing test suites. This can help narrowing down the possible location of the fault. Second, F^3 can also generate several passing executions that share commonalities with the failing ones, unlike in typical test suites, where passing and failing executions tend to be quite different from one another. This can help filtering out irrelevant entities.

5.1.2.3 Fault Localization with Profiling

Most fault-localization approaches are based on the concept of coverage of a given program entity, where the entity is typically a statement, a branch, or a more generic predicate. If considering the formula used to compute the suspiciousness of an entity *en* in Ochiai, for instance, which showed in Section 2.2.1, the value a_{11} would be increased by one for each failing execution that exercises *en*. In my initial experience with F^3, however, I have observed that coverage is not necessarily always the best kind of dynamic information for fault localization, especially for common faults (*e.g.,* memory related bugs) in field failures. Consider, for instance, the code snippet in Figure 6, which is a simplified version of a real bug I encountered in one of the programs I studied.

In this example, the program reads characters from the input stream until a newline character is encountered. The code contains a buffer-overflow bug, which is triggered when the size of the input stream is greater than 10. Both passing and failing executions could cover line 4, with the difference that failing executions would execute that line more than 10 times. From the standpoint of a technique based on coverage, however, statement 4 appears in every passing and failing run and is therefore not more suspicious than any

other statement executed in all runs.

In general, I observed that many failures, especially those common field failures that are related to memory errors, depend on the number of times one or more program entities are executed. Therefore, in my approach, I also consider an extension of Ochiai that uses profiling, rather than coverage information. (I do not apply this extension to the other three techniques because it is not possible to consider profiling information in these techniques without redefining them extensively. To embed profiling information in OBM, for instance, we would have to modify the way in which OBM computes conditional probabilities for each diagnosis.)

To extend Ochiai with profiling information, I replace the original suspiciousness formula for an entity en with the following one:

$$suspiciousness(en) = \frac{p_{11}}{\sqrt{(p_{11} + p_{01}) \times (p_{11} + p_{10})}}$$

In this new formula, p_{11} indicates the number of times all failing executions exercised en, instead of the number of failing executions that exercised en, as in the original formula. Similarly, p_{10} indicates the number of times all passing executions exercised en, and p_{01} indicates the number of failing executions that did not exercise en. Using this new formula, a program entity en with higher p_{11} and lower p_{10} would be consider more likely to be faulty.

Besides modifying the formula for computing suspiciousness values, in my definition of Ochiai I also introduce an additional optimization that aims at breaking ties for entities with the same suspiciousness value: when two entities have the same suspiciousness value, I order these entities based on the value of p_{11}. This optimization, which could be used also with the original Ochiai approach, accounts for situations in which the effect of p_{11} is masked (*e.g.,* when both p_{01} and p_{10} are zero), which occurred several times in my experiments.

Let us consider again the buggy code snippet in the example of Figure 6. If I apply my customized version of Ochiai to that code, the branch that leads to statements 4 and 5 would be (correctly) ranked higher than other entities, unlike what happens when using coverage information.

5.1.2.4 Fault Localization with Grouping

In addition to the two main optimizations discussed above, I also implemented an engineering optimization to improve the results of fault localization. Similar to many other techniques, my current instantiations of statistical fault-localization techniques instrument low-level code to collect the runtime information used for the statistical analyses. As a result, statements that appear in a single source code location (*i.e.,* a single line in the source code) are commonly split into multiple low-level instructions (and even multiple basic blocks). Because developers would inspect a program at the source code level, and thus consider statements from the same source code location together during debugging, low-level entities should be suitably grouped when reporting the fault localization results to developers. To do so, I leverage the source code information in the low-level program entities and use a simple heuristic: if program entities that have the same suspiciousness value correspond either to the same line of source code or to consecutive lines, my approach groups them together and reports them to developer as a single entry.

This optimization is different from the previous two optimizations, as grouping mainly helps developer understand and consume the ranked list produced through statistical fault localization. In other words, this optimization addresses an issue that is mainly related to the *engineering*, rather than the *definition*, of the approach. I discuss it here nevertheless because I have observed that it can make a considerable difference in practice and believe it may apply to other related fault-localization techniques. Without grouping, (low-level) related program entities may be spread among other entities with the same suspiciousness value, and the developers would likely waste time going back and forth in the list before they understand the relationships among these entities.

5.2 Empirical Evaluation

To assess the practical usefulness of F^3, I implemented it in a prototype tool and applied the tool to a set of 11 real-world programs and corresponding failures. More precisely, I investigated the following research questions:

- **RQ1:** Can F^3 synthesize multiple passing and failing executions for a given set of crash

data?

- **RQ2:** What is the degree of similarity of these synthesized passing and failing executions?
- **RQ3:** Can F^3 use these synthesized executions to perform fault localization effectively, and how does the degree of similarity affect the effectiveness of fault localization?
- **RQ4:** Do my optimizations actually improve the effectiveness of fault localization and, if so, to what extent?

In the rest of this section I discuss the implementation of F^3, my experimental protocol, and the results of my evaluation. For ease of presentation, hereafter, I use the name F^3 to refer to the implementation of my approach.

5.2.1 Implementation

F^3 works on C programs and is built on top of the BugRedux tool that I presented in Chapter 4, whose functionality it leverages and extends. The first component of F^3, the execution generator, leverages the symbolic execution engine KLEE [21], customized to (1) use crash data as a guide for its search and (2) generate both passing and failing runs. The fault localizer leverages the LLVM compiler infrastructure (`http://llvm.org/`) to add to the code probes that record various coverage and profiling information. LLVM is also used to compute the static control dependence information necessary to calculate set FIL_{dep}, which I discussed in Section 5.1.2.2. Finally, I implemented the fault localization approaches considered and their optimizations as Perl scripts that analyze the collected execution traces. F^3 performs fault localization at the basic-block (rather than statement) level and identifies a basic block by means of the branch leading to it. I accordingly report fault-localization results in terms of branches, and thus basic-blocks identified by those branches. I use branches for simplicity, as I also use them for filtering. Note that branch information subsumes block—and thus statement—information, and in fact branches are actually directly used in some fault-localization techniques (*e.g.,* [57, 58]).

5.2.2 Benchmark of Study

For this study, I selected programs from the benchmark programs I used in the empirical evaluation of BugRedux (see Section 4.2) and added a bug from GNU's findutils [41]. These are *real-world*, non-trivial programs that contain known faults and are available, together with a test suite, from three public repositories: SIR [2], exploit-db [18], and findutils [41]. The three faults in the programs from SIR were seeded by researchers and were designed to be representative of different types of real faults. The seven faults from exploit-db and the fault in findutils, conversely, are *real faults* reported by users in the field (*i.e.,* exactly the kinds of *real field failures* that my technique targets).

Note that, for convenience, to avoid bias, and similar to what I did in BugRedux, I only selected programs with crashing bugs. Although there are ways to identify non-crashing failures, such as anomaly-detection techniques or assertions, crashes are commonly targeted failures because they can be effectively treated as built-in, completely objective oracles. In order to focus on interesting faults, I also excluded all programs whose faults could either be easily revealed through exhaustive exploration of the program space (*i.e.,* without any guidance) or easily found because in close proximity to the point of the crash (*i.e.,* a simple analysis of the crash stack would allow for localizing these faults). In summary, I believe that all of the faults, and corresponding failures, that I selected are sophisticated enough that (1) state-of-the-art in-house testing techniques would miss them, and (2) developers would have a difficult time investigating them with only limited information from the field.

Table 5 shows the relevant information about the programs and faults I considered. For each program, the table shows its name, repository of provenance, size, number of faults, and the average number of source code lines in a basic block. (We report the average size of basic blocks to help readers better understand and interpret our results, as we use branches as the suspicious entities for fault localization.) For the faults in the exploit-db programs, I identified the location of the faulty entities using documentation and bug fixes available for these programs. For the seeded faults in the SIR programs, I used the positions of the seeded faults as fault locations.

Table 5: Programs from SIR and exploit-db used in my study

Name	Repository	Description	Size (kLOC)	# Faults	Average BB Size (LOC)
gzip	SIR	compression utility	5	1	1.5
grep	SIR	pattern-matching utility	10	1	1.3
sed	SIR	stream editor	14	2	1.5
aspell	exploit-db	spell checker	0.5	1	1.4
xmail	exploit-db	email server	1	1	1.5
htget	exploit-db	file grabber	3	1	1.6
socat	exploit-db	multipurpose relay	35	1	1.7
rsync	exploit-db	file synchronizer	67	1	1.5
exim	exploit-db	message transfer agent	241	1	1.4
find	findutils	file system search utility	8	1	1.9

5.2.3 Experiment Protocol

To collect my experiment data, for each fault considered I proceeded as follows. First, to simulate the field failure for a program from SIR, I executed the test cases distributed with the program until I found a test case t able to cause a program crash. For each of the programs from exploit-db and findutils' bug tracker, conversely, I directly executed the failing test case t provided for each failure (uploaded to the repository by the real users who reported the failure). I used t as a proxy for the field failure to be investigated, and executed an instrumented version of the program against t to collect the crash data CS_{orig}. I then provided CS_{orig} to BugRedux, which generated the corresponding minimized crash data CS_{min}. (Note that this was done for convenience, as an optimized implementation of my approach would integrate the minimization and execution-generation phases.)

Next, I used F^3's execution generator to try to generate the sets of failing and passing executions *FAIL* and *PASS*, using a time threshold of five hours. This step could terminate with one of two possible outcomes: the execution generator was either (1) able to create two non-empty sets *FAIL* and *PASS* or (2) unable to synthesize any passing execution using CS_{min}. In this latter case it would try again after eliminating entries from CS_{min}, as discussed in Section 5.1.1 (in the worst case, the execution generator would synthesize passing executions using an empty set of crash data, that is, with no guidance that could make the execution similar to those generated using CS_{min}).

I then collected complete program traces for each execution in the *FAIL* and *PASS* sets

Table 6: Number of failing and passing executions generated by F^3. For the programs without a star, passing executions are generated by CS_{min}; for the programs with a single star, passing executions are generated using a subset of CS_{min}; and for programs with two stars, passing executions are generated using an empty list, that is, without guidance.

Fault	CS_{min}	Number of failing executions	Number of passing executions
exim	326	598	4
xmail	363	303	1001
sed.fault2	7	54	30
find	49	2	66
sed.fault1*	12	1017	296
grep*	2	567	137
aspell*	256	134	10
htget*	2	44	210
gzip.fault2**	2	5	27
socat**	3	46	5
rsync**	2	156	2576

to measure the degree of similarity between these executions and to compute the ranked list of suspicious entities using both the unmodified fault-localization techniques considered and my optimized versions in F^3. As discussed in Section 5.2.1, my implementation operates at the branch level, so when a branch is filtered out, all statements control dependent on that branch are also eliminated.

Finally, to be able to assess the effectiveness of each individual optimization, I computed my results first with one optimization enabled at a time, and then with all optimizations enabled.

5.2.4 Results and Discussion

This section presents the results of my empirical study and discusses their implications in terms of the four research questions.

5.2.4.1 RQ1: Execution Generation

Table 6 shows the execution generation results. For each fault considered, identified by a unique fault ID, the table reports the number of entries in the minimized crash data (*i.e.,* the number of goals in the list fed to the execution generator) and the number of failing and passing executions synthesized by F^3 using this data.

As the table shows, the execution generator was able to synthesize both passing and failing executions for all faults considered. However, I can divide the faults in three groups, based on how these executions were synthesized. For three of the faults, the ones not marked, the generator was able to generate passing and failing executions using CS_{min}. For four other faults, those marked with a single star, the generator had to use a reduced list of goals to generate passing executions because all of the executions synthesized using the original CS_{min} were failing ones. For the remaining three faults, marked with two stars, the generator had to use an empty list of goals to generate passing executions. According to my intuition, and as discussed in Section 5.1.1, I expect the degree of similarity between the synthesized passing executions and the originally observed one to decrease as I go from the first group to the third. I discuss how this degree of similarity affects the effectiveness of the fault localizer when I present the fault-localization results.

Overall, F^3's execution generator was able to generate passing and failing executions for all programs, and corresponding failures, considered using CS_{min} or a subset thereof. I can therefore answer the first research question in a positive manner.

Answer to RQ1: *For the programs and failures considered,* F^3 *was able to synthesize multiple passing and failing executions.*

Before moving to RQ2, and discussing whether the generated executions can be useful for debugging, it is worth clarifying an important point. Because in several cases the number of entries in CS_{min} is extremely low, it is legitimate to wonder whether the program points in such CS_{min} can be used directly for fault localization. Intuitively, if program entities in CS_{min} were to provide enough guidance to the execution generator to allow it to reproduce the failure at hand, they could provide enough information to locate the fault(s) causing the failure. I investigated whether this was the case by manually checking the entries in all CS_{min} sets with size 20 or less. I found that, although such entries are useful to guide the execution towards program regions that contain the faulty code, they have a subtle and indirect connection with the exact location of the faults. In one case, for instance, the entry was in the same basic block of an assignment whose effect allowed the execution to reach

the faulty code in a completely different part of the program.

5.2.4.2 RQ2: Path Similarity of Generated Executions

Before I apply my customized fault localization techniques to the executions generated by my approach, I want to first answer RQ2 by evaluating the degree of similarity among the generated passing and failing executions. I will also leverage these results later on in the paper to find correlations between execution similarity and effectiveness of fault localization.

To measure similarity, I leverage the metrics proposed by Sumner, Bao, and Zhang [75] and, in particular, their formula for measuring path similarity of two executions e_1 and e_2:

$$execution_similarity(e_1, e_2) = \frac{2I_{both}}{I_{e_1} + I_{e_2}}$$

where I_{both} is the number of dynamic instructions executed in both e_1 and e_2, I_{e_1} is the number of dynamic instructions executed in e_1, and I_{e_2} is the number of dynamic instructions executed in e_2.

The first measure I computed is the degree of similarity within the set of generated failing executions. The first three columns in Table 7[1] show these similarity results. For each fault considered, identified by a unique fault ID, I report the average path similarity and the minimum path similarity between any two executions in the generated failing execution set, *FAIL*. As the table shows, the failing executions generated by my approach have a high degree of similarity: over 95% in 9 over 11 cases. The table also shows that the degree of similarity does not vary considerably across faults.

The second measure I computed is the degree of similarity between the sets of generated failing and passing executions, *FAIL* and *PASS*. Because my approach generates multiple different passing and failing executions for a given fault, I considered two sets of instructions: I_{FAIL}, the set of instructions that are executed by at least one of the generated failing executions, and I_{PASS}, the set of instructions that are executed by at least one of the generated passing executions. I then used the following formula to measure the similarity

[1] In this and in the following tables, the annotations no star, one star, and two stars have the same meaning they have in Table 6 and are repeated for the reader's convenience.

Table 7: The degree of path similarity among failing executions and the degree of path similarity between failing and passing sets.

