IMPROVING THE UTILITY OF COMPILER FUZZERS

by

Yang Chen

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2014

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Yang Chen**

has been approved by the following supervisory committee members:

John Regehr	, Chair	**11/01/2013** Date Approved
Matthew Flatt	, Member	**11/01/2013** Date Approved
Mary Hall	, Member	**11/01/2013** Date Approved
Matthew Might	, Member	**11/01/2013** Date Approved
Alex Groce	, Member	**11/01/2013** Date Approved

and by **Alan Davis** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Aggressive random testing tools, or fuzzers, are impressively effective at finding bugs in compilers and programming language runtimes. For example, a single test-case generator has resulted in more than 460 bugs reported for a number of production-quality C compilers. However, fuzzers can be hard to use. The first problem is that failures triggered by random test cases can be difficult to debug because these tests are often large. To report a compiler bug, one must often construct a small test case that triggers the bug. The existing automated test-case reduction technique, delta debugging, is not sufficient to produce small, reportable test cases. A second problem is that fuzzers are indiscriminate: they repeatedly find bugs that may not be severe enough to fix right away. Third, fuzzers tend to generate a large number of test cases that only trigger a few bugs. Some bugs are triggered much more frequently than others, creating needle-in-the-haystack problems. Currently, users rule out undesirable test cases using ad hoc methods such as disallowing problematic features in tests and filtering test results.

This dissertation investigates approaches to improving the utility of compiler fuzzers. Two components, an aggressive test-case reducer and a tamer, are added to the fuzzing workflow to make the fuzzer more user friendly. We introduce C-Reduce, an aggressive test-case reducer for C/C++ programs, which exploits rich domain-specific knowledge to output test cases nearly as good as those produced by skilled humans. This reducer produces outputs that are, on average, more than 30 times smaller than those produced by the existing reducer that is most commonly used by compiler engineers. Second, this dissertation formulates and addresses the *fuzzer taming problem*: given a potentially large number of random test cases that trigger failures, order them such that diverse, interesting test cases are highly ranked. Bug triage can be effectively automated, relying on techniques from machine learning to suppress duplicate bug-triggering test cases and test cases triggering known bugs. An evaluation shows the ability of this tool to solve the fuzzer taming problem for 3,799 test cases triggering 46 bugs in a C compiler.

To my wife and my parents, for their unwavering love, support, and encouragement

CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I want to thank my advisor John Regehr, for his endless support and motivating advice to my research and for his great patience and tolerance to my ineptitude and mistakes over these years. I am fortunate and grateful to being supervised by John, from whom I learned how to do research. John has always shown me how to solve research problems in simple and effective ways and guided me to focus on the important ones. I am indebted to the innumerous amount of time he spent on answering my research questions, sending me insightful feedback on my research, and helping me in preparing presentation and technical writing. Without his guidance, this dissertation would never have been completed.

I also thank my thesis committee members: Matthew Flatt, Alex Groce, Mary Hall and Matt Might. Matthew enforced my knowledge on programming language design, from basic lambda calculus to advanced language semantics, and various functional programming techniques. Alex has been extremely helpful since we collaborated on our *Swarm Testing* paper; we have co-authored three papers since then. Alex contributed immensely to the chapters of taming compiler fuzzers in this dissertation. I learned CUDA from Mary; her deep thoughts on GPU and CUDA had great influence on my internship work at NVIDIA. Matt taught me how to analyze higher-order programs using advanced static analysis techniques such as k-CFA.

I am especially indebted to Eric Eide, with whom we published five research papers. Eric has helped me in almost all of my compiler-testing research. I always enjoy his sharp insights and comments on the papers that we discussed in the compiler reading group.

Many thanks to Xuejun Yang, my fellow Ph.D. pal and friend. Xuejun's work on Csmith forms a solid basis for my dissertation research.

I also thank Jaydeep Marathe and Vinod Grover, who supervised me throughout my internship at NVIDIA. I spent a great summer working with them.

I thank all my computer science fellow friends: Anton Burtsev, Jianjun Duan, Zhisong Fu, Peng Li, Suying Liang, Xin Lin, Qingyu Meng, Jon Rafkind, Weibin Sun, and Lu Zhao.

Finally and foremost, I thank my wife and my parents. Nothing would have meaning without their unwavering love, support, encouragement, and understanding.

CHAPTER 1

INTRODUCTION

Compilers and programming language runtimes are part of the trusted computing base for many software systems, and hence they need to be correct. On the other hand, modern compilers and runtimes are complex software artifacts, and bugs are hard to avoid. Random testing tools, or fuzzers, have shown their strength and effectiveness by finding bugs in production-quality compilers and runtimes. However, fuzzers can be hard to use: (1) they tend to generate large bug-inducing test cases, and (2) they repeatedly find bugs that may not be severe enough to fix right away. This dissertation therefore addresses the following research question:

> While fuzzers are powerful bug-finding tools for compilers and language runtimes, how can we improve the utility of fuzzers and make them more user-friendly for both fuzzer developers and fuzzer users?

1.1 Motivation

Bugs in compilers and programming language runtimes can affect the correct behaviors of software systems. For example, it is undesirable to the developer if the compiler crashes. Moreover, buggy compilers and runtimes can silently generate incorrect object code from a source program. Compared to *crash bugs*, which cause the compiler to crash, *wrong-code bugs*, which miscompile programs, may be even worse for software developers because erroneous behavior introduced by miscompilation is often hard to reason about—even for experienced programmers.

Bugs in compilers and runtimes can cause serious issues for security-critical and safety-critical software. For example, crashes in a language runtime running in a web browser could become exploitable vulnerabilities, which would compromise the host computer and cause unwanted disclosure of private data. More importantly, miscompiling safety-critical software, where human lives are involved, is a serious matter.

1.1.1 Fuzzers are Effective at Finding Bugs in Compilers

In recent years, two kinds of techniques, verification [10,43,44,69,78–80] and random testing [21,30,53,61,74], have been advanced to improve the correctness of compilers and programming language runtimes. For example, Compcert [2,44], a verified C compiler supporting almost all features of the ISO C 90/ANSI C language, relies on machine-assisted mathematical proofs to ensure that the compiled executable conforms with the semantics of the source program and therefore is able to guarantee the absence of miscompilation in a rigorous way. Vellvm [78–80] aims to reason about the intermediate representation of LLVM [39], a production-quality compiler, and its SSA-based transformations.

On the testing side, random testing, or fuzzing, has been proved as a powerful bug-finding tool for uncovering bugs in compilers and language runtimes. For example, Csmith, a randomized C program generator, has uncovered more than 460 previously unknown bugs in widely used C compilers [74]. *jsfunfuzz* [61] has identified more than 1,700 bugs in SpiderMonkey, the JavaScript engine used in Firefox [63]. LangFuzz [30], another fuzzer for testing the same JavaScript engine, has led to the discovery of more than 500 previously unknown bugs. Moreover, Google's ClusterFuzz is built on a cluster of several hundred virtual machines that run thousands of Chrome instances simultaneously and is able to generate about 50 million test cases per day [9]. Within a 4-month period after it was brought fully online at the end of 2011, ClusterFuzz had detected 95 vulnerabilities in Chrome.

Fully verifying existing compilers such as LLVM and GCC would certainly require tremendous amount of efforts, and by no means could be done in a short time. Thus, it is reasonable to assume that testing will still be playing an important role in improving the correctness of compilers and runtimes for the foreseeable future.

1.1.2 Fuzzers Can Be Hard to Use

Although fuzzers are effective bug-finding tools, they can be hard to use for three reasons. First, bug-inducing test cases generated by fuzzers are often large. Larger test cases are able to expose more compiler bugs [74]. One main reason is that a large test case can trigger feature interactions and exercise implementation limits in the compiler under test and therefore is more likely to expose more faults in the system. On the other hand, a large test case makes it difficult to locate the underlying fault, and hence it needs to be reduced to a small one that still triggers the failure. As a rule of thumb, bug reporters should take the responsibility to reduce bug-inducing test cases, and then compiler developers can focus their efforts on fixing bugs. In fact, the importance of test-case reduction is emphasized by both LLVM and GCC developers [25, 40]. GCC's documentation states that, "Our bug reporting instructions ask for the preprocessed version of the file that triggers the bug. Often this file is very large; there are several reasons for making it as small as possible..." A similar description appears in LLVM's page about how to submit bug reports: "The bug description should contain the following information: The reduced test-case that triggers the bug..." Manually reducing failure-inducing test cases is often an exercise of failed and successful tries. It is error-prone and is an unnecessary waste of human resources. As a consequence, a good automated test-case reducer is necessary to saving human effort.