Fault	Among Failing Executions		Between Failing and Passing Sets
	Average similarity	Minimum similarity	Set Similarity
exim	99.9%	99.8%	99.3%
xmail	97.8%	93.1%	78.4%
sed.fault2	93.3%	86.6%	90.1%
find	95.1%	95.1%	76.0%
sed.fault1*	98.0%	93.0%	49.2%
grep*	99.9%	99.6%	65.9%
aspell*	88.4%	85.7%	30.4%
htget*	98.7%	98.5%	82.9%
gzip.fault2**	98.5%	97.8%	33.0%
socat**	97.5%	96.0%	34.7%
rsync**	97.7%	95.7%	48.8%

of the two sets of executions $FAIL$ and $PASS$:

$$set_similarity(FAIL, PASS) = \frac{2|I_{FAIL} \cap I_{PASS}|}{|I_{FAIL}| + |I_{PASS}|}$$

The fourth column in Table 7 shows this second set of similarity results. As the results show, the similarity between the sets of failing and passing executions follows a general (expected) trend: the more entries our approach removed from the original execution data CS_{min} (indicated by the number of stars next to a fault ID), the more dissimilar were the generated passing executions to the generated failing executions. For the faults for which F^3 was able to generate passing executions using the original execution data (*i.e.,* `exim`, `xmail`, `sed.fault2`, and `find`), for instance, the generated passing executions are considerably more similar to the generated failing executions than for the other faults.

It is interesting to compare my similarity data with the similarity data presented in Sumner and colleagues' work, which uses pure symbolic execution to generate test inputs for a similar set of subjects and computed the similarity of the generated passing and failing executions [75]. Their similarity results were always below 1%, whereas my results

are always above 30%, which indicates that my guided generation does result in passing and failing executions that are more similar with one another and are likely to result in better statistical fault localization. I discuss how this degree of similarity affects the effectiveness of fault localization when I present my investigation of RQ3, in the next section.

In summary, and based on the similarity results, I can answer RQ2 as follows.

> ***Answer to RQ2:*** *For the programs and failures considered, there is high similarity among failing executions. The degree of similarity between passing and failing executions, conversely, is more variable; as expected, it decreases when the amount of execution data that* F^3 *must drop to generate passing executions increases.*

5.2.4.3 RQ3: Fault Localization

To answer RQ3, I applied F^3's fault localizer to the sets of passing and failing executions synthesized by

bugredux and summarized in Table 6. Table 8 presents the results of this study. The columns in the table show the fault ID, the similarity of the generated passing and failing executions (*i.e.,* the fourth column from Table 7), and the results of applying my customized versions of Ochiai, OBM, Naish1, and Naish2 with all three optimizations enabled, indicated as Ochiai+, OBM+, Naish1+, and Naish2+, to the corresponding failure. The results are presented as the absolute position of the actual fault in the ranked list of suspicious entities produced by the fault localizer over the size of that list. The last row of the table shows the Pearson product-moment correlation coefficients [71] between the similarity of passing and failing executions and the ranks of the actual faults.

The position in the table represents the number of program entities that developers would have to examine before finding the fault. For `exim`, for instance, my approach generates a ranked list of three suspicious entities, and developers would have to examine only the first one, which corresponds to the basic block that contains the actual fault. Note that, for cases in which the faulty entity had the same suspiciousness value as other entities, I reported in the table the worst-case result, that is, the case in which the actual fault was placed last in the list among the entities with the same suspiciousness. For example, if the

Table 8: Ranks of the faulty entity (over the total number of entities reported) using F^3's fully optimized fault-localization techniques.

Faults	Similarity	Ochiai+	OBM+	Naish1+	Naish2+
exim	99.3%	1/3	1/3	1/3	1/3
xmail	78.4%	1/3	1/3	1/3	1/3
sed.fault2	90.1%	1/11	8/11	8/11	8/11
find	76.0%	2/32	2/32	2/32	2/32
sed.fault1*	49.2%	13/19	13/19	13/19	13/19
grep*	65.9%	12/72	12/72	12/72	12/72
aspell*	30.4%	-/0 (1/45)	-/0 (6/45)	-/0 (6/45)	-/0 (6/45)
htget*	82.9%	-/0 (1/93)	-/0 (67/93)	-/0 (67/93)	-/0 (67/93)
gzip.fault2**	33.0%	3/80	49/80	49/80	49/80
socat**	34.7%	11/14	11/14	11/14	11/14
rsync**	48.8%	6/28	6/28	6/28	6/28
Rank-Similarity Correlation		-0.58	-0.64	-0.64	-0.64

real fault shared a same top suspiciousness value with 3 other program entities, the number of program entities we would report as those a developer has to inspect would be 4.

In two cases, `aspell` and `htget`, the filtering was too aggressive and eliminated all suspicious entities from the list. For these cases, I also report, in parentheses, the position of the fault in the list (over the total size of the list), before filtering was performed. I believe this is justified because, in cases in which the list of entities after filtering is empty, the sensible course of action would be to report the ranking computed without filtering.

Looking at the results in the table, I can make several observations. One first observation is that F^3's fault localizer, when operating on the synthesized passing and failing executions produced by the execution generator, is in most cases quite effective: for 2 out of 10 cases (3 in the case of Ochiai+), the real faulty entity is the first entry in the ranked list; and for another 4 cases (5 in the case of Ochiai+), it is within the first 15 entities in the list. Moreover, even for aspell and htget, the faulty entity is in position 1 when using Ochiai+ without filtering. Overall, the only negative results are for htget and gzip.fault2 when using OBM+, for which the faulty entities are ranked in position 67 and 49. In all other cases, F^3's fault localizer produces quite encouraging results.

Another observation I can make looking at the table is that the Pearson correlation coefficients support that the degree of similarity between the synthesized passing executions

and failing executions is negatively correlated to the rank of actual fault. In other words, the degree of similarity is positively correlated to the effectiveness of fault localization performed using that set. (*i.e.,* The more similar between the synthesized passing and failing executions, the higher positions the actual faults are ranked.) For the four faults for which my approach was able to generate passing executions that are more than 70% similar to the failing executions, the faulty entities were ranked first by Ochiai+. For other faults, for which my approach generated passing executions that are not as similar to the failing executions, fault localization is generally less effective. (One counterexample is `aspell`, for which Ochiai+ was able to rank the real fault first using sets of less similar executions. I found that, in this case, the passing executions executed instructions that were not executed by failing executions, which lowered the degree of similarity without affecting the effectiveness of fault localization.)

To get more insight on this aspect of the technique, I manually checked the ranked lists generated by F^3. Interestingly, I found that, in most cases, the fault localizer did assign the faulty entity the highest suspiciousness value. However, other program entities related to this faulty code were also assigned the same value. (Because I report the worst-case result, as described above, this lowered the rank of the faulty entity that I report.)

One final observation I can make about the results is that, although these techniques have slightly different metrics, they produce very similar fault localization results. In particular, Ochiai+ produces consistently better results mostly because of the profiling optimization. This result indicates that the effectiveness of my fault localization techniques mainly depends on the executions generated by my approach as well as the optimizations in F^3 and is independent from the four baseline statistical fault localization techniques, as they all leverages a similar set of dynamic information and similar heuristics. However, I would need further experimentation to confirm this result.

Overall, F^3's fault localizer was in most cases effective when applied to the passing and failing executions synthesized by the execution generator, which lets us answer also the third research question in a positive manner.

Answer to RQ3: *For the programs and failures considered,* F^3 *can leverage the passing and failing executions it synthesizes to perform fault localization effectively in most cases. In addition, there is initial evidence that the degree of similarity between the synthesized passing and failing executions positively affects the effectiveness of fault localization performed on these executions.*

5.2.4.4 RQ4: Effectiveness of My Fault Localization Optimizations

To answer RQ4, in this section I study the results of applying my two main fault-localization optimizations and one engineering optimization separately, so as to assess the improvement made by each individual optimization.

Filtering Before discussing the extent to which filtering can improve fault localization for field failures, in Table 9 I show the size of the filtering sets (see Section 5.1.2.2) computed by F^3 for the programs and faults considered in my study. For each fault considered, the table shows the total number of branches exercised by all failing and passing executions (Total) and the size of the FIL_f, FIL_{fp}, and FIL_{dep} sets. Because these sets are increasingly restrictive, their sizes decrease when going from FIL_f to FIL_{dep}. Intuitively, the numbers in the table show how aggressive are the different filters in reducing the amount of code to be considered by statistical fault localization. For xmail, for instance, all failing and passing executions exercise 141 branches, whereas filtering based on FIL_f, FIL_{fp}, and FIL_{dep} results in only 74, 6, and 3 branches to consider.

Interestingly, sets FIL_{fp} and FIL_{dep} are considerably smaller than sets FIL_f, whose size is close to the total number of exercised branches. Moreover, the difference in the sizes tend to decrease as we go down the table, which provides further evidence of the difference in the similarity of passing and failing executions for the different programs. Intuitively, higher similarity would result in a higher number of common branches between passing and failing executions, and thus in smaller FIL_{fp} (and FIL_{dep}) sets.

As I explained in Section 5.1.2.2, when using filtering, F^3 selects a subset of branches and applies fault localization only to the code corresponding to these branches, thus eliminating

Table 9: Total number of branches exercised by the synthesized passing and failing executions and size of the corresponding filtering sets (see Section 5.1.2.2).

Faults	Total	# FIL_f	# FIL_{fp}	# FIL_{dep}
exim	1379	1359	3	3
xmail	141	74	6	3
sed.fault2	715	456	29	11
find	195	177	69	32
sed.fault1*	158	130	44	19
grep*	278	276	96	72
aspell*	45	6	0	0
htget*	93	67	0	0
gzip.fault2**	213	202	153	80
socat**	53	36	28	14
rsync**	106	91	57	28

from consideration a large percentage of program entities. Because of the way the filtering sets are computed, they are not guaranteed to be non-empty for a given program, fault, and set of executions. As Table 9 shows, for the programs considered, set FIL_{dep} is non-empty in 8 out of 10 cases.

Table 10 shows the results of fault localization enhanced with filtering. Similar to Table 8, the results are presented in terms of position of the actual fault in the ranked list produced by the fault localizer. The columns in the table are divided into two groups, one for each type of fault-localization technique considered, that is, Ochiai and OBM. The table presents four results for each of these two techniques: results computed without filtering (Orig) and results computed with FIL_f, FIL_{fp}, and FIL_{dep} filtering.

The results in the table show that the first two filters, FIL_f and FIL_{fp}, do not improve the fault-localization results (with the only exception of OBM, when run on grep with the FIL_{fp} filter). This result is somehow expected, as entities that are executed by all failing (resp., passing) executions should be assigned high (resp., low) suspiciousness values when using traditional fault localization techniques. Even in these cases, however, filtering based on the FIL_f and FIL_{fp} sets can still be useful because it can considerably reduce the overall size of the ranked list of suspicious entities, as discussed in Section 5.1.2.2 and shown in Table 9.

The results are different in the case of the third filter, FIL_{dep}, which can considerably

Table 10: Positions of the actual faults in the ranked list produced by the fault localizer when using traditional Ochiai, Ochiai with three customized types of filtering, traditional OBM, and OBM with three customized types of filtering

Faults	Ochiai				OBM				Naish1				Naish2			
	Orig	FIL$_f$	FIL$_{fp}$	FIL$_{dep}$	Orig	FIL$_f$	FIL$_{fp}$	FIL$_{dep}$	Orig	FIL$_f$	FIL$_{fp}$	FIL$_{dep}$	Orig	FIL$_f$	FIL$_{fp}$	FIL$_{dep}$
exim	1	1	1	**1**	1	1	1	**1**	1	1	1	**1**	1	1	1	**1**
xmail	1	1	1	**1**	1	1	1	**1**	1	1	1	**1**	1	1	1	**1**
sed.fault2	24	24	24	**8**	24	24	24	**8**	24	24	24	**8**	24	24	24	**8**
find	3	3	3	**2**	3	3	3	**2**	3	3	3	**2**	3	3	3	**2**
sed.fault1*	38	38	38	**15**	38	38	38	**15**	38	38	38	**15**	38	38	38	**15**
grep*	40	40	40	**23**	48	48	40	**23**	40	40	40	**23**	40	40	40	**23**
aspell*	6	**6**	-	-	6	**6**	-	-	6	**6**	-	-	6	**6**	-	-
htget*	67	**67**	-	-	67	**67**	-	-	67	**67**	-	-	67	**67**	-	-
gzip.fault2**	84	84	84	**49**	84	84	84	**49**	84	84	84	**49**	84	84	84	**49**
socat**	26	26	26	**13**	26	26	26	**13**	26	26	26	**13**	26	26	26	**13**
rsync**	11	11	11	**6**	11	11	11	**6**	11	11	11	**6**	11	11	11	**6**

improve the effectiveness of fault localization. For the faults in my study, in fact, FIL_{dep} filtering was able to improve the ranking of the faulty entities by 50% on average. Ideally, with these improvements, developers would have to inspect only half of the entities that they would have to inspect using a set of unfiltered results.

In general, the results show that filtering can improve traditional fault-localization techniques in two ways: (1) filtering based on the FIL_f and FIL_{fp} sets can almost never improve the ranking of the faulty entity, but it can dramatically reduce the size of the list of suspicious entities; (2) filtering based on the FIL_{dep} set can also improve the ranking of the faulty entities.

Profiling I now discuss the results of using profiling instead of coverage information for calculating suspiciousness values in Ochiai. As I explained in Section 5.1.2.3, I considered only Ochiai because OBM cannot be easily extended to use this additional information.

Table 11 shows the results of fault localization enhanced with profiling information. Similar to Table 10, the results in the Orig column are those for Ochiai without any modification, whereas the results in column Profiling are based on the modified Ochiai. As the results in the table shows, using profiling information improves the effectiveness of the Ochiai technique dramatically in 4 out of 10 cases, for sed.fault2, aspell, htget and

Table 11: Positions of the actual faults in the ranked list produced by Ochiai with and without profiling information

Faults	Ochiai	
	Orig	Profiling
exim	1	1
xmail	1	1
sed.fault2	24	1
find	3	3
sed.fault1*	15	15
grep*	40	38
aspell*	6	1
htget*	67	1
gzip.fault2**	84	3
socat**	26	26
rsync**	11	11

gzip.fault2, and marginally in one additional case, for grep. (For these cases, using F^3 instead of the traditional Ochiai technique would likely allow developers to locate the fault at once.) In 2 of the other cases, exim and xmail, there is no room for improvement, and for the remaining three cases profiling does not affect the results at all. After manually checking the types of the four faults where profiling information improves effectiveness of fault-localization techniques significantly, I observed that they were mostly buffer overflows, which were caused by bugs inside loops. This finding supports my intuition that profiling is the ideal type of runtime information for localization of some types of faults, as discussed in Section 5.1.2.3.

Overall, the results provide evidence that the use of profiling information can dramatically improve the effectiveness of fault localization for at least some types of faults.

Grouping I now examine the usefulness of my engineering optimization, grouping, when applied to both Ochiai and OBM. To this end, Table 12 shows the comparison between the effectiveness of fault localization with and without grouping.