Second, fuzzers tend to generate a large number of test cases that trigger only a few bugs, some of which are triggered many times and others only once. For example, Figure 1.1 shows two bug distributions over thousands of test cases that were generated by Csmith. It illustrates that among 46 distinct bugs, the majority are triggered fewer than 10 times, while 16 bugs are triggered by only one test case. Reporting all of these bug-inducing test cases is completely infeasible—a DoS attack on the bug-reporting system.

Third, fuzzers are indiscriminate. They, for example, cannot avoid generating test cases triggering bugs that are known to developers but marked as low priority. Compilers such as GCC or LLVM often have a large number of noncritical bugs remaining unfixed. For instance, in June 2013, GCC's bugzilla database still had more than 3,000 open bugs that were considered "normal" or higher severity and tagged as priorities P1, P2, and P3. Under the pressure of adding new features and meeting release deadlines, compiler developers may have limited time to fix bugs. It is therefore a reasonable assumption that quite a lot of the low-priority bugs would remain unfixed for a long time. It would be undesirable if a fuzzer kept generating test cases triggering those known, low-priority bugs. In practice, fuzzer users rely on a variety of ad hoc methods to rule out test cases that are noncritical or trigger known bugs, such as turning off features in the fuzzer and using hand-coded rules to filter out test cases triggering noninteresting bugs [27, 64]. Thus, manual work

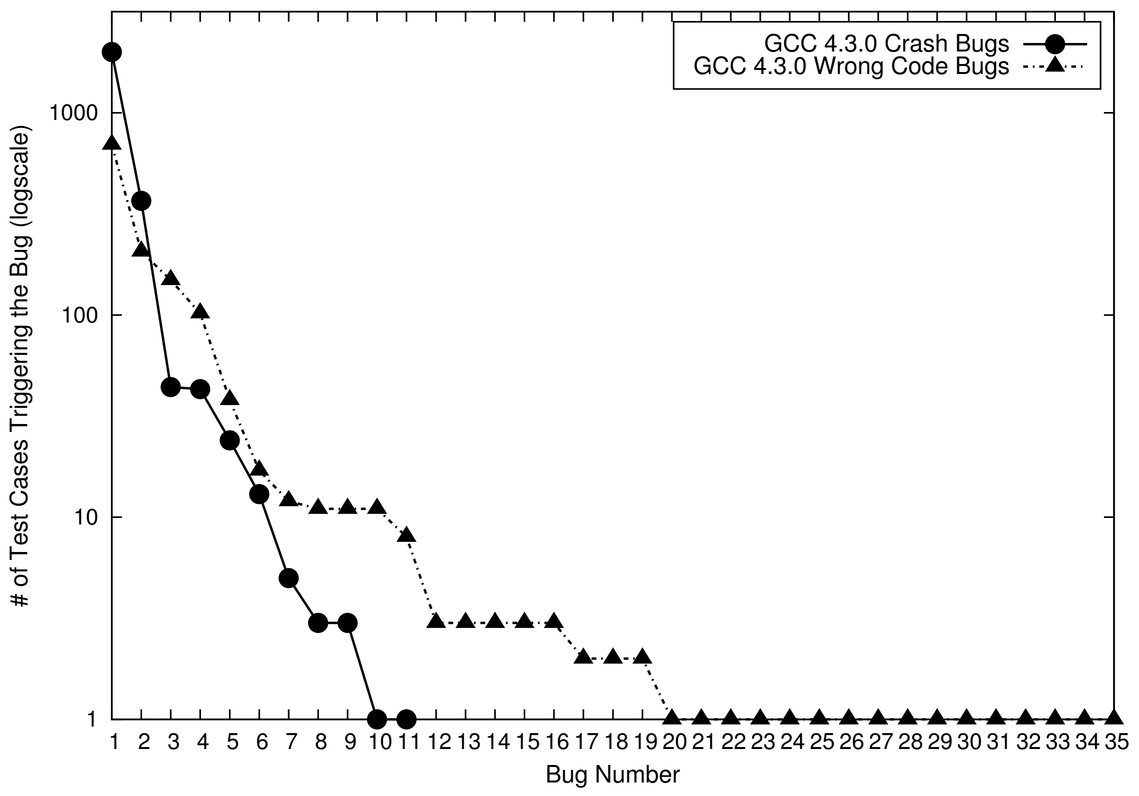

Figure 1.1: A fuzzer tends to hit some bugs thousands of times more frequently than others

is still heavily involved in determining which bug-inducing test cases are presented to developers. In fact, Csmith has lost users not because it stopped finding bugs, but because it is indiscriminate.

1.1.3 Research on Improving Compiler Fuzzers' Utility

There is surprisingly only a little research work on improving the utility of compiler fuzzers. Delta debugging, formalized by Zeller and Hildebrandt [76], is a generalized test-case reduction approach. One of their algorithms, *ddmin*, aims to minimize the size of a bug-inducing input by removing contiguous chunks of the input by greedy search. Hierarchical Delta Debugging, or HDD, was developed by Micherghi and Su [52]. HDD is able to remove code chunks other than lines, such as tree structures. Berkeley delta [51], developed by McPeak and Wilkerson, is a commonly used delta debugging tool by compiler engineers. Berkeley delta is an instance of *ddmin*, and operates at line granularity. One major problem of Berkeley delta is that the reduced output is still often too large to be directly copied into a bug report and thus, a fair amount of further manual reduction efforts is often involved in generating small, reportable test cases from the original tests.

Furthermore, little research has addressed the problem of creating bug reports for only new and important test cases by automatically examining a collection of failure-inducing test cases generated by a fuzzer. Previous research has sought to detect duplicate bug reports based on user-supplied metadata [67, 68, 72], for example, text in bug descriptions provided by bug reporters. Clustering has also been used to separate test cases triggering distinct bugs [1, 24, 33, 57]. However, compiler fuzzers have unique characteristics: First, user-supplied metadata is not available for fuzzer output, unlike human-created bug reports. Second, optimizing production-quality compilers are complex and hence, it is infeasible to predict what bug-inducing test cases would look like. Third, the compiler under test shows no observable behavior upon wrong-code bugs—the compiler fails "silently" without any assertion errors. Fourth, fuzzers tend to find a small number of bugs much more frequently than others; some rare bugs are triggered only once out of a million test cases. These characteristics impose new research challenges that cannot be solved simply by adopting the aforementioned techniques.

1.2 Thesis Statement

My dissertation research is about making compiler fuzzers more useful for a broader audience such as compiler developers. To make compiler fuzzers more user-friendly, I claim that a better test-case reducer needs to perform reduction aggressively. A better reducer should reduce original bug-inducing inputs to small ones that are comparable to those produced by skilled humans. Moreover, a much more sophisticated approach is feasible to automatically suppress duplicate bug-inducing test cases and present compiler developers only those that are important, each triggering a distinct bug that compiler developers are willing to fix right away. Thus, the thesis statement of my dissertation is

> *The utility of a fuzzer can be greatly improved by first integrating automated aggressive test-case reduction and then taming the fuzzer: a bug triage process relying on machine learning techniques to automatically suppress duplicate bug-inducing test cases and test cases triggering the bugs that are already known.*

1.3 Research Outline

Figure 1.2 shows the workflow for fuzzing a compiler. The oracle determines the "interestingness" of each test case—whether the test case discovers some erroneous behavior of the compiler under test. The reducer reduces each bug-triggering test case to a much smaller one, while keeping the interestingness of the original test case. Finally, the tamer presents the end user a list of minimized bug-triggering test cases that are ranked in such an order that interesting test cases are early in the list. Meanwhile, the end user is able to pass the tamer feedback, improving test-case ranking.