The results in the table show that also grouping can considerably improve the effectiveness of fault localization. Out of the 10 cases considered, and excluding the 2 cases that cannot be improved because the faulty entity is already ranked first, applying grouping

Table 12: Positions of the actual faults in the ranked list produced by the fault localizer when using the original Ochiai and OBM techniques with and without grouping

Faults	Ochiai		OBM		Naish1		Naish2	
	Orig	Grouping	Orig	Grouping	Orig	Grouping	Orig	Grouping
exim	1	**1**	1	**1**	1	**1**	1	**1**
xmail	1	**1**	1	**1**	1	**1**	1	**1**
sed.fault2	24	**17**	24	**17**	24	**17**	24	**17**
find	3	**2**	3	**2**	3	**2**	3	**2**
sed.fault1*	38	**23**	38	**23**	38	**23**	38	**23**
grep*	40	**22**	48	**26**	40	**22**	40	**22**
aspell*	6	**4**	6	**4**	6	**4**	6	**4**
htget*	67	**21**	67	**21**	67	**21**	67	**21**
gzip.fault2**	84	**35**	84	**35**	84	**35**	84	**35**
socat**	26	**14**	26	**14**	26	**14**	26	**14**
rsync**	11	**9**	11	**9**	11	**9**	11	**9**

always increases the ranking of the faulty entities. In some cases, such as for aspell and rsync, the improvement is marginal. In most other cases, however, the improvement is considerable.

Answer to RQ4: *For the programs and failures considered, most of my optimizations can actually improve the effectiveness of fault localization, albeit to different extents. Aggressive filtering and grouping can often increase the ranking of faulty entities, and profiling information can dramatically improve the effectiveness of fault localization for at least some types of faults.*

5.2.4.5 Limitations and Threats to Validity

One of the main limitations of my approach is that it reuses part of the implementation of BugRedux, which in turn relies on symbolic execution—a complex and expensive approach. Despite the inherent technical limitations of symbolic execution, however, recent advances in this area have considerably improved the practical applicability of these approaches.

Like for every empirical evaluation, there are threats to the internal and external validity of my results. First, there may be faults in my implementation that might have affected my results. To address this threat I manually checked many of my results (and did not

encounter any error). Another threat is that I used my own implementation of the fault-localization techniques for my studies. Because the techniques considered consist of fairly simple formulas, I feel confident that I implemented the techniques correctly and in an unbiased way. Finally, in my studies I considered only ten programs, so my findings may not generalize to other programs or faults. However, the programs I considered are from different repositories, were used also in previous research [18,29,51], and seven of the faults in these programs are *real bugs found by real users in the field*, that is, *real field failures*. I nevertheless plan to perform an additional extensive empirical evaluation, including user studies, to confirm my results.

5.3 Conclusion

Understanding and debugging field failures is a notoriously difficult task because of the increasing complexity of modern software systems and the limited availability of information from the field (typically limited to crash stacks). To mitigate this problem, and better help developers identify the likely causes of field failures, I have proposed F^3, a technique that extends my field-failure reproduction technique with automated fault-localization capabilities. Given a field failure, F^3 (1) synthesizes passing and failing executions that are similar to the original failure and (2) leverages the generated executions to perform fault localization using a set of suitably optimized fault-localization techniques.

To assess the effectiveness of my approach, I implemented it in a prototype tool and performed an empirical evaluation on a set of *real-world programs and real field failures*. My results, albeit still preliminary, are promising. First, they showed that F^3 was able to synthesize multiple passing and failing executions and successfully use the synthesized executions for fault localization. Second, the results also showed that the optimizations that I propose are effective; in many cases, my optimizations were able to (1) reduce the length of the lists of suspicious program entities reported to developers and (2) improve the ranks of the faulty program entities in such lists. Although user studies are required to confirm my findings, both of these improvements have the potential to considerably reduce the developers' effort needed to identify the causes of field failures.

CHAPTER VI

FURTHER IMPROVEMENT ON FAULT LOCALIZATION

In this chapter, I propose two techniques to improve a principle fault localization technique, formula-based debugging. Formula-based debugging, different from traditional debugging techniques, can generate as effective, more accurate, and actionable suggestions than traditional debugging techniques and address some existing problems of such techniques. In the rest part of this chapter, I will first describe the motivation, introduce the two approaches, and finally present some preliminary investigations on different aspects of my proposed approaches.

6.1 Motivation

As discussed in previous chapters, debugging is a notoriously difficult, expensive, and time consuming development activity. For this reason, there has been a great deal of research on automated techniques for supporting various debugging tasks. For example, in F^3, I have leveraged statistical debugging (*e.g.,* [12, 15, 19, 53, 57, 58, 60, 73, 87]), which performs statistical inference on a set of passing and failing test cases and uses the results of such inference to rank program entities in decreasing order of "suspiciousness" (*i.e.,* likelihood of being related to the failure). Another type of debugging techniques, various flavors of delta debugging (*e.g.,* [83–85]), which aims to identify failure causes through differencing, can be also used for debugging purposes. Unfortunately, these techniques mostly rely on heuristics and intuitions, which would introduce some noises and make the final result difficult to consume by developers as shown in several previous studies [52, 68].

To mitigate this problem, more recently, there has been a considerable interest in techniques that can perform fault localization in a more principled way (*e.g.,* [26, 39, 54, 76]). These techniques, collectively called *formula-based debugging*, model faulty programs and failing executions as formulas and perform fault localization by manipulating and solving these formulas. As a result, they can provide developers with the possible location of the

fault, together with a mathematical explanation of the failure (*e.g.,* the fact that an expression should have produced a different value or that a different branch should have been taken at a conditional statement).

One technique of particular interest, in this arena, is Jose and Majumdar's BugAssist [54]. Given a faulty program, a failing input, and a corresponding (violated) assertion, BugAssist performs fault localization by constructing an unsatisfiable Boolean formula that encodes (1) the input values, (2) the semantics of (a bounded version of) the faulty program, and (3) the assertion. It then uses a pMAX-SAT solver to find maximal sets of clauses in this formula that can be satisfied together and outputs the complement sets of clauses (CoMSS) as potential causes of the error. Intuitively, each set of clauses in CoMSS indicates a corresponding set of statements that, if suitably modified (*e.g.,* replacing the statements with angelic values [24]), would make the program behave correctly for the considered input.

Although effective, BugAssist is extremely computationally expensive, as it builds a formula for (a bounded unrolling of) *all possible paths* in a program. This can lead to formulas with millions of terms [54] and high computational cost even for small programs. Moreover, BugAssist, like most formula-based debugging approaches, does not take into account passing test cases, thus missing two important opportunities. First, passing executions can help identify statements, and thus parts of the formulas, that are less likely to be related to the fault, which can help optimizing the search for a solution to such formulas. Second, passing executions can help filter out locations that may be potential fixes for the failing executions considered but could break previously passing test cases if modified [24].

In this chapter, I propose two possible ways of addressing these issues and improving formula-based debugging approaches: *on-demand formula computation (OFC)* and *clause weighting (CW).* OFC is a novel on-demand algorithm that can dramatically reduce the number of paths encoded in a formula, and thus the overall complexity of such formula and the cost of computing a pMAX-SAT solution for it. Intuitively, my algorithm (1) builds a formula for the path in the original failing trace, (2) analyzes the formula to identify additional relevant paths to consider, (3) expands the formula by encoding these additional paths, (4) repeats (2) and (3) until no more relevant paths can be identified, at which

point it (5) reports the computed solution. CW accounts for the information provided by passing test cases by assigning weights to the different clauses in an encoded formula based on the suspiciousness values computed by a statistical fault localization technique. Doing so has the potential to improve the accuracy of the results by helping the solver compute CoMSSs that are more likely to correspond to faulty statements. (The guidance provided to the solver can also unintentionally improve the efficiency of the approach, as we show in Section 6.3.2.1.)

To assess the effectiveness of OFC and CW, I selected BugAssist as a baseline and considered four different formula-based debugging techniques: the original BugAssist, BugAssist+CW, OFC, and OFC+CW. I implemented all four techniques in a tool that works on C programs and used the tool to perform an empirical study. In the study, I first applied the four techniques to 52 versions of two small programs to assess several tradeoffs involved in the use of CW and OFC and compare with related work. My results are encouraging, as they show that CW and OFC can improve the performance of BugAssist in several respects. First, the use of CW resulted in more accurate results—in terms of position of the actual fault in the ranked list of statements reported to developers—in the majority of the cases considered. Second, CW and OFC were able to reduce the computational cost of BugAssist by 27% and 75% on average, respectively, with maximum speedups of over 70X for OFC. To further demonstrate the practicality of CW and OFC, I also performed a case study on a real-world bug in Redis, a popular open source project. Overall, the results show that CW and OFC are promising, albeit initial, steps towards more practically applicable formula-based debugging techniques and motivate further research in this direction.

6.2 Improving Formula-based Debugging

In this section, I present two approaches for improving formula-based debugging: clause weighting and on-demand formula computation. We discuss them in detail using BugAssist [54] as a representative of state-of-the-art formula-based debugging techniques and our baseline.

6.2.1 Clause Weighting (CW)

CW consists of using the information from passing executions to inform a wpMAX-SAT solver. More precisely, CW leverages the suspiciousness values computed by a statistical fault localization technique and assigns to each program entity en, and thus to the corresponding clause in the program formula, a weight inversely proportional to its suspiciousness $susp(en)$: $weight(en) = 1/susp(en)$. If the suspiciousness value of an entity is zero, which means that the entity is only executed by passing tests, CW assigns to it the largest possible weight. By assigning different weights to different clauses, CW transforms the original pMAX-SAT problem in BugAssit into a wpMAX-SAT problem. The rationale for CW is that, by the definition of wpMAX-SAT, clauses with higher weights are more likely to be included in an MSS (*i.e.,* less likely to be identified as causes of the faulty behavior), while clauses with lower weights are less likely to be included in an MSS (*i.e.,* more likely to be included in a CoMSS and thus be identified as causes of the faulty behavior).

Formula-based debugging techniques such as BugAssist consider all possible pMAX-SAT solutions equally and simply report them. Conversely, by leveraging the heuristics in statistical fault localization, CW is more likely to rank the set of clauses corresponding to the fault at the top of the list of solutions, thus reducing developers' debugging effort. This potential advantage, however, comes at a cost. Solving wpMAX-SAT problems can be computationally more expensive than solving a pMAX-SAT problem, which can outweigh CW's benefits. To understand this tradeoff, in the empirical evaluation in this chapter, I assess how CW affects the accuracy and efficiency of formula-based debugging (see Section 6.3.2.1).

6.2.2 On-demand Formula Computation (OFC)

OFC is my second, and more substantial, improvement over traditional formula-based debugging techniques. Figure 7 shows an overall view of OFC and its workflow. The inputs to the algorithm are a faulty program, represented as an Inter-procedural Control Flow Graph (ICFG), and a test suite that contains a set of passing tests and one failing test. As it is common practice for debugging techniques, we assume that a failure can be expressed as the violation of an assertion in the program. Given these inputs, OFC produces as output

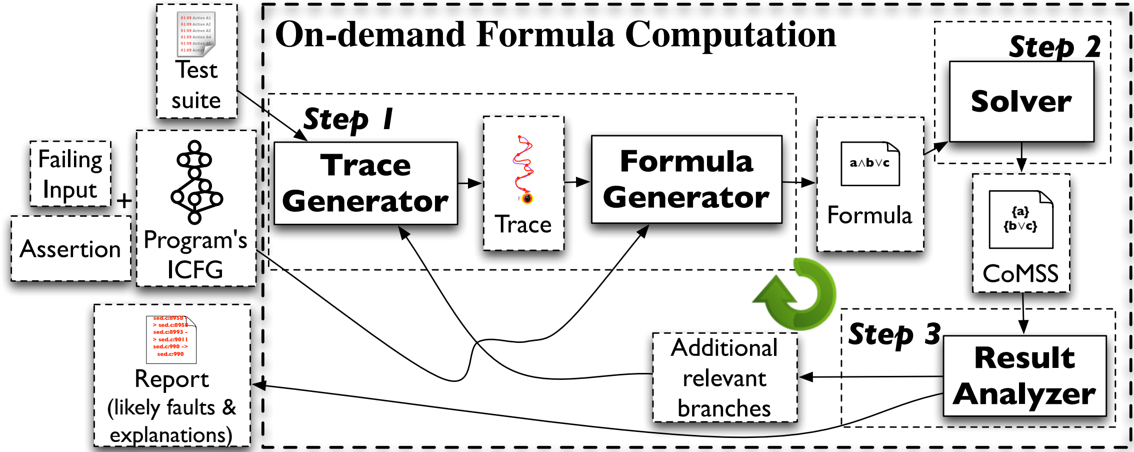

Figure 7: Overview of on-demand formula computation

a set of clauses and their corresponding program entities (*i.e.,* branches and statements). These are entities that, if suitably modified, would make the failing execution pass. The expressions in the reported clauses provide developers with additional information on the failure, and can be considered a "mathematical explanation" of the failure.

As Figure 7 shows, OFC consists of three main steps. The key idea behind OFC is to reason about the failure (and the program) incrementally, by starting with the entities traversed in a single failing trace, computing CoMSS solutions for the partial program exercised by the trace, and then expanding the portion of the program considered in the analysis when such solutions indicate that additional control-flow paths should be taken into consideration to "explain" the failure. Specifically, in its first step (Section 6.2.2.1), OFC generates a new trace (the original failing trace, in the first iteration) and suitably updates the trace formula, a formula that encodes the semantics of the traces generated so far. OFC's second step (Section 6.2.2.2) computes the CoMSSs of the (unsatisfiable) formula built in the previous step. Finally, in OFC's third step, the algorithm checks whether there is any additional relevant branch to consider in the program (Section 6.2.2.3). If so, OFC returns to Step 1. Otherwise, it computes all possible CoMSSs of the final formula to report to developers the set of relevant clauses and their corresponding program entities.

Algorithm 3 shows the main algorithm, which takes as inputs the ICFG of the faulty program and the program's test suite and performs the three steps we just described. I

```
    Input  : ICFG: ICFG of the faulty program
             TestSuite: test suite for the program
    Output:  faulty statements and their corresponding clauses
 1  begin
 2      FIN ← GetFailingInput(TestSuite)
 3      ASSERT ← GetFailingAssertion(TestSuite)
 4      TF ← {}
 5      SP ← {}
 6      clause_origin ← {}
 7      visited_branches ← {}
 8      flip_br ← null
        // Step 1
 9      new_trace ← TraceGenerator(FIN, visited_branches, flip_br)
10      flip_br ← null
11      TF ← FormulaGenerator(new_trace, TF, ICFG, SP, clause_origin)
        // Step 2
12      CoMSSs ← Solver(FIN, ASSERT, TF)
        // Step 3
13      foreach CoMSS in CoMSSs do
14          foreach clause in CoMSS do
15              st ← clause_origin(clause)
16              if st is a conditional statement then
17                  ¡true_br, false_br¿ ← getBranches(st)
18                  if visited_branches(true_br)==null then
19                      flip_br ← false_br
20                      go back to Step 1
21                  end
22                  if visited_branches(false_br)==null then
23                      flip_br ← true_br
24                      go back to Step 1
25                  end
26              end
27          end
28      end
29      foreach CoMSS in CoMSSs do
30          foreach clause in CoMSS do
31              report clause and clause_origin(clause)
32          end
33      end
34  end
```

Algorithm 3: OFC

discuss each step in detail in the rest of this section.

6.2.2.1 Trace Generator and Formula Generator

After an initialization phase, OFC iterates Steps 1, 2, and 3. Step 1 performs two tasks: trace generation and formula generation.

Trace Generator In its first part, Step 1 invokes the *Trace Generator* (Algorithm 4). In the first iteration of the algorithm, *Trace Generator* generates the trace corresponding to the failing input. In subsequent iterations, it generates a trace that covers the new program entities identified as relevant by Step 3 (see Section 6.2.2.3), so as to augment the scope of the analysis. The inputs to *TraceGenerator* are the failing input, the map that associates

```
   Input  : FIN: failing input
            visited_branches: map from branches to traces that covered them
            flip_br: branch for which a new trace must be generated
   Output: new_trace: newly generated trace
 1 begin
 2     if flip_br==null then
 3         new_trace ← Execute(Input, null, null)
 4     else
 5         old_trace ← visited_branches(flip_br)
 6         new_trace ← Execute(Input, old_trace, flip_br)
 7     end
 8     foreach br in new_trace do
 9         if visited_branches(br)==null then
10             visited_branches(br) ← new_trace
11         end
12     end
13     return new_trace
14 end
```

Algorithm 4: TraceGenerator

each branch covered so far with the trace in which it was first covered, and the new relevant branch for which a trace must be generated (by flipping it).

If *flip_br* is null, which only happens in the first iteration of the algorithm, *TraceGenerator* generates a trace by simply providing the failing input to the program and collecting its execution trace (line 3). Otherwise, for subsequent iterations, *TraceGenerator* retrieves *old_trace* (line 5), the trace that first reached branch *flip_br* and generates a new trace, *new_trace* (line 5). To generate the trace, the algorithm provides the failing input to the program, forces the program to follow *old_trace* up to *flip_br*, and flips *flip_br* so that the program follows its alternative branch (using execution hijacking [77]). The algorithm also updates map *visited_branches* by adding to it an entry for every branch newly covered by *new_trace*, including *flip_br*'s alternative branch (lines 8–12).

Formula Generator After generating a trace, OFC invokes *FormulaGenerator* (Algorithm 5), which constructs a new formula *TF*, either from scratch (in the first iteration) or by expanding the current formula based on the program entities in *new_trace* (in subsequent iterations).

The inputs to *FormulaGenerator* are the ICFG of the faulty program, the current trace formula, the portion of the program currently considered (and encoded in the current trace formula), the trace newly generated by *TraceGenerator*, and a map from clauses to statements that originated them.

```
Input  : ICFG: ICFG of the faulty program
         TF: current trace formula
         SP: portion of the program currently considered
         new_trace: newly generated trace
Output: TF: updated trace formula
         SP: updated portion of the program currently considered
1  clause_origin: map from clauses to statements that originated them
2  begin
3      foreach st ∈ new_trace do
4          if st ∉ SP then
5              SP ← SP + st
6              if st is a conditional statement then
7                  predicate_st ← GetPredicate(st)
8                  clause_st ← (guard_st = predicate_st)
9              else
10                 if st is a φ function then
11                     phi ← φ function in st
12                     cs ← φ's conditional statement
13                     guard_cs ← cs's condition
14                     clause_st ← ((guard_cs ∧ (st_LHS = st_RHS,t)) ∨ (¬guard_cs ∧ (st_LHS = st_RHS,f)))
15                 else
16                     clause_st ← (st_LHS = st_RHS)
17                 end
18             end
19             clause_origin(clause_st) = st
20             TF ← TF ∧clause_st
21         end
22     end
23     return TF
24 end
```

Algorithm 5: FormulaGenerator

In its main loop, *FormulaGenerator* processes each statement *st* in the new trace, *new_trace*, one at a time. If *st* is not yet part of *SP*, the portion of the program currently considered, the algorithm (1) adds *st* to *SP*, (2) encodes its semantics in a new Boolean clause $clause_{st}$, (3) conjoins $clause_{st}$ and *TF*, and (4) updates map *clause_origin* by mapping $clause_{st}$ to *st*.

Similar to other symbolic analyses (*e.g.,* [27, 54, 76]), OFC operates on an SSA form of the faulty program (see Section 2.4). The formula generator models three types of statements in the program (and its trace): conditional statements (*e.g.,* line 1 in Figure 8), definitions that involve a ϕ function (*e.g.,* line *phi*1 in Figure 8 (right)) and definitions that do not involve a ϕ function. Intuitively, whereas the last type of statements represent traditional data-flow information about uses and definitions, the other two types encode control-flow information about branch conditions and ϕ function selection conditions. To perform a correct semantic encoding, when deriving $clause_{st}$ from *st*, *FormulaGenerator* must treat these three types of statements differently.

If *st* is a conditional statement with predicate $predicate_{st}$, the algorithm retrieves such predicate from *st* (line 7) and encodes *st* as $(guard_{st}{=}predicate_{st})$, where $guard_{st}$ is a Boolean variable that represents *st*'s condition (line 8).

If *st* involves a ϕ function *phi*, the algorithm generates a clause $(guard_{cs} \wedge (st_{LHS} = st_{RHS,t})) \vee (\neg guard_{cs} \wedge (st_{LHS} = st_{RHS,f}))$, where (1) *cs* is *phi*'s conditional and, similar to above, $guard_{cs}$ represents *cs*'s condition, (2) st_{LHS} is the variable being defined at *st*, and (3) $st_{RHS,t}$ and $st_{RHS,f}$ are the definitions selected by *phi* along *cs*'s *true* and *false* branches. Basically, this clause explicitly represents the semantics of *phi* and encodes both the data- and the control-flow aspects of the execution, which allows OFC to handle faults in both. Algorithm 5 performs this encoding at lines 10–14.

Finally, if *st* is a traditional assignment statement, the algorithm encodes *st* as $st_{LHS} = st_{RHS}$, the equivalence relation between the variable on *st*'s lefthand side and the expression on its righthand side (line 16). Because each assignment in SSA form defines a new variable, $clause_{st}$ can be simply conjoined with the current formula *TF* (line 20).

After processing a statement *st* and generating the corresponding clause $clause_{st}$, the algorithm records that $clause_{st}$ was generated from *st* and suitably updates the trace formula *TF* (lines 19 and 20). Finally, after processing all statements in *new_trace*, *FormulaGenerator* returns *TF*.

6.2.2.2 Solver

In its second step, OFC leverages a pMAX-SAT solver to find all possible causes of the failure being considered. To do so, it invokes function *Solver* and passes to it the failing input, the failing assertion, and the trace formula constructed in Step 1 (line 12 of Algorithm 3). Function *Solver* will first generate a formula by conjoining the input clauses (*i.e.,* clauses that assert that the input is the failing input *FIN*), the current trace formula *TF*, and the failing assertion *ASSERT*. Because *FIN* causes the program to fail, that is, to violate *ASSERT*, the resulting formula is unsatisfiable.

To suitably define the pMAX-SAT problem, *Solver* encodes (1) the input clauses and the failing assertion as hard clauses, (2) the clauses in *TF* generated from ϕ functions as

hard clauses, and (3) the other clauses in *TF* as soft clauses. The input clauses and the assertion are encoded as hard clauses because the failure could be trivially eliminated by changing the input or the assertion, which would not provide any information on where the problem is in the program. Encoding clauses generated by ϕ functions as hard clauses, conversely, ensures that control-flow related information is kept in the results, which is necessary to handle control-flow related faults. At this point, function *Solver* passes the so defined pMAX-SAT problem to an external solver and retrieves from it all possible CoMSSs for the problem (see Section 2.3).

If CW were also used, OFC would generate a wpMAX-SAT problem instead by assigning a weight to each soft clause based on the suspiciousness of the corresponding program entity (*i.e., clause_origin(clause)*), as described in Section 6.2.1.

6.2.2.3 Result Analyzer

OFC's third step takes the set of CoMSSs for the failure being investigated, produced by Step 2, and generates a report with a set of program entities (or an ordered list of entities, if I use CW and a wpMAX-SAT solver) and corresponding clauses. The entities are statements that, if suitably modified, would make the failing execution pass (*i.e.,* the potential causes of the failure being investigated). The expressions in the clauses associated with the statements provide developers with additional information on how the statements contribute to the failure, and as stated above, can thus be seen as a mathematical explanation of the failure.

This part of OFC, corresponding to lines 13–27 of Algorithm 3, iterates through each clause of each CoMSS computed in Step 2. For each clause, it first retrieves the corresponding statement *st*. If *st* is a conditional statement, the predicate in the conditional statement is potentially faulty, and taking a different branch may fix the program. To account for this possibility, the algorithm checks whether the conditional has one branch that has not been executed in any previously computed trace and, if so, expands the scope of the analysis by selecting that branch as a new branch to analyze and going back to Step 1 (lines 16–24). Step 1 would then add such branch to the list of relevant branches, generate a new trace, constructs a new formula, and perform an additional iteration of the analysis. Conversely, if

```
int  P(int x, int y) {
1.  if (x>=0)
2.    a = x;
3.  else
4.    a = -x;

5.  if (y<5)
6.    b = a+1;
7.  else
8.    b = a+2;

9. assert(b<=a);
}
```

```
int  P(int x1, int y1) {
   1.  if (x1>=0)
   2.    a1 = x1;
   3.  else
   4.    a2 = -x1;
phi1.  a3 = φ1(a1,a2);
   5.  if (y1<5)
   6.    b1=a3+1;
   7.  else
   8.    b2=a3+2;
phi2.  b3 = φ2(b1,b2);
   9.  assert(b3<=a3);
}
```

Figure 8: Example code in normal (left) and SSA (right) form

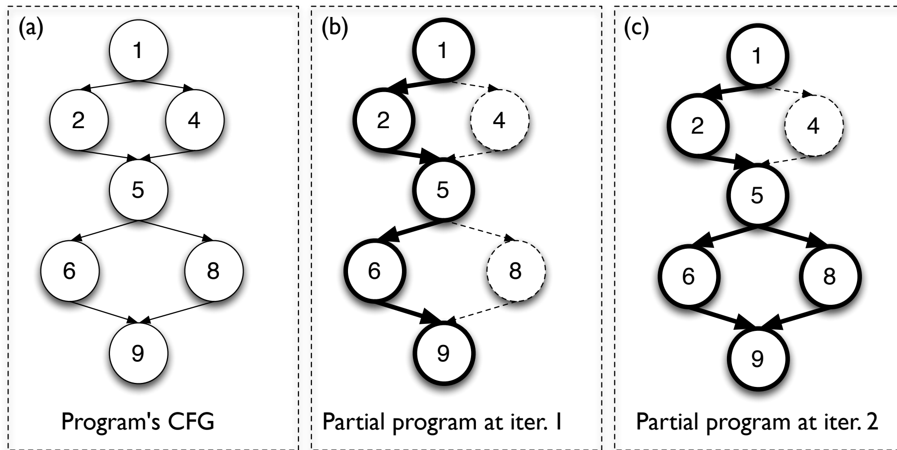

Figure 9: Control flow graph of P (a) and partial P considered during the first (b) and second (c) iteration of OFC

both branches have already been covered, or *st* is not a conditional statement, the algorithm continues and processes the next clause.

If no clause in any CoMSS contains a conditional statement for which one of the branches has not been covered, it means that the analysis already considered the portion of the program relevant to the failure, so the algorithm can terminate and produce a report (lines 29–32). To do so, OFC iterates once more through the set of CoMSSs computed during its last iteration. For each clause in each CoMSS, OFC reports it to developers, together with its corresponding statement, as a possible cause (and partial explanation) of the failure.

6.2.2.4 Illustrative Example

I now recap how the different parts of OFC work together using a simple program P as an illustrative example. Figure 8 shows P (left) and its SSA form (right), whereas Figure 9(a) shows the P's Control Flow Graph (CFG). P takes two integer inputs and contains an assertion at line 9. If we provide P with input $\{x = 0, y = 0\}$, the assertion is violated, as the execution results in $b = 1$ and $a = 0$ at line 9.

This is the starting point of OFC: A faulty program in SSA form (P), a failing test case ($\{x_1 = 0, y_1 = 0\}$), and an assertion violated by the failing test case ($b_3 \leq a_3$). Given these inputs, OFC operates in three iterative steps.

In Step 1 of the first iteration, the *Trace Generator* feeds the failing input to P, which results in the failing trace $\{1, 2, phi1, 5, 6, phi2, 9\}$. This trace identifies the partial program shown in Figure 9(b), where the entities drawn in boldface are in the partial program, and those drawn with dashed lines are ignored. (For simplicity, in the CFG I do not show nodes corresponding to ϕ functions.) Given the generated trace, the *Formula Generator* computes a trace formula that encodes the semantics of the partial program with respect to the trace:

$$
\begin{aligned}
TF_1 = & (guard_1 = (x_1 \geq 0)) \wedge (a_1 = x_1) \wedge \\
& ((guard_1 \wedge (a_3 = a_1)) \vee (\neg guard_1 \wedge (a_3 = a_2))) \wedge \\
& (guard_2 = (y_1 < 5)) \wedge (b_1 = a_3 + 1) \wedge \\
& ((guard_2 \wedge (b_3 = b_1)) \vee (\neg guard_2 \wedge (b_3 = b_2)))
\end{aligned}
$$

In the trace, the second and fifth clauses correspond to statements that do not involve ϕ functions. The first and fourth clauses correspond to the predicates at lines 1 and 5 and contain two extra variables, $guard_1$ and $guard_2$, that represent such predicates. The third and sixth clauses represent the two statements at lines $phi1$ and $phi2$, in which ϕ_1 and ϕ_2 define a_3 and b_3. These clauses encode the information on which variable a ϕ function may select and under which condition (*i.e.,* the outcome of the guard), as described in Section 6.2.2.1.

Step 2 of the algorithm then conjoins input conditions and failing assertion with TF_1 to obtain the following complete, unsatisfiable formula: $CF_1 = (x_1 = 0) \wedge (y_1 = 0) \wedge TF_1 \wedge (b_3 \leq a_3)$

The algorithm now marks input clauses, failing assertion, and the two clauses generated from ϕ functions as hard clauses, marks all other clauses in TF_1 as soft clauses, and feeds the result to a pMAX-SAT solver. In this case, the solver would return two CoMSSs: $\{b_1 = a_3 + 1\}$ and $\{guard_2 = (y_1 < 5)\}$.

When analyzing the set of clauses in all the CoMSSs, the third step of the algorithm finds that there is one clause associated with a conditional statement c (the one at line 5, in this case). It thus identifies the branches corresponding to c (*i.e.,* branches $(5, 6)$ and $(5, 8)$ in the CFG), and checks whether one of the branches was not visited before. This is the case for branch $(5, 8)$, so the algorithm selects the unvisited branch as the branch to be expanded and returns to Step 1.

In the second iteration of the algorithm, the *Trace Generator* re-executes P with the same failing input, but forces the execution to follow branch $(5, 8)$ at conditional statement 5 [77], which results in a new trace: $\{1, 2, phi1, 5, 8, phi2, 9\}$. Given this trace, the algorithm first adds the newly covered program entities to the partial program (see Figure 9(c)) and then computes a new trace formula based on the expanded partial program. Since the execution of statement 8 instead of statement 6 is the only difference between this trace and the one generated in the previous iteration, the new trace formula is identical to TF_1, except for clause $(b_1 = a_3 + 1)$ (corresponding to statement 6), which is replaced by clause $(b_2 = a_3 + 2)$ (corresponding to statement 8). This trace formula, when conjoined with TF_1, would thus result in the trace shown below. (In practice, OFC simply conjoins the previous formula and the clause(s) corresponding to the new statement(s), which produces the same result, as explained in Section 6.2.2.1.)

$$
\begin{aligned}
TF_2 =& (guard_1 = (x_1 \geq 0)) \wedge (a_1 = x_1) \wedge \\
& ((guard_1 \wedge (a_3 = a_1)) \vee (\neg guard_1 \wedge (a_3 = a_2))) \wedge \\
& (guard_2 = (y_1 < 5)) \wedge (b_1 = a_3 + 1) \wedge \\
& ((guard_2 \wedge (b_3 = b_1)) \vee (\neg guard_2 \wedge (b_3 = b_2))) \wedge (b_2 = a_3 + 2)
\end{aligned}
$$

Similar to the first iteration, the algorithm then conjoins input conditions (TF_2) and failing assertion to obtain a new unsatisfiable formula, marks hard and soft clauses, and feeds the formula to the pMAX-SAT solver, which would return two CoMSSs: $\{b_1 = a_3 + 1\}$ and $\{guard_2 = (y_1 < 5), b_2 = a_3 + 2\}$.