My dissertation research advances the techniques of test-case reduction and fuzzer taming. First, I show that source-to-source transformations are crucial to producing small bug-inducing test cases comparable to those produced by skilled humans. It is infeasible to do fine-grained source-code transformation without the knowledge of language-level syntax and semantics. For example, modifying a function signature also requires changing every call site of the function simultaneously. Similarly, reducing the dimension of an array needs to alter both the array definition and the corresponding array subscript expressions. Our test-case reducer for C/C++ programs, C-Reduce, enforces modularity and is able to

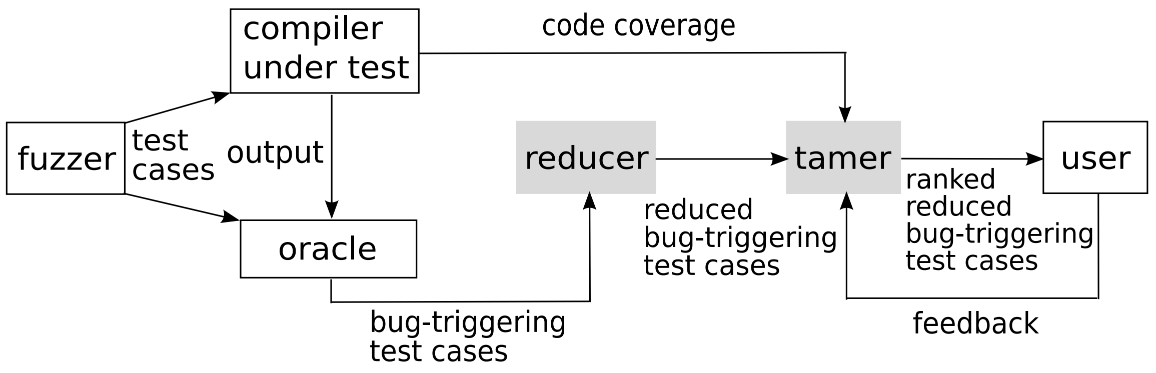
compiler under test
code coverage
fuzzer
test cases
output
reducer
tamer
user
oracle
reduced bug-triggering test cases
ranked reduced bug-triggering test cases
bug-triggering test cases
feedback

Figure 1.2: Workflow for fuzzing a compiler

do automated aggressive test-case reduction by invoking pluggable transformations until a fixpoint is reached. In C-Reduce, I implemented 63 source-to-source transformations using LLVM's Clang front-end. For 3,799 test cases triggering bugs in a C compiler, C-Reduce is able to produce outputs that are of 174 bytes on average, more than 500 times smaller than 101,744 bytes, the average size of unreduced test cases. Moreover, those reduced test cases often cannot be improved, even by skilled developers.

Second, I show that the challenge of automatic bug triage can be solved as the *fuzzer taming problem*, which is phrased as the following:

> *Given a potentially large collection of bug-inducing test cases, each of which triggers a bug, rank them in such a way that test cases triggering distinct bugs are early in the list.*

Based on the insight that bugs are highly diverse, we first exploit various features from diverse resources that may assist in differentiating test cases in terms of the underlying bugs, including test-case text, execution traces of the compiler under test running against test cases, and the compiler's failure outputs (if they exist). Then the tests can be ranked in a feature-metric space using the *furthest point first* (FPF) technique from machine learning [26]. Another obvious approach to fuzzer taming is clustering: given the features extracted from the tests, we can use a clustering algorithm to generate clusters where the test cases in the same cluster are more similar to each other than those in other clusters and then select one test case from each cluster. However, compared with FPF ranking, clustering suffers from some drawbacks. First, it is computationally expensive, especially when feature vectors are large. More importantly, clustering is less effective than FPF ranking. Our approach can effectively solve the fuzzer taming problem for 2,501 test cases triggering 11 distinct crash bugs in a C compiler and 1,298 test cases triggering 35 distinct wrong-code bugs in the same C compiler. The user can see all 11 crash bugs by examining the first 15 tests ranked in the FPF order, 32 times faster than randomly looking through the tests. For wrong-code bugs, the improvement is 2.6×. But more importantly, using our approach, the user is able to find 12 distinct bugs by examining the first 15 tests from the list—a substantial improvement over random inspection and clustering, which expose only 5 and 6 distinct bugs, respectively.

1.4 Research Merits

C-Reduce is the first work where compiler-like transformations are used for aggressive test-case reduction, as far as I am aware of. These transformations are necessary to produce test cases nearly as good as those produced by skilled developers. Moreover, C-Reduce has shown its impact in compiler communities:

1. It is used by compiler developers;
2. It is included in GCC's documentation page about test-case reduction [4];
3. Industrial hackers have submitted patches to us, including new transformations and various improvements.

Second, to the best of my knowledge, the fuzzer taming problem has not been addressed by the research community. My work has shown that we can solve this problem by applying a lightweight FPF ranking algorithm on diverse features pertaining to bug-inducing test cases. With these techniques, the fuzzer user can find distinct bugs by looking through a much smaller number of bug-triggering test cases than examining test cases in random order.

My dissertation work addresses the problem of making compiler fuzzers more user-friendly, and shows that the problem can be solved by *automated aggressive test-case reduction* and *fuzzer taming*.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 describes C-Reduce, the design and implementation of its compiler-like transformations. Chapter 3 gives the results of applying C-Reduce to reduce 3,799 test cases that trigger 46 bugs in GCC 4.3.0 and 10 test cases that were reported to crash a variety of GCC and LLVM versions. Related work of test-case reduction is discussed in Chapter 4. Chapters 2–4 have text from a PLDI 2012 paper [59], joint work with John Regehr, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Chapter 5 presents the approach to fuzzer taming. Chapter 6 describes the experiments to evaluate our fuzzer taming approach, and the results are also discussed. I survey related work for taming compiler fuzzers in Chapter 7. Chapters 5–7 contain text from a PLDI 2013 paper [17], which was a collaboration with Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. I conclude in Chapter 8.

CHAPTER 2

TEST-CASE REDUCTION

Large test cases generated by a fuzzer tend to uncover more bugs than do small test cases, given a fixed test budget [7, 74]. However, including a large bug-inducing test case in a bug report is unacceptable to compiler developers because the large test case complicates the debugging process of locating the underlying bug in the compiler under test and unnecessarily wastes the developers' time. Distilling a bug-triggering test case to its essence is therefore an important step towards making an acceptable bug report. Given a large bug-inducing test case, test-case reduction is the process of constructing a small test case that still triggers the compiler bug. A test-case reducer automates the test-case reduction process and can significantly save human effort. Moreover, a test-case reducer should have more general use cases: it is not limited to reducing large test cases generated by a fuzzer; but more importantly, it is capable of producing small bug-triggering inputs in a more general sense—e.g., reducing a bug-triggering test case encountered when building a large project.

Ideally, a test-case reducer should automatically reduce a large bug-triggering test case into a reportable one: a self-contained small bug-triggering test case that can be directly pasted into a bug report. In other words, the test-case reducer needs to produce small bug-inducing test cases comparable to those produced by experienced developers. Figure 2.1 shows a reportable test case that demonstrates a bug in GCC 4.3.0. The test case was produced by our test-case reducer from a 57 KB bug-inducing test case generated by Csmith 2.1.0. To the best of my knowledge, none of the contemporary reducers are able to yield results as good as the one in Figure 2.1.

In this Chapter, I describe the design and implementation of our test-case reducer, C-Reduce. Some of the text in this Chapter is from a PLDI 2012 paper [59], a joint work with

```
int printf (const char *, ...);
struct
{
   int f5:1;
}
a;
int b = 1;
int
main ()
{
   a.f5 = 0 != b;
   printf ("%d\n", a.f5);
   return 0;
}
```

Figure 2.1: A reportable test case that triggers a bug in GCC 4.3.0

John Regehr, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang.[1]

2.1 Background

Manual test-case reduction is often an expensive exercise that requires a lot of successful and failed attempts to alter different parts of the test case being reduced until the test case cannot be reduced further. Given a test case triggering a compiler bug, a typical manual reduction process works as follows:

1. turn the current test case into a new variant by removing certain components of the code from it;
2. check if the variant still triggers the compiler bug: if it does, then save it as the current test case and go to step (1); otherwise roll back to the previous test case and try to remove another code component;
3. terminate if no new transformation could make the current test case smaller while still triggering the compiler bug.

The last successful variant is the final result.

[1]The definitive version was published in *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, Jun. 2012. http://doi.acm.org/10.1145/2345156.2254104

Manually repeating those reduction steps is tedious and error-prone. For example, it would take more than an hour for a moderately experienced user to transform a large Csmith-generated bug-triggering test case into a reportable one. In this case, manual test-case reduction is time-consuming but still manageable. In other cases, for example, reducing a heavily-templated C++ program of hundreds of thousands of lines, manual test-case reduction would be too difficult to be worth trying. For instance, the developer says, "I first wrote delta out of necessity. Scott and I had a quarter-million line (after prepossessing) C++ input that would crash the C++ front-end we were working on, Elsa; there was just no way we were going to minimize that by hand" [51]. Even worse, undefined behavior is very easy to introduce in the course of reducing test cases triggering compiler wrong-code bugs and therefore, the final reduced test case would become invalid. Since it is often hard to identify which reduction step brings in undefined behavior, the reduction has to be restarted.