Step 3 of the algorithm then checks whether any of these clauses is associated with a conditional statement *cs* and, if so, whether *cs* has any outgoing branches not yet visited by a trace. In this case, both of the branches corresponding to the first clause in the second CoMSS have been covered in our analysis. Therefore, the algorithm stops iterating and reports to developers these two CoMSSs, together with their corresponding program entities: {line 6} and {line 5, line 8}.

This would inform developers that suitably changing either (1) the statement at line 6 or (2) both the conditional statement at line 5 and the statement at line 8 could fix the program, so that input ($x_1 = 0,\ y_1 = 0$) would not violate the assertion at line 9. The clauses associated with the statements would provide additional information that could help understand the fault and find a fix—a (partial) mathematical explanation of how the statements contribute to the failure.

6.2.2.5 Further Considerations

Compared to BugAssist, OFC tends to generate a simpler formula that is as effective as one that encodes the whole program but less expensive to solve. Considering the example, for instance, OFC explored only 2 of the 4 paths in the program. Compared to an all-paths analysis, OFC included only *relevant* program entities in the trace formula: the assertion violation in the example is independent from the outcome of the predicate at statement 1,

and my algorithm successfully identifies the statement as irrelevant and avoids exploring both of its branches. However, although I expect the cost of finding solutions for a formula constructed by OFC to be lower than that of solving the formula generated by an all-paths analysis, OFC can perform a number of iterations when constructing the formula, and thus make multiple calls to the solver. Therefore, whether OFC is more efficient than an all-paths analysis depends on the number and cost of iterations it performs. I study this tradeoff in the empirical evaluation of OFC (see Section 6.3.2.2).

Compared to approaches that consider only the failing trace (*e.g.,* [26, 39]), OFC can conservatively identify all parts of the program that are relevant to the failure. In the example, if the algorithm had stopped after the first iteration, it would have missed the second CoMSS: $\{guard_2 = (y_1 < 5), b_2 = a_3 + 2\}$. That is, by reasoning about the original failing trace alone, developers could only infer that the fault may be related to the conditional statement at line 5. Conversely, by considering also the additional trace, OFC can discover that a fix involving that conditional statement should also consider possible changes to statement 8. OFC can thus make formula-based debugging more efficient without losing accuracy and effectiveness.

Compared to more traditional debugging techniques, OFC is likely to produce more accurate results. If I applied dynamic slicing to the example, for instance. A dynamic slice computed for the failing assertion at line 9 would include not only statements 5 and 6, which is correct, but also statements 1 and 2, which are irrelevant for the failure.

6.2.2.6 Additional Details and Optimizations

Handling Multiple Failing Inputs Although it is defined for a single failing input, OFC can take advantage of the presence of multiple failing tests *for the same fault*. Because, by definition, the faulty statement(s) should be executed by all failing tests and be responsible for all observed failures, OFC can handle multiple failing inputs as follows: (1) generate a report for each individual failing input, (2) identify the potential faulty entities (and corresponding clauses) that appear in all individual reports, and (3) report to developers only these entities, ranked based on the average of their original ranks in the individual

reports.

Loop Unrolling In the presence of loops, the size of a formula is in general unbounded. As it typical for symbolic analysis approaches (*e.g.,* [27,54,62]), in OFC I address this issue by performing loop unrolling [13]. One advantage of OFC over other all-paths analyses is that it can decide how many times to unroll a given loop based on concrete executions, rather than on some arbitrary threshold. Nevertheless, for practicality reason, OFC still needs to define an upper bound for loop unrolling, to limit the overall size of trace formulas.

Dynamic Symbolic Execution OFC, like dynamic symbolic execution [42, 74], may replace symbolic variables in the trace formula with their corresponding concrete values, so as to allow the solver to handle formulas that go beyond its theories (*e.g.,* non-linear expressions, dynamic memory accesses). Doing so makes the approach more practical, but can introduce unsoundness (in the form of discrepancies between the actual semantics of the program and the semantics encoded in the formula) and reduce the number of possible solutions the solver can compute. This can result in both false positives—program entities that, even if suitably changed, could not eliminate the failure at hand—and false negatives—solutions that do not include the faulty statement(s).

Solution Space Pruning Because the number of CoMSSs for a given MAX-SAT problem may be too large and affect the ability of OFC to enumerate and analyze all solutions in a reasonable amount of time, OFC allows developers to specify an upper bound for the number of clauses in a CoMSS (*i.e.,* the number of statements reported *together as a single fault*) and terminates the search for new solutions when the solver starts reporting CoMSSs that exceed this bound. The rationale is that a potential bug generally involves a limited number of statements, whereas a CoMSS that contains a large number of clauses suggests a large semantic change in the program (which may be able to eliminate a failure but is usually not an ideal fix).

6.3 Preliminary Evaluation

To evaluate CW and OFC, I have developed a prototype tool for C programs that implements four different formula-based debugging techniques: BugAssist (BA), BugAssist with clause weighting (BA+CW), on-demand formula computation (OFC), and on-demand formula computation with clause weighting (OFC+CW). I have then empirically investigated the following research questions:

- **RQ1:** Does BA+CW produce more accurate results than BA? If so, what is CW's effect on efficiency?
- **RQ2:** Does OFC improve the efficiency of an all-paths formula-based debugging technique?
- **RQ3:** Does OFC+CW combine the benefits of OFC and CW? If so, can it handle programs that an all-paths technique could not handle?
- **RQ4:** Can OFC+CW scale to larger programs than an all-paths technique? If not, what are the major limitations?

I now discuss my evaluation setup and the results of the study.

6.3.1 Evaluation Setup

6.3.1.1 Implementation

I implemented OFC, as presented in Section 6.2.2, in Java and C. My tool leverages the LLVM compiler infrastructure (`http://llvm.org/`) to transform programs into SSA form and add instrumentation that (1) dumps dynamic traces and concrete program states and (2) performs execution hijacking [77] to force the program along specific branches in the *Trace Generator*. I implemented BA as a version of OFC that builds a formula for all (bounded) paths in the program instead of operating on demand. I implemented Ochiai [11], the statistical fault localization technique that I use for CW, as a Java program that operates directly on the dumped dynamic traces. Finally, to handle wpMAX-SAT and pMAX-SAT

problems, I implemented interfaces to invoke the Yices SMT solver [36] and the Z3 theorem prover [7].

Implementing the OFC algorithm, and in particular the *Trace Generator* and the *Formula Generator* components, is extremely challenging both from the conceptual and the engineering standpoint [27]. To avoid spending too much development effort, I decided to build a prototype that implements OFC completely, but has some limitations when handling some constructs of the C language related to heap memory management.

6.3.1.2 Benchmarks

For the evaluation, I selected two programs and faults from the SIR repository [2] and a real fault from the Redis open-source project [8]. The first two benchmarks, from SIR, consist of multiple (faulty) versions of two programs: tcas (41 versions, ˜200 LOC) and tot_info (11 versions, ˜500 LOC). These programs also come with test cases and a golden (supposedly fault-free) version that can be used as an oracle. I selected these programs for two main reasons. First, tcas is an ideal subject for my evaluation because it allows me to find all possible solutions of the program formulas considered and thus precisely compute the savings that OFC achieves in terms of complexity reduction. (This is in general impossible for larger, more complex programs.) Second, these two programs were also used to evaluate BugAssist [54], which lets me directly compare my results with those of a state-of-the-art all-paths formula-based technique in terms of accuracy and efficiency. The third benchmark I considered is a faulty module of Redis, a widely used in-memory key-value database, which also comes with a set of test cases. To address the last research question, RQ4, I also ran my tool on larger benchmarks in the SIR repository.

6.3.1.3 Study Protocol

For each faulty program version considered, I proceeded as follows. *First*, I identified passing and failing test cases for that version. For tcas and tot_info, I did so by defining the assertion for a test using the output generated by the same test when run against the golden implementation. For the bug in Redis, I used the bug description [8] and the corresponding test [9]. I then ran all programs instrumented to collect coverage information

for all passing and failing tests at the same time. I used this coverage information to compute the suspiciousness values for the branches and statements in each program version using the Ochiai metric [11]. These are the values that BA+CW and OFC+CW use to assign weights to the clauses in the program formula. *Second*, for each failing input, I ran all four techniques on the faulty version. Because the all-paths techniques timed out or could not build a formula for the bugs in tot_info and Redis (see Section 6.3.2.3 for details), I could only investigate RQ1 and RQ2 on tcas, whereas I used tcas, tot_info, and Redis for RQ3. (For fairness, I note that Reference [54] reports results for 2 versions of tot_info. However, the authors mention that those results were obtained working on a program slice, and there are no details on how the slice was computed and on which version, so I could not replicate them using either our or their implementation of BA.) For each faulty version and for each technique that successfully ran on it, the technique generated a report for the developers. To do a complete assessment of the performance of the techniques, I also recorded the average CPU time of each technique for each failing input, the number of iterations of the OFC algorithm, whether the generated report contained the fault, and, if so, the rank of the fault in the report.

6.3.2 Results and Discussion

6.3.2.1 RQ1—BA+CW Versus BA

To answer this research question, I compared the accuracy and the computational cost of BA+CW and BA to evaluate the impact of leveraging information from statistical fault localization. To do so, I ran both techniques on the 41 faulty versions of tcas and computed the results as described in Section 6.3.1. Table 13 presents these results. The columns in the table show the version ID, the number of lines of code a developer would have to examine before getting to the fault, and the average CPU time consumed by BA and BA+CW to compute their results. For comparison purposes, in the last column I also report the results of a traditional fault-localization technique (Ochiai). For tcas.v3, for instance, it took 292 seconds (BA) and 183 seconds (BA+CW) to generate the results, and developers would have to examine 8.5 lines of code (BA), 1 line of code (BA+CW), or 3 lines of code (Ochiai).

Table 13: Results for BA and BA+CW when run on tcas.

Version	BA		BA+CW		Ochiai	Version	BA		BA+CW		Ochiai
	rank	time	rank	time	rank		rank	time	rank	time	rank
v1	7.5	26s	2	27s	4	v22	4	7s	5	7s	22
v2	4	15s	4	16s	3	v23	5.5	15s	10	12s	23
v3	8.5	292s	1	183s	3	v24	7.5	30s	8	23s	23
v4	8	11s	3	11s	1	v25	5.5	297s	4	216s	2
v5	7.5	352s	3	323s	18	v26	8	160s	5	123s	21
v6	7.5	569s	5	316s	4	v27	9.5	443s	4	393s	21
v7	8	484s	8	238s	8	v28	5	41s	3	40s	2
v8	7.5	21s	13	18s	48	v29	5	25s	1	27s	20
v9	4.5	18s	10	15s	23	v30	5	11s	6	14s	20
v10	8	125s	3	96s	4	v31	8.5	958s	2	909s	4
v11	5.5	130s	1	91s	21	v32	8.5	171s	1	145s	3
v12	8	22s	11	20s	49	v33	6	79s	1	70s	3
v13	8	24s	7	21s	1	v34	7.5	164s	5	144s	23
v14	7	28s	1	28s	1	v35	5	38s	3	40s	2
v15	6.5	14s	5	14s	21	v36	2.5	19s	1	17s	1
v16	8	331s	12	228s	49	v37	7.5	127s	1	136s	3
v17	8	626s	8	285s	49	v38	6.5	8s	1	8s	2
v18	6	378s	6	245s	49	v39	6	244s	4	272s	2
v19	8	399s	5	167s	49	v40	5.5	219s	3	219s	4
v20	8	504s	8	247s	21	v41	7.5	6s	2	5s	6
v21	7.5	252s	8	194s	21	Average	6.5	187s	4.7	137s	17

Note that, for BA+CW, the number of lines of code to examine corresponds to the actual rank of the faulty line of code in the report produced by the technique. BA, however, does not rank the potentially faulty lines of code, but simply reports them as an unordered set to developers. Therefore, the number in the table corresponds to the number of lines of code developers would have to investigate if I assume they examine the entities in the set in a random order (*i.e.,* half of the size of the set).

As the results in Table 13 show, both techniques were able to identify the faulty statements for all versions considered. I can also observe that both BA and BA+CW produced overall more accurate results than Ochiai (significance level of 0.05 for both BA and BA+CW for *a paired t-test*). Although this was not a goal of the study, it provides evidence that formula-based techniques, by reasoning on the semantics of a failing execution, can provide more accurate results than a purely statistical approach. As for the comparison of BA and BA+CW, BA+CW produced better results than BA, with a significance level of 0.05 for *a paired t-test*. On average, a developer would have to examine 4.7 statements

per fault for BA+CW versus 6.5 for BA. By leveraging the suspiciousness values computed by statistical fault localization, BA+CW can thus outperform BA in most cases (33 out of 41). For the 8 cases in which BA+CW did not outperform BA, manual analysis of the results identified one main reason. In some cases, the weights computed by fault localization were too inaccurate and caused the solver to first produce CoMSSs that did not include the actual faulty statements. Despite these negative cases, the overall performance of BA+CW is remarkable and justifies the use of statistical fault-localization information. BA+CW ranked the faulty statement first for 9 out of 41 versions, among the top 3 statements in another 8 cases, and at a position greater than 10 in only 3 cases.

The data in Table 13 also allow me to investigate the second part of RQ1, that is, the effect of CW on efficiency. As I discussed in Section 6.2.1, solving wpMAX-SAT problems may be computationally more expensive than solving a pMAX-SAT problem, so the use of CW may negatively affect the efficiency of formula-based debugging. As the table shows, on average BA+CW performs significantly better than BA (137s versus 187s, significance level of 0.05). Although these results may seem counterintuitive, I discovered that the extra information provided by the weights can in many cases unintentionally help the solver find CoMSSs more efficiently.

Answer to RQ1: *In summary, my results provide initial evidence that CW can improve formula-based debugging, both in terms of accuracy and in terms of efficiency.*

6.3.2.2 RQ2—OFC Versus BA

To investigate RQ2, I compared OFC and BA in terms of efficiency. As I did for RQ1, I ran the two techniques on the 41 faulty versions of `tcas` and measured their performance. The results are shown in Table 14. The table shows the version ID, the average CPU time spent by BA and OFC, respectively, on each failing input, the average number of iterations (*i.e.,* path expansions) of the OFC algorithm, and the average CPU time spent by OFC in each iteration. For example, for a failing input in `tcas.v1`, it took, on average, 26 seconds (BA) and 7 seconds (OFC) to generate the results, OFC iterated 9 times, and, for each expansion, it took OFC 0.8 seconds to find all CoMSS solutions.

Table 14: Performance results for BA and OFC on tcas

Version	BA	OFC	*#Iteration*	*Time per iteration*	Version	BA	OFC	*#Iteration*	*Time per iteration*
v1	26s	7s	9	0.8s	v22	7s	6s	13.2	0.4s
v2	15s	38s	12	3.2s	v23	15s	24s	11	2.1s
v3	292s	19s	14	1.4s	v24	30s	7s	10	0.7s
v4	11s	6s	9.2	0.6s	v25	297s	244s	12	20.3s
v5	352s	15s	13.4	1.1s	v26	160s	17s	13	1.3s
v6	569s	17s	13	1.3s	v27	443s	15s	13.4	1.1s
v7	484s	104s	14.8	7.1s	v28	41s	24s	11.2	2.2s
v8	21s	5s	10	0.5s	v29	25s	6s	9.8	0.6s
v9	18s	28s	12	2.4s	v30	11s	24s	11	2.2s
v10	125s	22s	14	1.6s	v31	958s	33s	10.8	3s
v11	130s	11s	8.4	1.3s	v32	171s	14s	13	1.1s
v12	22s	17s	14.2	1.2s	v33	79s	178s	13	13.7s
v13	24s	15s	13.3	1.2s	v34	164s	21s	13	1.6s
v14	28s	20s	13.8	1.4s	v35	38s	22s	14	1.5s
v15	14s	20s	13.2	1.5s	v36	19s	11s	11.2	1s
v16	331s	16s	13	1.2s	v37	127s	251s	14	18s
v17	626s	73s	14.2	5.1s	v38	8s	95s	16	5.9s
v18	378s	96s	13.4	7.2s	v39	244s	213s	12	17.8s
v19	399s	17s	13.2	1.3s	v40	219s	180s	10.4	17.3s
v20	504s	7s	9.4	0.8s	v41	6s	5s	8.2	0.6s
v21	252s	6s	8.8	0.7s	Average	187s	48s	12	4s

Overall, OFC was more efficient than BA in 33 out of 41 cases by looking at the second and third columns and could achieve almost 4X speed-ups on average (48 versus 187 seconds, significance level of 0.05) and over 70X speed-ups in the best case (504 versus 7 seconds).

To understand the reason of OFC outperforming BA in terms of efficiency, I also present the average number of iterations and time spent in each iteration in OFC. The second and fifth columns in the table clearly show that it took considerably less time for the pMAX-SAT solver to find solutions for formulas generated in one iteration of OFC (4 seconds) than for formulas generated by BA (187 seconds). The statistically significant gain of efficiency (significance level of 0.05) is caused, as expected, by the difference in the complexity of the encoded formulas—OFC only encodes the subset of the program relevant to the failure into the formulas passed to the solver, while BA generates a much more complex formula that encodes the semantics of the entire program.

The results in the fourth column of Table 14 indicate that OFC performed 12 iterations per fault, on average. Therefore, as I discussed in Section 6.2.2.5, the benefits of generating a simpler formula were in some cases (*e.g.,* tcas.v2) outweighed by the cost of solving multiple

Table 15: Average time for processing tcas faults

BA	BA+CW	OFC	OFC+CW
187s	137s	48s	36s

pMAX-SAT problems during on-demand expansion, thus making OFC less efficient than BA. In fact, comparing the results in the second and third columns of the table, I can observe that there were 8 cases in which OFC performed worse than BA.

It is also worth noting that my results on the number of iterations performed by OFC provide some evidence that techniques that operate on a single-trace formula (*e.g.,* [26, 39]) may compute inaccurate results, even when they encode both data- and control-flow information. Because each expansion adds new constraints that must be taken into account in the analysis, limiting the analysis to a single trace is likely to negatively affect the quality of the results.

Finally, as a sanity check, I examined the sets of suspicious entities reported by the two techniques. This examination confirmed that OFC reports the same sets as BA (*i.e.,* the fault-ranking results for OFC were the same as those for BA, shown in Table 13). That is, it confirmed that OFC is able to build smaller yet conservative formulas and can thus produce the same result as an approach that encodes the whole program.

Answer to RQ2: *In summary, my results for RQ2 provide initial, but clear evidence that OFC can considerably improve the efficiency of formula-based debugging without losing effectiveness with respect to an all-paths technique such as BugAssist.*

6.3.2.3 RQ3—OFC+CW Versus BA, BA+CW, and OFC

To answer the first part of RQ3, I compared the performance of OFC+CW with that of the other three techniques considered, in terms of both accuracy and efficiency, when run on the 41 tcas versions. For accuracy, I found that the results for OFC+CW, not reported here for brevity, were the same as those listed in the "BA+CW" column of Table 13. This is not surprising, as OFC reports the same sets as BA, as I just discussed, and I expect CW to benefit both techniques in the same way. Therefore, the results show that OFC+CW is as accurate as BA+CW and more accurate than BA and OFC.

To compare the efficiency of the four techniques considered, I measured the average CPU time required by the techniques to process one fault in tcas, shown in Table 15. As the table shows, for the cases considered, combining OFC and CW can further reduce the cost of formula-based debugging by 25% with respect to OFC and by over 80% with respect to my baseline, BA. (Note that, to assess whether my results depended on the use of a specific solver, I replaced Yices with Z3 [7] and recomputed the results in Table 15. I obtained comparable results also with this alternative solver.) Although these are considerable improvements, it is unclear whether they can actually result in more practically applicable debugging techniques. This is the focus of the second part of RQ3, which aims to assess the potential increase in applicability that my two improvements can provide. To answer this part of RQ3, I ran the techniques considered on my two additional benchmarks: tot_info and Redis.

tot_info results for RQ3 Unlike tcas, tot_info contains loops, calls to external libraries, and complex floating point computations. (I considered all faults except those directly related to calls to external *system* libraries, which my current implementation does not handle.) Because of the presence of loops, I set an upper bound of 5 to the size of clauses in a CoMSS. (I believe 5 is a reasonable value, as it means that the technique would be able to handle all faults that involve up to 5 statements.) As I discussed in Section 6.3.1, for BA and BA+CW the program formula generated was too large, and the solver was not able to compute the set of CoMSSs within two hours (the time limit I used for the study) for the faults considered. Conversely, OFC and OFC+CW were able to compute a result within the time limit for all faults, which provides initial evidence that my improvements can indeed result in more scalable formula-based techniques. By focusing only on the relevant parts of a failing program and leveraging statistical fault localization, OFC+CW can reduce the complexity of the analysis and successfully diagnose faults that an all-paths technique may not be able to handle. To also assess the accuracy of the produced results, in Table 16 I show the results computed by OFC+CW. The columns in the table show the program version and the number of lines of code a developer would have to examine before getting

Table 16: Ranking results of OFC+CW on tot_info

Version	OFC+CW
tot_info.v1	2
tot_info.v3	1
tot_info.v4	1
tot_info.v11	3
tot_info.v14	1
tot_info.v15	1
tot_info.v16	2
tot_info.v18	3
tot_info.v20	3
tot_info.v22	6
tot_info.v23	8

Table 17: Results for OFC+CW when run on Redis's bug

Rank	Source Location	Statement
1	scripting.c:237	if (obj_s == NULL) break;
2	scripting.c:236	obj_s = (char*)lua_tolstring(lua,j+1,&obj_len);
3	scripting.c:232	j++
4	scripting.c:206	int j, argc = lua_gettop(lua);
5	scripting.c:218	if (argc == 0)

to the fault in that version. As the table shows, OFC+CW was able to rank all 11 faults within the top 10 statements in the list reported to the developer, and 4 of them at the top of the list.

Redis results for RQ3 To further assess the practical applicability of OFC+CW, I run the techniques considered on my third benchmark, a real-world bug [8] in Redis, which is considerably larger and more complex than tcas and tot_info. The bug is a potential buffer overflow in a module of Redis that processes Lua scripts (www.lua.org/) and consists of 1 KLOC). Figure 10 shows an excerpt of the bug. The original version of the code fails to check whether the size of the script from the command line is greater than the size of the memory in which it is stored. If the script is too large, the program generates an out-of-boundary memory access and fails.

I inserted assertions that are triggered when a buffer overflow occurs, and applied OFC+CW to the faulty code. My tool generated the report shown in Table 17, which

```
203 #define LUA_CMD_OBJCACHE_SIZE 32
...
206 int j, argc = lua_gettop(lua);
...
214 static robj *cached_objects[LUA_CMD_OBJCACHE_SIZE];
...
218 if (argc == 0)
...
221  return 1;
222
...
232  for (j = 0; j < argc; j++) {
233    char *obj_s;
234    size_t obj_len;
236    obj_s = (char*)lua_tolstring(lua,j+1,&obj_len);
237    if (obj_s == NULL) break; /* Not a string. */
    /* Try to use a cached object. */
    /* bug fixes */
240-   if (cached_objects[j] && cached_objects_len[j] >= obj_len) {
240+   if (j < LUA_CMD_OBJCACHE_SIZE && cached_objects[j] &&
241+     cached_objects_len[j] >= obj_len) {
...
```

Figure 10: Excerpt code of the bug in Redis

contains five suspicious statements and program locations. The first entry in my report suggests that a control statement should be changed after line 237 of scripting.c to avoid the out-of-boundary access in the next statement. This is also the location where the developers of Redis fixed the bug [8]. Also in this case, I tried to run the all-paths techniques on the module, but they were not successful. Because BA relies on a static model checker that unrolls loops based on a predetermined (low) bound, whereas the loop in the code needs to be executed a large number of times for the bug to be triggered, BA is unable to build a formula for the failure at hand. Unfortunately, increasing the number of times loops are unrolled is not a viable solution, as it causes the number of encoded paths to explode and results in the solver timing out.

Although this is just one bug in one program, and I cannot claim generality of the results, I find the results very encouraging. They provide evidence that my approach could make formula-based debugging applicable to real-world programs and faults.

Answer to RQ3: *In summary, the results for RQ3 provide initial evidence that the combination of OFC and CW can make formula-based debugging more applicable.*

6.3.2.4 RQ4—Scalability

The previous three research questions show that my optimizations, OFC and CW, can produce more accurate results than BA, and more efficiently. The benchmark programs I used in my study, however, all consist of less than 1000 LOC—including Redis's module. In RQ4, I therefore investigate whether my optimizations can enable formula-based debugging to scale to larger programs.

To answer this research question, I applied OFC and CW to a set of larger programs and faults in the SIR repository [2], namely, `grep` (10K LOC), `sed` (14K LOC), and `gzip` (5K LOC). Unfortunately, OFC+CW timed out after at most a few iterations when Yices was used as the MAX-SAT solver, and I obtained similar results when replacing Yices with Z3. In these cases, the technique produced either no results, when the solver timed out at the first iteration, or extremely inaccurate results when the solver timed out after a few iterations. These results indicate that, even if OFC and CW can considerably improve the efficiency of formula-based debugging, these approaches are still too expensive to scale to larger programs.

To better understand the reasons for this limited scalability, I performed an additional study, in which I investigated my intuition that solving MAX-SAT (or wpMAX-SAT) problems accounts for the majority of the cost of formula-based debugging. To do so, I measured the percentage of time spent on solving MAX-SAT problems when I applied OFC to `tcas`. In the study, I used both Yices and Z3 as MAX-SAT solvers to assess whether my results depended on the use of a specific solver. (Because Z3 currently does not support wpMAX-SAT, I did not consider CW in this study.)

Table 18 shows the results of the study. In the table I provide, for each version of `tcas` and for each of the two solvers considered, the total CPU time used by OFC ("Total Yices" and "Total Z3") and the percentage of CPU time spent in the solver ("Yices %" and "Z3 %"). The results in the table confirm my hypothesis: the percentage of CPU time spent in the solver is over 70% in a majority of cases and is 75% and 72% on average when using Yices and Z3, respectively. Although these numbers confirm my intuition, there is a far more important finding that emerges from the data: the existence of a clear correlation

Table 18: Percentage of CPU time spent in the solvers when running OFC on tcas

Version	Total Yices	Yices %	Total Z3	Z3 %	Version	Total Yices	Yices %	Total Z3	Z3 %
v1	7s	58%	8s	59%	v22	6s	49%	6s	55%
v2	38s	93%	20s	87%	v23	24s	75%	19s	68%
v3	19s	81%	16s	78%	v24	7s	52%	8s	58%
v4	6s	54%	6s	59%	v25	244s	85%	98s	61%
v5	15s	75%	13s	71%	v26	17s	68%	16s	66%
v6	17s	79%	17s	78%	v27	15s	73%	14s	70%
v7	104s	100%	31s	99%	v28	24s	79%	19s	72%
v8	5s	54%	6s	60%	v29	6s	48%	7s	52%
v9	28s	87%	20s	83%	v30	24s	76%	21s	72%
v10	22s	80%	19s	77%	v31	33s	78%	22s	68%
v11	11s	70%	11s	70%	v32	14s	61%	14s	61%
v12	17s	78%	14s	72%	v33	178s	87%	73s	69%
v13	15s	73%	15s	72%	v34	21s	76%	19s	75%
v14	20s	78%	17s	74%	v35	22s	83%	16s	76%
v15	20s	80%	17s	77%	v36	11s	74%	11s	73%
v16	16s	78%	14s	75%	v37	251s	97%	90s	92%
v17	73s	99%	24s	98%	v38	95s	93%	104s	94%
v18	96s	80%	44s	57%	v39	213s	94%	78s	84%
v19	17s	64%	16s	63%	v40	180s	94%	68s	84%
v20	7s	52%	8s	58%	v41	5s	54%	6s	65%
v21	6s	49%	7s	54%	Average		75%		72%

between the total CPU time and the percentage of CPU time spent in the solvers.

To better illustrate this point, Figure 11 shows such correlation between total CPU time (x-axes) and percentage of CPU time spent in the solvers (y-axes). Looking at the figure, I can observe that, modulo some outliers, the longer the execution, the higher the percentage of CPU time used by the solvers. (This is confirmed by the Spearman's rank correlation coefficient, which is 0.91 (very strong correlation) for OFC with Yices and 0.62 (moderate-to-strong correlation) for OFC with Z3.) This correlation suggests that the cost of constraint solving becomes increasingly relevant as the complexity of the debugging problem increases (as indicated by the longest overall time needed to solve it). The cost of MAX-SAT solving is thus a bottleneck that limits the scalability of the approach. My optimizations are a first step towards reducing that cost, as they tend to build much smaller formulas than an all-paths approach (which compensates for the fact that the solvers are called multiple times, as the results in Section 6.3.2 show). Nevertheless, further and more aggressive reductions of the size of the MAX-SAT problems generated by the approach (and possibly optimizations of the solvers themselves) could be a future step to make formula-based debugging really

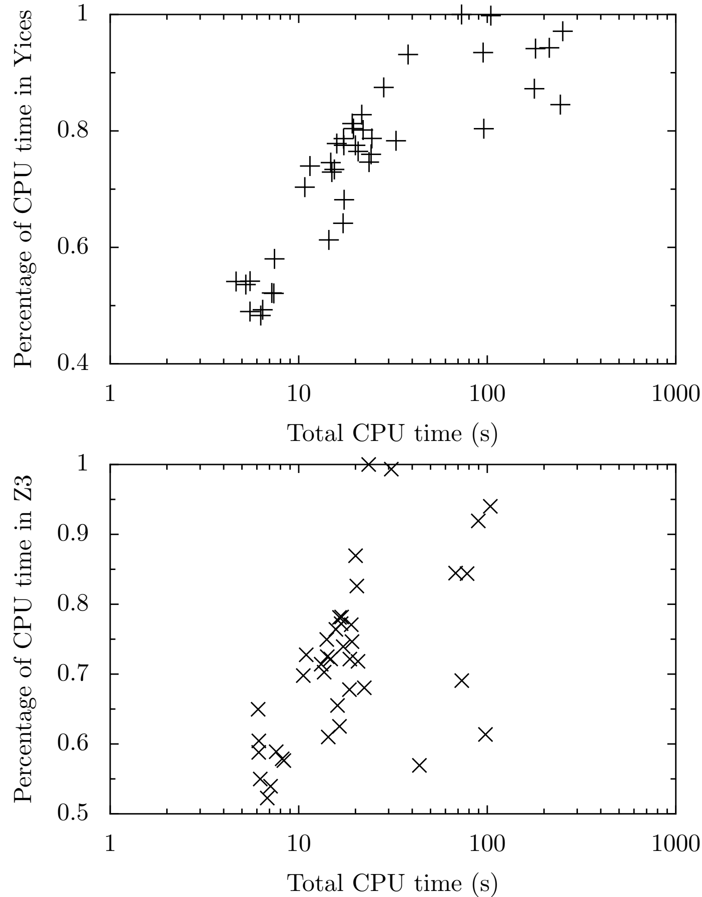

Figure 11: Correlation between the percentage of CPU time spent in the solvers and the total CPU time for our OFC technique when applied to each failing input in `tcas`

scalable and applicable in practice.