Due to the high cost of producing small bug-inducing inputs, a compiler user could easily decide to just find a work-around when confronted with a compiler bug, rather than spending significant effort to narrow it down. On the other hand, both GCC and LLVM developers emphasize the importance of test-case reduction in their instructions for submitting bug reports [25, 40]. It would not be hard to imagine that if the user reports a large unreduced test case to compiler developers, the possible bug triggered by this test case would go unfixed forever.

2.1.1 Delta Debugging

Test-case reduction needs to be automated because of the inefficiency and the difficulty of manual test-case reduction. Delta debugging, formulated by Zeller and Hildebrandt [76], is the most widely used technique for conducting automated test-case reduction. They developed two delta debugging algorithms: an isolation algorithm *dd* and a simplification algorithm *ddmin*. *Dd* is a general delta debugging algorithm, which tries to isolate the difference between a failure-inducing (or failing) test case and a passing test case, i.e., a test case that does not trigger the desired behavior in the system under test. A minimized test case is obtained by minimizing the failing input as well as maximizing the passing input. *Ddmin* is a special case of the *dd* algorithm, and it simplifies the failing input by reducing the size of the test input.

Ddmin minimizes the size of a failing test by removing contiguous substrings, or chunks, to generate a series of *variants*. Unsuccessful variants are those that fail to trigger the bug behavior in the system and are simply discarded. Successful variants, on the other hand, are those that still uncover the desired behavior. Each successful variant is then used as the new basis for producing further variants. If no successful variants can be generated from the current basis, the chunk granularity is increased by splitting the current chunks into smaller ones. The algorithm terminates when meeting two criteria: (1) no more successful variants can be produced and (2) the chunk granularity cannot be increased further. The last successful variant is the final result, which is, by the nature of the search, the smallest test case that preserves the sought-after behavior. Removing any single chunk of the final variant would make the bug disappear.

Figure 2.2 shows a simple example that illustrates the above minimization steps. At step (1), the entire test forms a single chunk. Because removing this chunk would obviously make the bug disappear, it is split into two subsets. Then the algorithm checks the variants obtained by removing either of those subsets. If neither of them is a successful variant, the chunk size is decreased further. At step (3), removing the second chunk is able to produce a successful variant, which then becomes the new basis for further reduction. The algorithm continues until reaching step (7), where no successful variants could be produced and the chunk size could not be decreased any more. The result is the test where each of the remaining chunks is significant for preserving the faulty behavior.

Hierarchical Delta Debugging [52], or HDD, generalizes the idea of the *ddmin* to exploit hierarchical structure of the input. Instead of treating the test input as a set of flat chunks, HDD first constructs a tree structure—e.g., an abstract syntax tree—for the test input and then applies a *ddmin*-like algorithm to each level of the tree, from the top to the bottom. Handling only one level of the tree each time enables the delta debugging algorithm to only work on a small portion of the input. Consequently, the reduction runtime can be improved by removing irrelevant nodes in the early stage. HDD was evaluated on several XSLT files and C programs, which crashed a Mozilla web browser and a C compiler, respectively. Compared with *ddmin*, HDD was able to take many fewer minimization steps to produce even simpler outputs.

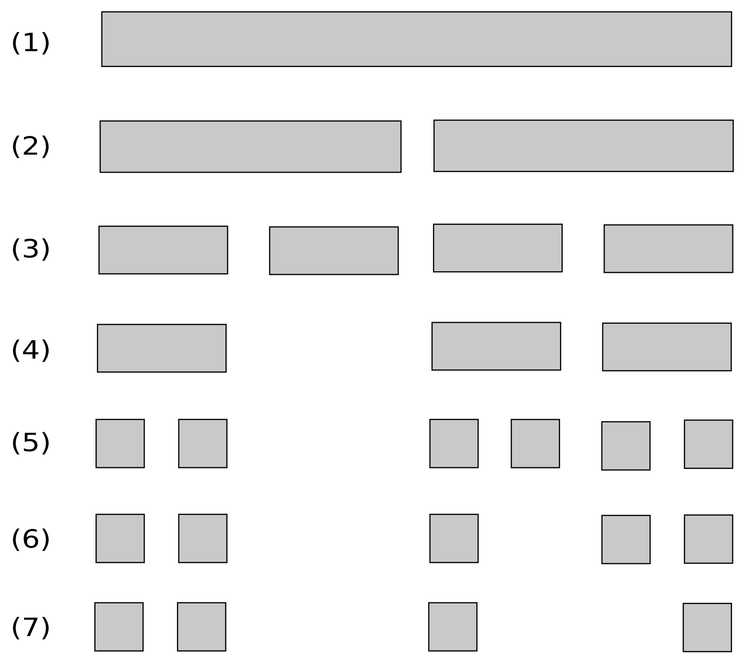

Figure 2.2: Minimizing an input using the *ddmin* algorithm

Berkeley delta, implemented by McPeak and Wilkerson [51], is a variant of *ddmin*. It is the de facto test-case reducer widely used by compiler developers. Berkeley delta is line-based: it produces variants by removing one or more contiguous lines from the input test. *Topformflat*, a utility tool companioned with Berkeley delta, is able to preprocess a test input and put all nested substrings between two balanced delimiters onto a single line. *Topformflat* greatly improves the effectiveness of Berkeley delta because it enables conducting reduction based on certain hierarchical structure of the input. Berkeley delta is simple and effective.

2.1.2 Limitation of State-of-the-art Delta Debugging

The state-of-the-art delta debugging algorithms such as *dd*, *ddmin*, and HDD are effective at minimizing failure-inducing test cases, but they are insufficient to turn them into reportable ones. So, significant manual effort often needs to be performed on the output from these tools in order to produce test cases nearly as good as those produced by skilled developers. For example, for the same unreduced test case used in Figure 2.1, Berkeley delta produces a 6.2 KB output, which is about 40 times larger than the reportable one produced by C-Reduce.

There are two main reasons for the insufficiency of the existing delta debugging techniques. First, *ddmin*, and thus Berkeley delta, cannot remove several chunks at a time and therefore, they would fail to take any minimization opportunities that require making multiple changes batched in a single iteration on the test input. Second, neither *ddmin* nor HDD takes a full advantage of exploiting language rules: syntax and semantics of the language in which the test case is written. In most cases, relying on balanced delimiters to restructure lines or removing nodes at the same abstract-syntax-tree (AST) levels are not enough to produce very small test cases.

Figures 2.3–2.5 demonstrate this limitation of the existing delta debugging techniques. Figure 2.4 and 2.5 display some "bad" transformations—producing syntactically-ill-formed outputs—and "good" transformations—producing syntactically correct outputs, respectively, for the code shown in Figure 2.3. Line-based delta debugging tools such as Berkeley Delta would transform the original code into the codes in Figures 2.4(a) or 2.4(b), neither of which is able to pass compiler's syntax checking. Figure 2.4 shows one more syntactically incorrect code produced by another transformation that only removes the parameter and the corresponding argument and hence leaves an undefined reference of the parameter. This kind of code would also be rejected by the compiler's syntax checking pass.

To the best of my knowledge, none of the existing test-case reducers for C/C++ programs would perform one-step transformations as those in Figure 2.5. For example, the code shown in Figure 2.5(a) is obtained by making the following changes all at once:

- removing parameter *p1* from function *foo*'s signature,
- removing any use of parameter *p1* inside function *foo*'s body, and
- removing parameter *p1*'s corresponding argument at function *foo*'s call site at line 7.