> ***Answer to RQ4:*** *Unfortunately, OFC+CW failed to scale to larger programs and bugs, and the main bottleneck lies in the MAX-SAT solver as indicated in the additional study.*

6.3.3 Threats to Validity

In addition to the usual internal and external validity threats, a specific one is that I re-implemented BugAssist, one of the techniques against which I compare. Unfortunately, I were not able to use the command-line implementation of BugAssist from its authors, and the Eclipse plugin was problematic to use in a programmatic way. Moreover, the tool's source code was not available, which I needed to integrate CW and BugAssist for my evaluation.

Another threat to validity is that I performed the evaluation only on a few simple programs and a module of a real open-source project. Therefore, my results may not generalize. However, my main goal was to investigate whether CW and OFC could improve the state of the art in formula-based debugging, so using the same programs used in related work allowed us to directly compare my techniques with such work.

6.4 Conclusion

In this chapter, I presented clause weighting (CW) and on-demand formula computation (OFC), two ways to improve existing formula-based debugging techniques and mitigate some of their limitations. CW incorporates the (previously ignored) information provided by passing test cases into formula-based debugging techniques to improve their accuracy. OFC is a novel formula-based debugging algorithm that, by operating on demand, can analyze a small fraction of a faulty program and yet compute the same results that would be computed analyzing all paths of the program, at a much higher cost.

To evaluate CW and OFC, I performed an empirical study. In the study, I assessed the improvements that CW and OFC can achieve over a state-of-the-art formula-based debugging approach. My results show that formula-based debugging remains an expensive approach with limited scalability (mostly because of the cost of solving MAX-SAT problems). Nevertheless, CW and OFC can considerably improve the accuracy and efficiency of this approach and therefore represent a first step towards making formula-based debugging more practically applicable and motivate further research in this area.

CHAPTER VII

RELATED WORK

Debugging is an extremely prolific area of research, and the related work is consequently vast. In this section, I will focus on the work that is most closely related to approaches in my thesis proposal.

7.1 Techniques for Reproducing Field Failures

BugRedux is related to automated test-input generation techniques, such as those based on symbolic execution (*e.g.,* [21, 42, 74, 78]) and random generation (*e.g.,* [65, 66]). Generally, these techniques aim to generate inputs to discover faults, and they are not directly applicable to the problem I am targeting, as I have discussed in Section 4.2.2.

Techniques that capture program behaviors by monitoring or sampling field executions are also related to BugRedux (*e.g.,* [4, 28, 44, 58]). These techniques usually record execution events and possibly interactions between programs and the running environment to later replay or analyze them in house. These approaches tend to either capture too much information, and thus raise practicality and privacy issues, or too little information, and thus be ineffective in my context.

More recently, researchers started investigating approaches to reproduce field failures using more limited information. For example, some researchers used weakest preconditions to find inputs that can trigger certain types of exceptions in Java programs [23,40,64]. These approaches, however, target only certain types of exceptions and tend to operate locally at the module level. Another approach, SherLog [80], makes use of run-time logs to reconstruct paths near logging statements to help developers to identify bugs. LogEnhancer [81], a followup work, improves the diagnose ability of SherLog by adding more information to log messages. These approaches differ from BugRedux because they do not aim to generate program inputs, but rather to highlight code areas potentially related to a failure. Artzi and colleagues present ReCrash [16], a technique that records partial object states at method

levels dynamically to recreate an observed crash at different levels of stack depth. Although this approach can help recreate a crash, the recreated crash can be very different from the original one: if the stack depth is low, the information is in most cases too local to help (*e.g.,* a null value passed as a parameter); conversely, if the stack depth is high, the technique may have to collect considerable amounts of program state, which can make it impractical and raise privacy issues. Similar to ReCrash in terms of using crash call stacks as the starting point, RECORE [72] is an approach that applies genetic algorithms to synthesize executions from crash call stacks. However, the current empirical evaluation of RECORE shows that it mainly focuses on partial or shallow executions (*i.e.,* executions of standalone library classes), so it is unclear whether the approach would be able to reproduce complete executions systematically. Failures in library classes usually result in shallow crash stacks, and in my experience BugRedux should work quite well in these cases.

The two approaches most related to BugRedux are ESD, by Zamfir and Candea [82] and the technique by Crameri, Bianchini, and Zwaenepoel [32]. ESD is a technique for automated debugging based on input generation. Given a POF, ESD uses symbolic execution to try to generate inputs that would reach the POF. As I showed in Chapter 4, without additional guidance, symbolic execution techniques are unlikely to be successful in this context. In fact, as I stated in Section 4.2.2, the programs and failures used in Reference [82] can also be recreated through unguided symbolic execution. Unlike BugRedux, however, one of the strengths of ESD is that it can recreate concurrency-related failures. With this respect, BugRedux and ESD are complementary, and it would be interesting to investigate a combination of the two techniques. (Another approach that aim to reproduce concurrency bugs, and is thus also complementary to BugRedux, is PRES, by Park and colleagues [67].) The approach by Crameri and colleagues improves ESD by using partial branch traces—where the relevant branches are identified through a combination of static and dynamic analysis—to guide symbolic execution for field failures reproduction. Although their approach can reduce dramatically the number of branches considered, I found in my experience that the use of branch-level traces can be problematic. Their empirical evaluation is also performed on programs whose failures can be reproduced using unguided

symbolic execution. It is therefore unclear whether their approach would work on larger programs and harder-to-reproduce failures.

It is nowadays common practice to use software (*e.g.,* Breakpad [1]) or OS capabilities (*e.g.,* Windows Error Reporting [6] and Mac OS Crash Reporter [3]) to automatically collect crash reports from the field. As I discussed in Introduction, these reports can be used to correlate different failures reported from the field. DebugAdvisor [17], for instance, is a tool that analyzes crash reports to help find a solution to the reported problem by identifying developers, code, and other known bugs that may be correlated to the report. Although these techniques have been shown to be useful, they target a different problem, and the information they collect is too limited to allow for recreating field failures.

7.2 Fault Localization Techniques

Statistical fault-localization techniques are a set of techniques that identify suspicious statements by leveraging a large number of passing and failing executions. F^3 and CW leverage this type of techniques. These techniques share a similar intuition—entities that executed mostly in failing executions are more likely the potential causes of failures. Tarantula [53], uses statement coverage as criteria to compute suspiciousness values and rank them. CBI and followup work from Liblit and colleagues [57, 58] use predicate outcomes, instead of statements, as criteria. Moreover, they operate on data collected from the field, rather than in-house. In recent years, many different statistical fault-localization approaches were proposed that operate on different entities and/or use different statistical analyses to compute suspiciousness values [11, 12, 15, 60, 73, 87]. Among these techniques, Ochiai [11] has shown to have a good performance compared to other similar techniques in several empirical studies, and recent research has shown, both empirically and analytically, that even better techniques can be defined [63]. Therefore, I decided to choose them as baseline techniques in the empirical evaluation. Also recently, Baah and colleagues defined a technique [19] that accounts for the confounding bias due to program dependences when computing suspiciousness values and can improve the effectiveness of fault localization. Usually, statistical fault-localization techniques are limited by the fact of requiring a high quality test suite

containing a large number of passing and failing inputs. F^3 leverages and extends these techniques (Ochiai and OBM, specifically) and addresses their limitations by generating the needed inputs.

Other researchers defined approaches that extend traditional fault localization techniques and/or try to address their limitations. To enable fault localization in the absence of multiple failing test cases, BugEx [70] uses an evolutionary approach that (1) aims to generate executions almost identical to a single failing test and (2) uses these executions to identify program facts that are relevant for the failure. Although the technique is potentially effective, generating executions that have minimal differences with one another (*e.g.,* only one branch) is extremely challenging, which limits the practical applicability of the approach. Artzi and colleagues use a specialized dynamic symbolic execution approach to (1) discover faults in web applications, (2) generate passing and failing executions similar to the executions that revealed the faults, and (3) use these executions to try to localize the faults [14]. Their technique is not meant to help the debugging of field failures and is specialized for web applications and the faults in such applications (*e.g.,* malformed HTML). Groce [45] propose an idea similar to filtering optimization in F^3. Similar to the filtering optimization, the effectiveness of their filters highly depends on the quality of the available test suite.

Another family of approaches to fault localization is experimental debugging. Among these techniques, one of the most popular technique is delta debugging [31], which can simplify and isolate failure causes through a differential analysis of inputs [85], code [83], or program states [84]. More recently, researchers have defined related techniques that perform fault localization by modifying the state of a program in several points during a failing execution and assessing whether the state change prevented the failure from occurring; if so, the corresponding point in the code is reported as potentially faulty (*e.g.,* [49, 86]). The main limitation of these approaches is that they are unsound, as the manipulation of program states (and executions in general) can result in infeasible behavior and can thus produce false positives.

7.3 *Formula-based Fault Localization Techniques*

My techniques in Chapter 6 are most closely related to formula-based debugging techniques, an increasingly popular research area. In particular, OFC builds on BugAssist [54], which encodes a faulty program as an unsatisfiable Boolean formula, uses a MAX-SAT solver to find maximal sets of satisfiable clauses in this formula, and reports the complement sets of clauses as potential causes of the error. The dual of MAX-SAT, that is the problem of computing minimal unsatisfiable subsets (or unsatisfiable cores), can also be leveraged in a similar way to identify potentially faulty statements, as done by Torlak, Vaziri, and Dolby [76]. This kind of techniques have the advantage of performing debugging in a principled way, but tend to rely on exhaustive exploration of (a bounded version of) the program state, which can dramatically limit their scalability. OFC, by operating on demand, can produce results that are at least as good as those produced by these techniques at a fraction of the cost. Moreover, by working on a single path at a time, OFC can directly benefit from various dynamic optimizations. Finally, CW leverages the additional information provided by passing test cases, which are not considered by most existing techniques in this arena.

Another related approach, called Error Invariants, transforms program entities on a single failing execution into a path formula [39]. This technique leverages Craig interpolants to find the points in the failing trace where the state is modified in a way that affects the final outcome of the execution. The statements in these points are then reported as potential causes of the failure. This technique cannot handle control-flow related faults because, as also recognized by the authors, it does not encode control-flow information in its formula. To address this limitation, in followup work the authors developed a version of their approach that encodes partial control-flow information into the path formula [26]; with this extension, their approach can identify conditional statements that may be the cause of a failure. However, compared to OFC's approach of suitably encoding SSA's ϕ functions, their approach generates much more complex preconditions, that is, conjunctions of all predicates that a statement is control dependent on. Conversely, my algorithm only needs to encode the predicate that the ϕ function is directly dependent on. In addition, the two approaches handle potentially faulty conditional statements very differently. OFC considers

additional paths induced by a possible modification of the faulty conditionals, and can therefore identify additional faulty statements along these paths. Their technique simply reports the identified conditionals to developers, who may miss important information and produce a partial, if not incorrect, fix (see Section 6.2.2.5).

Finally, automated repair techniques (*e.g.,* [34,48,56]) are related to my work. However, these techniques are mostly orthogonal to fault-localization approaches, as they require some form of fault localization as a starting point. (One exception is Angelic Debugging, by Chandra and colleagues [24], which combines fault localization and a limited form of repair.) In this sense, I believe that the information produced by my approach could be used to guide the automated repair generation performed by these techniques, which is something that I plan to investigate in future work.

CHAPTER VIII

CONCLUSION AND OPEN PROBLEMS

Reproducing and debugging field failures are notoriously difficult tasks because of the increasing complexity of modern software systems and the limited availability of information from crash reports generated on users' machines. To address and mitigate these problems, and better provide support for developers to debug field failures, I introduced an overall vision that contains several techniques: BugRedux, F^3, clause weighting (CW) and on-demand formula computation (OFC), in this dissertation.

BugRedux is a general technique that can reproduce actual failing executions that mimic the original field failures in a faithful way. BugRedux works by (1) collecting dynamic field execution data that contain a sequence of program entries, which will be used as intermediate goals during the search, and (2) using a guided symbolic execution technique to synthesize executions that reach goals in this sequence and POF in the given order and result in the same observed failure. F^3 extends BugRedux with automated fault localization capabilities. F^3 starts with the relevant entries in the collected execution data found by BugRedux and works by (1) synthesizing both passing and failing executions that are similar to the original failing execution using symbolic execution and (2) leveraging the generated executions and customized statistical fault localization techniques to identify and report ranked lists of suspicious program entries that are likely to be the direct causes of field failures. In addition to these two techniques, I also presented two approaches, CW and OFC, to improve an existing formula-based debugging technique, BugAssist, by considering information provided by passing test cases and by constructing formulas used for debugging in an on-demand manner.

To assess BugRedux and F^3, I implemented both techniques in prototype tools , made them freely available, and performed several different empirical evaluation on a large set of *real-world programs and real field failures*. The results of the studies are promising. The

results of studies of BugRedux provided evidence that it can reproduce observed field failures from a suitable set of execution data. I found that, among all options I investigated in the cost-benefit study of BugRedux, partial call sequence provide the best tradeoffs between amount of information collected and effectiveness of reproducing field failures, which also provided additional evidence that typical information in traditional bug reports (*e.g.,* call stack at the moment of crash) is usually not enough for reproducing field failures. The results of studies of F^3 showed that F^3 could synthesize multiple similar passing and failing executions from a given set of field execution data, and leverage these generated executions to successfully perform effective fault localization. The results also provided evidence that the optimizations I considered in F^3 indeed could improve the results of fault localization by reducing the length of final reported lists of suspicious program entities and improving the ranks of the real faulty entities in the lists.

To assess the effectiveness and efficiency of CW and OFC, I implemented it in an early prototype tool and performed an empirical evaluation, in which I applied the prototype tool on three small C programs and compared the performance of these two optimizations with that of both a statistical fault localization technique and an existing formula-based approach. My findings of CW and OFC are also promising. These two optimizations were able to report the potentially fault statements with some mathematical explanation as effective, more accurate, and considerably more efficient compared to alternative approaches that would model the semantics of the whole program.