Some compiler bugs can be triggered by syntactically incorrect code—e.g., the code in Figure 2.4(a). However, in my experience, failing to make batched alterations while generating syntactically correct code often prevents the reducer from making further reduction towards constructing smaller outputs. For example, in Figure 2.5(a), the definition of local variable *a* in function *bar* can be removed by the simple line-based delta—if this definition is irrelevant to the bug because now there is no more reference to variable *a*. On the other hand, removing the same definition while leaving the reference of *a* at line 6 in the original code results in syntactically incorrect code. This kind of dependency chain, e.g.,

```
int foo(int p1, int p2) {
  return p1 + p2;
}
int bar(void) {
  int a = 1;
  return foo(a, 2);
}
```

Figure 2.3: Original code to be transformed

```
int foo( int p2) {
  return p1 + p2;
}
int bar(void) {
  int a = 1;
  return foo(a, 2);
}
```

(a)

```
int foo(int p1, int p2) {
  return p1 + p2;
}
int bar(void) {
  int a = 1;
  return foo( 2);
}
```

(b)

```
int foo( int p2) {
  return p1 + p2;
}
int bar(void) {
  int a = 1;
  return foo( 2);
}
```

(c)

Figure 2.4: Transformations resulting in syntactically incorrect codes. (a) Code transformed by removing parameter *p1*; (b) Code transformed by removing argument *a*; (c) Code transformed by removing both parameter *p1* and the corresponding argument *a*.

```
int foo( int p2) {
  return p2;
}
int bar(void) {
  int a = 1;
  return foo( 2);
}
```

(a)

```
int foo( int p2) {
  int p1 = 0;
  return p1 + p2;
}
int bar(void) {
  int a = 1;
  return foo( 2);
}
```

(b)

Figure 2.5: Transformations producing syntactically correct results. (a) Code transformed by removing parameter *p1*, the use of p1 and the corresponding argument *a*; (b) Code transformed by removing argument *a* and turning parameter *p1* to a local definition.

the removal of *a*'s definition depends on the removal of its use, can be quite long, leaving outputs that are too large to be suitable for bug reports. In practice, users of Berkeley delta run the tool multiple times, and between each run they do certain manual reduction, which removes some obstacles that stop the tool from doing further reduction. In some cases, manually breaking dependencies is trivial. On the other hand, operations such as manually collapsing the class hierarchy or instantiating templates would be quite laborious. A test-case reducer better than the existing ones should make manual interference unnecessary during the reduction process.

2.2 Avoiding Undefined and Unspecified Behaviors

Undefined behavior is "behavior, upon use of nonportable or erroneous program construct or erroneous data, for which this International Standard imposes no requirements," as specified in the C99 standard [32]. In other words, the compiler has no obligation upon undefined behavior and is free to emit any kind of code, e.g., it would generate code to erase all data on the disk. An example of undefined behavior is signed integer overflow. Unspecified behavior, on the other hand, stands for the situation where the compiler is allowed to choose one from a variety of implementation alternatives. The evaluation order of a function's arguments is an example of unspecified behavior. The C99 standard describes more than 190 kinds of undefined and unspecified behaviors.

On one hand, compilers exploit undefined behavior to generate optimized code. They basically assume the absence of undefined behavior in the program being compiled. On the other hand, the execution of the program containing any undefined behavior is unpredictable: it may exhibit faulty behavior, or it might just work with certain compiler versions, but would suddenly fail with another compiler. Moreover, executing undefined behavior often causes program bugs that are hard to identify and thus must be avoided. Some detailed discussion about undefined behavior is presented in these blog posts [38,58].

Nevertheless, it is unacceptable to report a test case that exhibits wrong-code behavior due to any undefined behavior because the output of the execution is meaningless. The faulty behavior is caused by some bug in the test case, not in the compiler under test. Similarly, submitting wrong-code test cases exercising unspecified behavior is not appreciated by compiler engineers. In fact, Keil C compiler developers state "Fewer than

1% of the bug reports we receive are actually bugs” [6]. Therefore, test cases exploring compiler wrong-code bugs must not execute any undefined behavior and must not rely on any specified behavior. Consequently, test-case reduction for wrong-code bugs must avoid producing final outputs that rely on undefined and unspecified behaviors. There are two ways to achieve this goal.

First, the test-case reducer can generate only valid variants—variants that are undefined- and unspecified-behavior free. This approach would fundamentally guarantee the absence of invalid variants. However, implementing such a reducer for general C programs is extremely hard. Another approach is to allow the reducer to produce both valid and invalid variants, but leverage an external tool to filter out invalid ones. Two existing tools, KCC [22] and Frama-C [20] are capable of detecting a large number of undefined and unspecified behaviors in the C programming language.

KCC is a semantics-based interpreter and analysis tool for C programs. It is rooted from a single formal semantics that accurately captures C and can catch many undefined behaviors. KCC does not report false positives: all errors reported by KCC are guaranteed to be real bugs in the program. Frama-C is an extensible static-analysis framework for C. It supports a value-analysis plug-in [15] and relies on the value analysis to soundly detect a sizable set of C’s undefined and unspecified behaviors.

Both KCC and Frama-C assume that the input programs can be compiled by the compiler. C-Reduce and Berkeley delta, however, are not guaranteed to produce only compilable variants, e.g., some line-based delta algorithm may produce variants that are not even syntactically correct. In practice, we solve this problem by running another validity check that invokes any convenient C compiler to compile a variant and passing this variant to KCC or Frama-C only if it had been successfully compiled by the convenient compiler.

Test cases for compiler-crash bugs, in practice, have weaker validity requirements than wrong-code test cases. As a matter of implementation quality, a compiler vendor usually fixes a crash bug even if the bug-inducing test case manifests undefined or unspecified behavior, e.g., dereferencing NULL pointers or using uninitialized variables. As a consequence, reducing crash test cases often runs more rapidly and produces smaller outputs than reducing wrong-code test cases.

2.3 C-Reduce: A Better Test-case Reducer for C/C++ Programs

C-Reduce is an automated test-case reduction tool for C/C++ programs. It is inspired by Delta Debugging and able to produce much smaller outputs than those produced by Berkeley Delta. C-Reduce reduces a test case by performing a fixpoint computation over a collection of pluggable transformations, each of which alters the text of the test case and generates a variant. Similar to *ddmin*, C-Reduce saves a successful variant—one that triggers the bug—as the input of the subsequent transformation and simply discards failing ones. C-Reduce stops when the test case cannot be reduced further. The last successful variant is the final reduced test case, which can be copied into a bug report submitted to compiler developers. Figure 2.6 shows the workflow of C-Reduce.

C-Reduce has several kinds of transformations. For example, it includes "peephole transformations" that simplify a contiguous sequence of tokens of a test case. These include turning identifiers into dummy integer constants such as 0 or 1, replacing a binary expression with one of its operands, and removing a balanced pair of curly braces and all the text between them, etc. Following the idea of Berkeley delta, C-Reduce also supports line-based delta transformation that removes contiguous lines at different granularity levels. The line-based transformation is also in the help of the *topformflat* utility, borrowed from Berkeley delta. When the line granularity becomes one, the transformation will invoke *topformflat* to reformat the test.

2.3.1 Compiler-like Source-to-source Transformation

Among all the classes of the transformations in C-Reduce, one class is, in particular, designed for overcoming the limitation of the existing delta debugging techniques described in Section 2.1.2. When the reducer alters multiple parts of the test, two naïve approaches do not work. First, the reducer cannot enumerate all possible alterations that involve more than one token because the searching space would be too large. For example, even for the trivial program such as "int main(void) {return 0;}," there are more than $1,000$ combinations of removing n tokens from it, where n ranges from 2 to 10. If we count other kinds of operations, e.g., reordering or substitution, the number of possible alterations would become much larger. Second, making blindly random changes is not acceptable. For example, within the entire searching space for a test case, the reducer randomly modifies

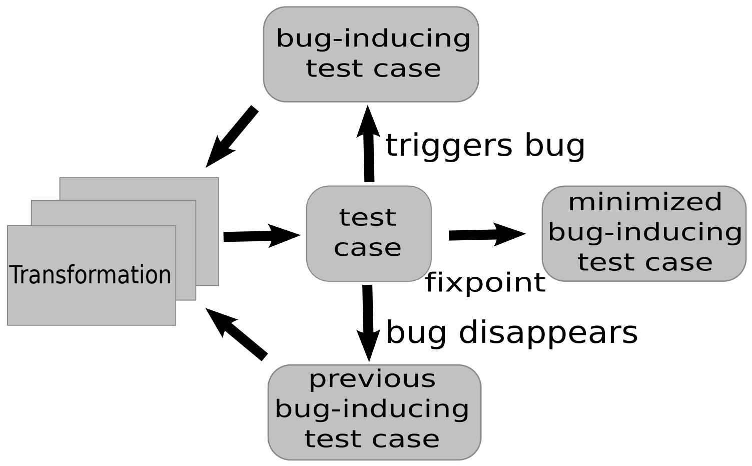

Figure 2.6: Workflow of C-Reduce

different parts of the test and terminates after a certain number of attempts. This approach is rarely useful because blindly altering the test would mostly produce codes that violate the language specification. In my experience, producing such variants has little contribution towards the goal of test-case reduction and therefore is a waste of CPU time.

Analogous to the compiler's transformation applied on the intermediate representation (IR), *Compiler-like source-to-source transformation* conducts code transformation at the source level. It follows the syntax and semantics rules defined by the language. Consequently, most of the variants produced by this transformation can be compiled by the compiler.