Several open problems could be potentially addressed in my overall vision to improve the practical usefulness of my approaches.

Besides performing a thorough user study to confirm the initial results I obtained and further assess the practical usefulness of my approaches, one potential direction involves investigating the use of fault-localization and debugging techniques, such as experimental debugging [31], other than statistical fault localization and formula-based debugging in my approaches.

Call sequences are only one possible type of execution data that can be collected from the field, and symbolic execution is just one possible input-generation technique for field

failure reproduction. Therefore, a possible extension of my dissertation is to investigate whether other types of field data and (possibly less expensive) input-generation techniques can be successfully used in the context of my approach.

One potential extension of techniques discussed in Chapter 6 is to investigate approaches that can further address the inherently limited scalability of formula-based debugging. In particular, two future research directions can be considered: (1) further simplifying the constructed MAX-SAT problems to decrease the cost of solving these problems and (2) trying to decompose the debugging problem into several subproblems (*e.g.,* at the procedure level) that can be solved efficiently and in a modular fashion.

Finally, my dissertation is also related to automated program repair. A future work can also investigate whether my debugging techniques can help automated program repair. Intuitively, the clauses in the CoMSSs produced by the improved formula-based debugging technique should be able to inform and guide automated program repair techniques in finding or synthesizing suitable repairs.

REFERENCES

[1] "Breakpad." http://code.google.com/p/google-breakpad/, Apr. 2012.

[2] "Software-artifact Infrastructure Repository." http://sir.unl.edu/, Apr. 2012.

[3] "Technical Note TN2123: CrashReporter." http://developer.apple.com/technotes/tn2004/tn2123.html, Apr. 2012.

[4] "The Amazing VM Record/Replay Feature in VMware Workstation 6." http://communities.vmware.com/community/vmtn/cto/steve/blog/2007/04/18/the-amazing-vm-recordreplay-feature-in-vmware-workstation-6, Apr. 2012.

[5] "tmin: Fuzzing Test Case Optimizer." http://code.google.com/p/tmin/, Apr. 2012.

[6] "Windows Error Reporting: Getting Started." http://www.microsoft.com/whdc/maintain/StartWER.mspx, 2012.

[7] "Z3." http://z3.codeplex.com/, Oct. 2013.

[8] "LUA_OBJCACHE segfault bug in Redis." https://github.com/antirez/redis/commit/ea0e2524aae1bbd0fa6bd29e1867dc1ca133bfa5, May 2014.

[9] "Test cases for scripting.c in Redis." https://github.com/antirez/redis/blob/7f772355f403a1be9592e60f606d457d117fccc5/tests/unit/scripting.tcl, May 2014.

[10] "Dr. Watson overview." https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx?mfr=true, 2015.

[11] Abreu, R., Zoeteweij, P., and Gemund, A. J. C. v., "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proc. of the 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46, 2006.

[12] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C., "An Observation-based Model for Fault Localization," in *Proc. of the 2008 International Workshop on Dynamic Analysis*, pp. 64–70, 2008.

[13] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[14] Artzi, S., Dolby, J., Tip, F., and Pistoia, M., "Fault Localization for Dynamic Web Applications," *IEEE Transactions on Software Engineering*, vol. 38, pp. 314 –335, March-April 2012.

[15] Artzi, S., Dolby, J., Tip, F., and Pistoia, M., "Directed Test Generation for Effective Fault Localization," in *Proc. of the 19th International Symposium on Software Testing and Analysis*, pp. 49–60, 2010.

[16] Artzi, S., Kim, S., and Ernst, M. D., "ReCrash: Making Software Failures Reproducible by Preserving Object States," in *Proc. of the 22nd European Conference on Object-Oriented Programming*, pp. 542–565, 2008.

[17] Ashok, B., Joy, J., Liang, H., Rajamani, S. K., Srinivasa, G., and Vangala, V., "DebugAdvisor: A Recommender System for Debugging," in *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 373–382, 2009.

[18] Avgerinos, T., Cha, S. K., Hao, B. L. T., and Brumley, D., "AEG: Automatic Exploit Generation," in *Proc. of the 18th Network and Distributed System Security Symposium*, Feb. 2011.

[19] Baah, G. K., Podgurski, A., and Harrold, M. J., "Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization," in *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 146–156, 2011.

[20] Bailey, J. and Stuckey, P. J., "Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization," in *Proc. of the 7th International Conference on Practical Aspects of Declarative Languages*, pp. 174–186, 2005.

[21] Cadar, C., Dunbar, D., and Engler, D., "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, pp. 209–224, 2008.

[22] Castro, M., Costa, M., and Martin, J.-P., "Better Bug Reporting with Better Privacy," in *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 319–328, 2008.

[23] Chandra, S., Fink, S. J., and Sridharan, M., "Snugglebug: A Powerful Approach to Weakest Preconditions," in *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 363–374, 2009.

[24] Chandra, S., Torlak, E., Barman, S., and Bodik, R., "Angelic Debugging," in *Proc. of the 33rd International Conference on Software Engineering*, pp. 121–130, 2011.

[25] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K., "Holmes: Effective Statistical Debugging via Efficient Path Profiling," in *Proc. of the 31st International Conference on Software Engineering*, pp. 34–44, 2009.

[26] Christ, J., Ermis, E., Schäf, M., and Wies, T., "Flow-sensitive Fault Localization," in *Proc. of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2013.

[27] Clarke, E., Kroening, D., and Lerda, F., "A Tool for Checking ANSI-C Programs," in *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2988, pp. 168–176, 2004.

[28] Clause, J. and Orso, A., "A Technique for Enabling and Supporting Debugging of Field Failures," in *Proc. of the 29th International Conference on Software Engineering*, pp. 261–270, 2007.

[29] Clause, J. and Orso, A., "Penumbra: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting," in *Proc. of the 2009 International Symposium on Software Testing and Analysis*, pp. 249–260, 2009.

[30] Clause, J. and Orso, A., "Camouflage: Automated Anonymization of Field Data," in *Proc. of the 33rd International Conference on Software Engineering*, pp. 21–30, 2011.

[31] Cleve, H. and Zeller, A., "Locating Causes of Program Failures," in *Proc. of the 27th International Conference on Software Engineering*, pp. 342–351, 2005.

[32] Crameri, O., Bianchini, R., and Zwaenepoel, W., "Striking a New Balance Between Program Instrumentation and Debugging Time," in *Proc. of the 6th European Conference on Computer Systems*, pp. 199–214, 2011.

[33] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K., "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transaction on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[34] Dallmeier, V., Zeller, A., and Meyer, B., "Generating Fixes from Object Behavior Anomalies," in *Proc. of the 24th IEEE International Conference on Automated Software Engineering*, pp. 550–554, IEEE Computer Society, 2009.

[35] Dijkstra, E. W., *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 1997.

[36] Dutertre, B. and de Moura, L., "The YICES SMT Solver.".

[37] Elbaum, S. and Diep, M., "Profiling Deployed Software: Assessing Strategies and Testing Opportunities," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 312–327, 2005.

[38] Eric Wong, W., Debroy, V., and Choi, B., "A Family of Code Coverage-based Heuristics for Effective Fault Localization," *Journal of System and Software*, vol. 83, pp. 188–208, Feb. 2010.

[39] Ermis, E., Schäf, M., and Wies, T., "Error Invariants," in *Proc. of the 18th International Symposium on Formal Methods*, pp. 187–201, 2012.

[40] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R., "Extended Static Checking for Java," in *Proc. of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 234–245, 2002.

[41] GNU, "Findutils - GNU find utilities." `http://www.gnu.org/software/findutils/`, 2005.

[42] Godefroid, P., Klarlund, N., and Sen, K., "DART: Directed Automated Random Testing," in *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 213–223, 2005.

[43] Godefroid, P., Levin, M. Y., and Molnar, D. A., "Automated Whitebox Fuzz Testing," in *Proc. of the Network and Distributed System Security Symposium*, 2008.

[44] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T., "RERAN: Timing- and Touch-sensitive Record and Replay for Android," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 72–81, IEEE Press, 2013.

[45] Groce, A., "Error Explanation with Distance Metrics," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 108–122, 2004.

[46] Hilbert, D. M. and Redmiles, D. F., "Extracting Usability Information from User Interface Events," *ACM Computing Surveys*, vol. 32, pp. 384–421, Dec. 2000.

[47] Holland, J. H., *Adaptation in Natural and Artificial Systems.* Cambridge, MA, USA: MIT Press, 1992.

[48] Jeffrey, D., Feng, M., Gupta, N., and Gupta, R., "BugFix: A Learning-based Tool to Assist Developers in Fixing Bugs," in *Proc. of the 17th International Conference on Program Comprehension*, pp. 70–79, 2009.

[49] Jeffrey, D., Gupta, N., and Gupta, R., "Fault Localization Using Value Replacement," in *Proc. of the 2008 International Symposium on Software Testing and Analysis*, pp. 167–178, 2008.

[50] Jiang, L. and Su, Z., "Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths," in *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 184–193, 2007.

[51] Jin, W. and Orso, A., "BugRedux: Reproducing Field Failures for In-house Debugging," in *Proc. of the 34th International Conference on Software Engineering*, pp. 474–484, 2012.

[52] Jin, W. and Orso, A., "F3: Fault Localization for Field Failures," in *Proc. of the 2013 International Symposium on Software Testing and Analysis*, pp. 213–223, 2013.

[53] Jones, J. A., Harrold, M. J., and Stasko, J., "Visualization of Test Information to Assist Fault Localization," in *Proc. of the 24th International Conference on Software Engineering*, pp. 467–477, 2002.

[54] Jose, M. and Majumdar, R., "Cause Clue Clauses: Error Localization Using Maximum Satisfiability," in *Proc. of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 437–446, 2011.

[55] King, J. C., "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[56] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W., "Genprog: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, pp. 54 –72, 2012.

[57] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I., "Bug Isolation via Remote Program Sampling," in *Proc. of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 141–154, 2003.

[58] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., and JORDAN, M. I., "Scalable Statistical Bug Isolation," in *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15–26, 2005.

[59] LIBLIT, B., *Cooperative Bug Isolation.* PhD thesis, University of California, Berkeley, 2004.

[60] LIU, C., YAN, X., FEI, L., HAN, J., and MIDKIFF, S. P., "SOBER: Statistical Model-based Bug Localization," in *Proc. of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 286–295, 2005.

[61] LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., and ZHOU, Y., "BugBench: Benchmarks for Evaluating Bug Detection Tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[62] MERZ, F., FALKE, S., and SINZ, C., "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR," in *Proc. of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, pp. 146–161, 2012.

[63] NAISH, L., LEE, H. J., and RAMAMOHANARAO, K., "A Model for Spectra-based Software Diagnosis," *ACM Transactions on Software Engineering Methodology*, vol. 20, pp. 11:1–11:32, August 2011.

[64] NANDA, M. G. and SINHA, S., "Accurate Interprocedural Null-Dereference Analysis for Java," in *Proc. of the 31st International Conference on Software Engineering*, pp. 133–143, 2009.

[65] PACHECO, C. and ERNST, M. D., "Randoop: Feedback-Directed Random Testing for Java," in *Proc. of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pp. 815–816, 2007.

[66] PACHECO, C., LAHIRI, S. K., and BALL, T., "Finding Errors in .NET with Feedback-Directed Random Testing," in *Proc. of the 2008 International Symposium on Software Testing and Analysis*, pp. 87–96, 2008.

[67] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., and LU, S., "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors," in *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp. 177–192, 2009.

[68] PARNIN, C. and ORSO, A., "Are Automated Debugging Techniques Actually Helping Programmers?," in *Proc. of the 2011 International Symposium on Software Testing and Analysis*, pp. 199–209, July 2011.

[69] PAVLOPOULOU, C. and YOUNG, M., "Residual Test Coverage Monitoring," in *Proc. of the 21st International Conference on Software Engineering*, pp. 277–284, 1999.

[70] RÖβLER, J., FRASER, G., ZELLER, A., and ORSO, A., "Isolating Failure Causes Through Test Case Generation," in *Proc. of the 2012 International Symposium on Software Testing and Analysis*, pp. 309–319, 2012.

[71] Rodgers, J. L. and Nicewander, W. A., "Thirteen ways to look at the correlation coefficient," *The American Statistician*, vol. 42, pp. 59–66, 1988.

[72] Rössler, J., Zeller, A., Fraser, G., Zamfir, C., and Candea, G., "Reconstructing Core Dumps," in *Proc. of the 6th International Conference on Software Testing*, pp. 114–123, 2013.

[73] Santelices, R., Jones, J. A., Yu, Y., and Harrold, M. J., "Lightweight Fault-localization Using Multiple Coverage Types," in *Proc. of the 31st International Conference on Software Engineering*, pp. 56–66, 2009.

[74] Sen, K., Marinov, D., and Agha, G., "CUTE: A Concolic Unit Testing Engine for C," in *Proc. of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 263–272, 2005.

[75] Sumner, W. N., Bao, T., and Zhang, X., "Selecting Peers for Execution Comparison," in *Proc. of the 2011 International Symposium on Software Testing and Analysis*, pp. 309–319, 2011.

[76] Torlak, E., Vaziri, M., and Dolby, J., "MemSAT: Checking Axiomatic Specifications of Memory Models," in *Proc. of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 341–350, 2010.

[77] Tsankov, P., Jin, W., Orso, A., and Sinha, S., "Execution Hijacking: Improving Dynamic Analysis by Flying Off Course," in *Proc. of the 4th International Conference on Software Testing*, pp. 200–209, IEEE, 2011.

[78] Visser, W., Păsăreanu, C. S., and Khurshid, S., "Test Input Generation with Java PathFinder," *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[79] Xie, X., Chen, T. Y., Kuo, F.-C., and Xu, B., "A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization," *ACM Transactions on Software Engineering Methodology*, vol. 22, pp. 31:1–31:40, October 2013.

[80] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S., "SherLog: Error Diagnosis by Connecting Clues from Run-time Logs," in *Proc. of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 143–154, 2010.

[81] Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S., "Improving Software Diagnosability via Log Enhancement," in *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–14, 2011.

[82] Zamfir, C. and Candea, G., "Execution Synthesis: A Technique for Automated Software Debugging," in *Proc. of the 5th European Conference on Computer Systems*, pp. 321–334, 2010.

[83] Zeller, A., "Yesterday, my program worked. Today, it does not. Why?," in *Proc. of the 7th European Software Engineering Conference Held Jointly With the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 253–267, 1999.

[84] Zeller, A., "Isolating Cause-effect Chains From Computer Programs," in *Proc. of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 1–10, 2002.

[85] Zeller, A. and Hildebrandt, R., "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, vol. 28, pp. 183–200, Feb. 2002.

[86] Zhang, X., Gupta, N., and Gupta, R., "Locating Faults Through Automated Predicate Switching," in *Proc. of the 28th International Conference on Software Engineering*, pp. 272–281, 2006.

[87] Zhang, Z., Chan, W. K., Tse, T. H., Jiang, B., and Wang, X., "Capturing Propagation of Infected Program States," in *The 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 43–52, 2009.

[88] Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., and Weiss, C., "What Makes a Good Bug Report?," *IEEE Transactions on Software Engineering*, vol. 36, pp. 618–643, Sept. 2010.