2.3.2 Design Principles of Compiler-like Source-to-source Transformation

Implementing compiler-like source-to-source transformation follows several design principles. First, it enforces modularity. Each transformation should be independent of the others and can be individually plugged into C-Reduce's main delta loop. Keeping high modularity is one of the key features of C-Reduce. It enables C-Reduce to draw a simple and clean interface between the core delta debugging algorithm and each transformation.

Second, a transformation does not need to preserve the meaning of the code, as compiler optimizers do. A test-case reducer aims to turn a large bug-inducing test case into a small one that preserves the desired behavior. A reduction transformation can modify the test text in various ways as long as the modification may produce a better result. It is even acceptable to produce syntactically ill-formed code, but not too often. Consequently, the code analysis involved in the transformation can be incomplete or unsound. For example, the transformation that canonicalizes class methods would fail to rename some methods due to the complexity of resolving certain kinds of class template instantiations. Another example is that there is no real array-out-of-bound analysis nor aliasing analysis involved in changing the size of an array or the index of accessing this array and thus, the related transformation is unsound in the typical static analysis point of view. However, this strategy simplifies the implementation of some complex transformations. In my experience, unsoundness does not have much impact on the quality of C-Reduce in terms of reducing failure-inducing test cases.

Third, not all transformations directly contribute to reducing the size of the test case. The insight is that some transformations temporarily increase code size, but they could create more reduction opportunities for other transformations and therefore improve overall reduction ratio. The function-inlining transformation is a good example in this category.

Fourth, learning reduction tricks is an important process of improving compiler-like transformations. To some extent, our compiler-like source-to-source transformation was motivated by our manual reduction experience and examining the ways in which those experienced developers produced small test cases. Our observation was that in order to construct reduced test cases nearly as small as the ones produced by those people, a collection of compiler-like transformations is needed. In some sense, those compiler-like source-to-source transformations mimic the operations that could be taken by a skilled developer to create a small test case. However, there are always cases where skilled developers can outperform automated test-case reducers such as C-Reduce. If we noticed that C-Reduce missed certain reduction opportunities that resulted in obviously bad reduction results, adding new transformations to handle these cases would be easy due to C-Reduce's high modularity.

2.3.3 The Choice of Language Frontend

Compiler-like source-to-source transformation needs support of a language frontend. Besides doing lexing, parsing, and building an abstract syntax tree (AST) for the input program, the frontend needs to meet several requirements for implementing efficient, effective, and useful source-to-source transformation for test-case reduction.

First, the frontend must support source-to-source transformation. It should provide a way to output a transformed program in the same language as the input program being written, along with all modifications made to it. This requirement is fundamental and obvious; otherwise, there is no ground for performing source-to-source transformation. Second, the frontend should have full-fledged support for language constructs. Incomplete support of language features can limit the use of the reducer—especially if we consider extending the reducer to handle more general codes other than only a fuzzer's outputs. For example, a C language family frontend that provides support of C, C++, and Object C is better than a frontend only targeting the C language. The reducer's workflow can be simplified due to integrating fewer external toolsets, and the starting cost of learning a new frontend framework can be amortized by working with a unified interface. Third, the frontend should facilitate AST-base program analysis. Source-to-source transformation conformable with the language specification needs to perform certain analysis on the AST generated from the input. For instance, renaming a variable requires changing its declared name and also replacing each of its references with the new name. In this case, the frontend should make it easy to traverse the AST to find all the references of the variable being renamed. Fourth, the front-end must not add any extra modification to the source code except for the designated pieces.

Although it may not be straightforward, the last requirement is actually important to implement an effective reduction tool so that it needs more explanation. In general, the bug being triggered by the test case under reduction is transparent to the reducer. The reducer has no knowledge about how the bug would look like, and any small change that it makes to the test could make the bug disappear. In some rare cases, even simple reformatting could turn a bug-triggering test case into one that was unable to trigger the bug any more. Some frontend tools pretty-print transformed code in their own way rather than keeping the original code layout, or implicitly alter code structure at the AST level, e.g., generating a single AST node "0" for the expression "a - a" instead of constructing separate nodes for the operator and its operands.

Having the reducer explicitly control all modifications naturally mimics the process of manual reduction, and makes it easy to examine what a transformation does for the given test. More importantly, it eliminates the possibility that implicit changes performed by the frontend—even if these changes preserve the semantics of the input—would cause the bug behavior to disappear. In fact, one of the key characteristics of C-Reduce is that it is free to change the semantics of the test being reduced as long as the transformation may contribute to a smaller test case while preserving the desired bug-behavior. In other words, C-Reduce has stronger bug-preserving requirement than semantics preserving.

2.3.4 Implementing Compiler-like Source-to-source Transformations

This section describes the implementation details of our compiler-like source-to-source transformation using LLVM's Clang frontend, which meets all the requirements described above. Furthermore, I explain the key components provided by the Clang Frontend to facilitate source-code transformation and the structure of our transformation framework. Several transformation outputs are also presented to demonstrate how those transformations work together to produce a small test case.

2.3.4.1 The Use of Clang Frontend

Clang's modular-library-based architecture simplifies the task of writing a source-to-source transformation tool. In particular, C-Reduce leverages three key features provided by Clang frontend to implement compiler-like source-to-source transformation.

- Clang frontend provides a pre-order, depth-first traversal over the AST using its RecursiveASTVisitor class template. The RecursiveASTVisitor implements a standard C++ technique, *Curiously Recurring Template Pattern* (CRTP), where a user-defined visitor class derives from a template instantiation that has the visitor class as its argument. In most cases, the user-defined visitor class only needs to override interesting visit methods to handle certain types of AST nodes such as types, declarations, and expressions. For example, overriding the VisitVarDecl method allows the user to process all variable declarations, each at a time, in the traversal order defined in the RecursiveASTVisitor. By default, the RecursiveASTVisitor ensures that each AST node built from explicit part of the source code is going to be visited exact once.
- Each AST node has an encoding of its corresponding location in the source code, if the location exists. Furthermore, the SourceManager class provides a way to retrieve the actual string representation of the code starting from a source location. This

facility makes it possible to have fine-grained control over the input source.

- The Rewriter class exposes interfaces for removing, replacing and inserting text based on given source locations. The Rewriter relies on a sophisticated rope data structure [11], which is used for efficiently manipulating text, e.g., insertion, deletion, and concatenation. With the source location information encoded in AST nodes, the Rewriter enables accurate alterations to the input. Moreover, the Rewriter directly modifies the original source code and explicitly controls every single write to the source. In other words, the frontend does not implicitly alter the input, and every modification to the input must have been coded by the user through some interface provided by the Rewriter.

The combination of these features gives a simple but powerful way to implement a source-code rewriting tool, usually in the following steps: visiting interesting AST nodes, constructing the replacing string using certain analysis, obtaining the location to rewrite, and firing the Rewriter to actually conduct rewriting.

2.3.4.2 The Structure of Compiler-like Source-to-source Transformations

Figure 2.7 shows the structure of the compiler-like transformation. The driver is the entrance to the underlying transformation infrastructure. It accepts the input program, interprets the command-line options, and then passes the control to the transformation manager. The transformation manager employs Clang's standard APIs to create the compiler instance and do necessary initializations, e.g., setting correct target information, creating the preprocessor and AST context, specifying the AST consumer, and initializing the source manager. Individual transformations register themselves to the transformation manager through a unique interface. The transformation manager invokes the transformation specified at the command line. Each transformation is a derived instance of Clang's ASTConsumer class, which accepts the parsed AST and performs some actions on it, e.g., visiting certain AST nodes and rewriting the input source code. AST visitation is executed with the RecursiveASTVisitor provided by Clang's runtime. The invoked transformation then performs some analysis along with the AST visitation, determines the part of the source code to be modified, and launches the rewriter to write the changes back to the original code.

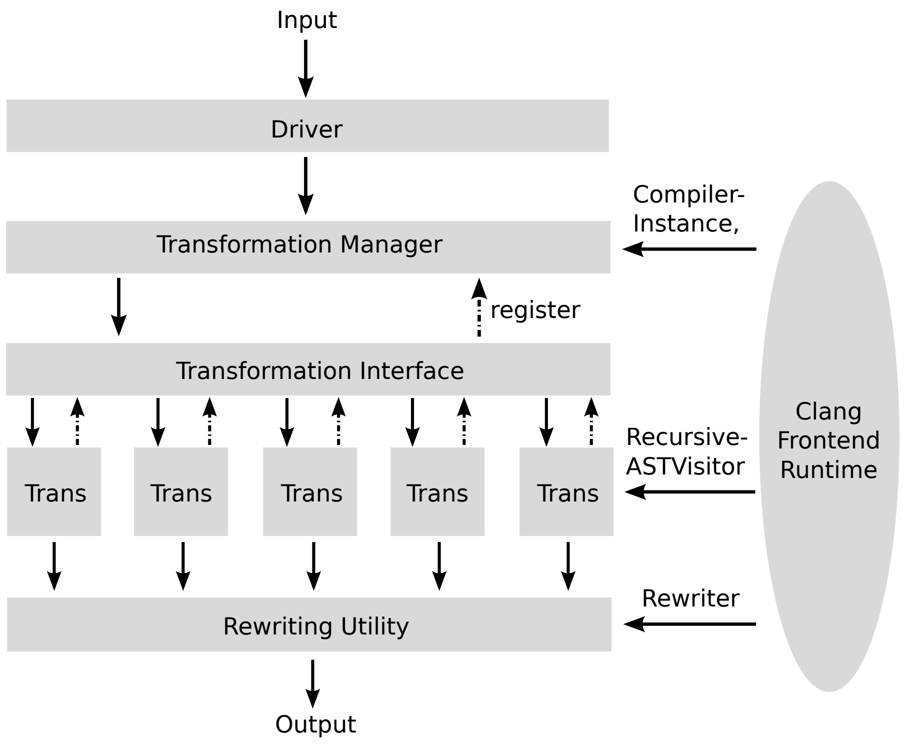

Figure 2.7: The structure of compiler-like source-to-source transformation

2.3.4.3 Kinds of Compiler-like Source-to-source Transformations

In total, C-Reduce has 65 compiler-like source-to-source transformations targeting various aspects of C and C++, including[2]

- removing an unused function, variable, or field in a struct/union;
- shortening the name of a parameter, variable, function, method, or class;
- replacing aggregates with scalars;
- replacing unions with structs;
- reducing the length or the dimension of an array;
- reducing the indirection level of a pointer variable;
- moving a local variable to global scope;
- removing a parameter from a function's declaration and deleting the corresponding argument at each call site of the function while adding a local variable of the same type and name at the function scope;

[2]Among these 65 transformations, two were provided by Konstantin Tokarev.

- factoring a call expression out of a complex expression;
- making a function return void and removing all return statements in the function;
- combining multiple same-typed variable declarations into a single compound declaration;
- inlining small functions;
- performing copy propagation;
- simplifying comma expressions, if-statements, or typedef declarations;
- turning a class template into a class;
- instantiating template parameters;
- replacing template arguments with integer types or values;
- removing an unused parameter of a class template from its declaration while erasing the corresponding template argument from each of its instantiation or specialization;
- removing namespaces;
- removing an initializer from a class constructor; and
- simplifying the class hierarchy.

2.3.5 Examples of Compiler-like Source-to-source Transformation

Figures 2.8–2.12 demonstrate how the compiler-like source-to-source transformation can benefit producing a small test case. The codes were generated during a real C-Reduce run for reducing a test case triggering a crash bug in GCC 4.3.0.[3] In each figure, the code before the transformation is on the left, and the transformed code is on the right. The changes made by the transformation are also highlighted in the transformed code.

The conducted transformations are summarized as follows (in the order of being invoked during the reduction):

1. Figure 2.8: a simple-inliner pass was performed to replace the invocation of function *foo* with its body;
2. Figure 2.9: reduce-pointer-level pass changed pointer *g7* to a scalar type at line 7 and modified the corresponding pointer dereference at line 17;
3. Figure 2.10: copy-propagation pass replaced variable *g6* with variable *tmp* at line 15;

[3]For better presentation, the codes were reformatted, and some intermediate results transformed by line-based delta and peephole passes were removed.

```
unsigned char g1;
int g2;
int g3;
int foo(void)
{
  return 0 % 0;
}
int g4;
char g5;
int g6;
int *g7;
void bar(void)
{




  g6 = foo();
  unsigned char l1 = g6;
  g5 = g1 ? l1 : l1 % 0;
  *g7 = g5;
  for (;;) ;
}
```

(a)

```
unsigned char g1;
int g2;
int g3;
~~int foo(void)~~
~~{~~
~~  return 0 % 0;~~
~~}~~
int g4;
char g5;
int g6;
int *g7;
void bar(void)
{
  int tmp;
  {
    tmp = 0 % 0;
  }
  g6 = tmp;
  unsigned char l1 = g6;
  g5 = g1 ? l1 : l1 % 0;
  *g7 = g5;
  for (;;) ;
}
```

(b)

Figure 2.8: Code transformation by applying simple-inliner pass. (a) Before transformation; (b) After transformation.

```
unsigned char g1;
int g2;
int g3;
int g4;
char g5;
int g6;
int *g7;
void bar(void)
{
  int tmp;
  {
    tmp = 0 % 0;
  }
  g6 = tmp;
  unsigned char l1 = g6;
  g5 = g1 ? l1 : l1 % 0;
  *g7 = g5;
  for (;;) ;
}
```

(a)

```
unsigned char g1;
int g2;
int g3;
int g4;
char g5;
int g6;
int g7;
void bar(void)
{
  int tmp;
  {
    tmp = 0 % 0;
  }
  g6 = tmp;
  unsigned char l1 = g6;
  g5 = g1 ? l1 : l1 % 0;
  g7 = g5;
  for (;;) ;
}
```

(b)

Figure 2.9: Code transformation by applying reduce-pointer-level pass. (a) Before transformation; (b) After transformation.

```c
unsigned char g1;
int g2;
int g3;
int g4;
char g5;
int g6;
int g7;
void bar(void)
{
  int tmp;
  {
    tmp = 0 % 0;
  }
  g6 = tmp;
  unsigned char l1 = g6;
  g5 = g1 ? l1 : l1 % 0;
  g7 = g5;
  for (;;) ;
}
```

(a)

```c
unsigned char g1;
int g2;
int g3;
int g4;
char g5;
int g6;
int g7;
void bar(void)
{
  int tmp;
  {
    tmp = 0 % 0;
  }
  g6 = tmp;
  unsigned char l1 = tmp;
  g5 = g1 ? l1 : l1 % 0;
  g7 = g5;
  for (;;) ;
}
```

(b)

Figure 2.10: Code transformation by applying copy-propagation pass. (a) Before transformation; (b) After transformation.

```
unsigned char g1;
int g2;
int g3;
int g4;
char g5;
int g6;
int g7;
void bar(void)
{
  int tmp;
  {
    tmp = 0 % 0;
  }
  g6 = tmp;
  unsigned char l1 = tmp;
  g5 = g1 ? l1 : l1 % 0;
  g7 = g5;
  for (;;) ;
}
```

(a)

```
unsigned char g1;
~~int g2;~~
~~int g3;~~
~~int g4;~~
char g5;
int g6;
int g7;
void bar(void)
{
  int tmp;
  {
    tmp = 0 % 0;
  }
  g6 = tmp;
  unsigned char l1 = tmp;
  g5 = g1 ? l1 : l1 % 0;
  g7 = g5;
  for (;;) ;
}
```

(b)

Figure 2.11: Code transformation by applying remove-unused-var pass. (a) Before transformation; (b) After transformation.

```
1  unsigned char g1;
2  char g5;
3  int g6;
4  int g7;
5  void bar(void)
6  {
7    int tmp;
8    {
9      tmp = 0 % 0;
10   }
11   g6 = tmp;
12   unsigned char l1 = tmp;
13   g5 = g1 ? l1 : l1 % 0;
14   g7 = g5;
15   for (;;) ;
16 }
```

(a)

```
1  int g1;
2  char g5;
3  ~~int g6;~~
4  ~~int g7;~~
5  void bar(~~void~~)
6  {
7    int tmp;
8    ~~{~~
9      tmp = 0 % 0;
10   ~~}~~
11   ~~g6 = tmp;~~
12   unsigned char l1 = tmp;
13   g5 = g1 ? 0 : l1 % 0;
14   ~~g7 = g5;~~
15   for (;;) ;
16 }
```

(b)

Figure 2.12: Code transformation by applying some line-based and peephole passes. (a) Before transformation; (b) After transformation.

4. Figure 2.11: three unused variables, *g2*, *g3*, and *g4*, were removed by three consecutive remove-unused-var passes;
5. Figure 2.12: several line-based and peephole passes simplified the declared type of variable *g1* at line 1, simplified the signature of function *bar* at line 5, removed the balanced curly parentheses at lines 8 and 10, replaced variable *l1* with constant 0 in the ternary operator at line 13, and removed lines 3, 4, 11, and 14.

CHAPTER 3

EVALUATION OF C-REDUCE

This Chapter evaluates C-Reduce on reducing a large number of test cases, which fall into three classes:

- 2,501 Csmith-generated test cases triggering crash bugs in GCC 4.3.0;
- 1,298 Csmith-generated test cases triggering wrong-code bugs in GCC 4.3.0;
- 10 test cases harvested from GCC and LLVM bugzilla databases, each of which crashes a version of GCC or LLVM.

The last type of test cases are used to demonstrate that C-Reduce is not only useful for reducing randomly-generated test cases—it can effectively reduce test cases included in compiler bug reports submitted by others. We also compared the outputs of C-Reduce against those produced by running Berkeley delta—the most commonly-used existing reducer by compiler developers—on the same set of bug-inducing test cases. C-Reduce's outputs were, on average, more than 30 times smaller than those produced by Berkeley delta. This Chapter reuses some text from our PLDI 2012 paper [59], a joint work with John Regehr, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang.

3.1 Test Cases

To thoroughly evaluate test-case reducers, one needs a number of test inputs that trigger compiler crash bugs and wrong-code bugs. Moreover, these test cases should map to a diverse collection of underlying defects. We obtained our test cases from two sources: Csmith and real-life bug reports.

3.1.1 Csmith-generated Test Cases

All test cases in this category were generated using the default configuration of Csmith 2.1.0 [74], which used swarm testing [28] to improve bug detection. Each program generated by Csmith was compiled by GCC 4.3.0 at several optimization levels: -O0, -O1, -O2,

-Os, and -O3.

After running Csmith for a few days, we collected 2,501 test cases that crashed the compiler and 1,298 test cases that triggered wrong-code bugs in GCC, respectively. Crash bugs were detected by inspecting the exit code of the compiler being tested—any nonzero value indicated a crash bug. Wrong-code bugs were detected using *differential testing* [50]: a wrong-code bug was revealed if the result of the compiler's output was different from the result produced by a reference compiler's output. In our experiments, the reference compiler was simulated by running GCC 4.6.0 and Clang 3.1 at their lowest optimization levels and considering only test cases where the executables generated by these compilers produce the same results.

3.1.2 Test Cases from Application Code

To evaluate the effectiveness of C-Reduce on reducing C/C++ programs other than test cases generated by Csmith, we harvested the bugzilla databases of LLVM and GCC to collect 10 large bug-inducing test cases included in real-life bug reports, five for LLVM and five for GCC, respectively. Each of those test cases crashed a version of LLVM or GCC; three of them are C programs and the others are C++ codes. Tables 3.1–3.3 summarize each of those test cases, including

- the language in which the test case is written,
- the compiler version that exhibits the faulty behavior,
- the compiler flags that trigger the bug,
- the crash string thrown by the compiler, and
- the URL of the bug report where the original test case was attached.

3.2 Nondeterminism in Test-case Reduction

Nondeterministic execution of the system under test can cause test-case reduction to fail. In our experience, the most common sources of nondeterminism are memory-safety bugs interacting with address space layout randomization (ASLR) and resource-exhaustion conditions such as timeouts and memory limits.

ASLR is a technique used in operating systems to protect the host computer from certain kinds of attacks, e.g., buffer overflow attacks. By randomly arranging the locations of a program's data segments such as stack and heap in a process's address space, ASLR makes

Table 3.1: Test cases that triggered crash bugs in various versions of LLVM and GCC and were attached in bug reports submitted by others

Test Case	Language	Compiler	Flags
GCC1	C++	gcc-r199760	-O2
GCC2	C++	gcc-4.6.0	-O3
GCC3	C++	gcc-r132142	-O
GCC4	C++	gcc-r155401	-g
GCC5	C	gcc-r145254	-c
LLVM1	C++	llvm-r116696	-m32 -O2
LLVM2	C	llvm-r171307	-cc1 -emit-obj -O3
LLVM3	C	llvm-r175960	-emit-obj -sys-header-deps -D "HAVE_CONFIG_H" -O2 -fsanitize=bounds
LLVM4	C++	llvm-r145532	-emit-obj -O2 -std=gnu++0x
LLVM5	C	llvm-r116097	-c

Table 3.2: Compiler crash strings uncovered by those test cases attached in bug reports submitted by others

Test Case	Crash String
GCC1	`internal compiler error: in curr_insn_transform, at lra-constraints.c:3007`
GCC2	`internal compiler error: Segmentation fault`
GCC3	`internal compiler error: in copy_to_mode_reg, at explow.c:621`
GCC4	`internal compiler error: tree check: accessed elt 3 of tree_vec with 2 elts in tsubst, at cp/pt.c:9860`
GCC5	`internal compiler error: tree check: expected integer_cst, have nop_expr in tree_int_cst_lt, at tree.c:4963`
LLVM1	`Assertion 'Node2Index[SU->NodeNum] > Node2Index[I->getSUnit()->NodeNum] && "Wrong topological sorting" failed`
LLVM2	`Assertion 'BitWidth == RHS.BitWidth && "Comparison requires equal bit widths"' failed`
LLVM3	`Assertion 'isa<X>(Val) && "cast<Ty>() argument of incompatible type!"' failed`
LLVM4	`Assertion 'MD->isVirtual() && "Method is not virtual!"' failed`
LLVM5	`Assertion 'NextFieldOffsetInBytes <= FieldOffsetInBytes && "Field offset mismatch!"' failed`

Table 3.3: The URLs of the bug reports where the original test cases were attached

Test Case	Bug Report URL
GCC1	`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=57447`
GCC2	`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=51737`
GCC3	`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=35056`
GCC4	`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43087`
GCC5	`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=41182`
LLVM1	`http://llvm.org/bugs/show_bug.cgi?id=8404`
LLVM2	`http://llvm.org/bugs/show_bug.cgi?id=14761`
LLVM3	`http://llvm.org/bugs/show_bug.cgi?id=15338`
LLVM4	`http://llvm.org/bugs/show_bug.cgi?id=12219`
LLVM5	`http://llvm.org/bugs/show_bug.cgi?id=6955`

it difficult for hackers to predicate the target address since the address can vary in each program run. However, ASLR also has the side-effect of making some compiler bugs to occur nondeterministically, e.g., certain compiler bugs caused by memory corruption. In practice, we just simply disabled ASLR in the Ubuntu box for our experiments.

Nondeterminism caused by resource-exhaustion is hard to avoid. For example, we have seen cases where C-Reduce transformed an original test case triggering a crash bug into one that triggered a compiler-hanging bug, i.e., the compiler never terminated when compiling the input. Our pragmatic approach is to set a reasonable timeout value for running the compiler or the generated executable. For example, a good timeout value for compiler's execution would be one that is slightly larger than the runtime of compiling the unreduced test case.

3.3 Evaluating Reducers

We ran C-Reduce and Berkeley delta on our corpus of bug-inducing test cases: Csmith-generated ones and those attached in the bug reports. The inputs to C-Reduce are composed of the test case to be reduced and a script that determines whether a variant is successful. In addition to the test case and the script, Berkeley delta requires a "level" parameter that specifies the nesting level of braces where its *topformflat* tool performs line breaks. As suggested by the tool's documentation, we ran the main delta script twice at level 0, twice at level 1, twice at level 2, and twice at level 10 for our Berkeley delta reductions. We used Frama-C as our external validity checker for wrong-code bugs because Frama-C imposed less runtime overhead than KCC based on our experience.

Table 3.4 summarizes the results of reducing all Csmith-generated bug-inducing test cases using Berkeley delta and C-Reduce. In each category, the size was averaged over all test cases.

3.3.1 Reduction Results of Csmith-generated Bug-inducing Test Cases

For our corpus of Csmith-generated test cases, C-Reduce produced much better outputs than Berkeley delta did—more than 30 times smaller on average. The original sizes were calculated on the preprocessed files, which were the inputs to the reducers. For the crash bugs, C-Reduce was able to produce, on average, test cases of 107 bytes from the original test cases whose average size was 121 KB. For our wrong-code bugs, the final outputs had an average size of 236 bytes, whereas the unreduced test cases were of 82 KB on average. Reduction ratios for crash bugs were higher than that for wrong-code bugs because the reducers could conduct more aggressive transformations on crash test cases without performing any validity check. Figures 3.1 and 3.2 show the median-sized reduced test cases produced by C-Reduce for the crash and wrong-code test cases, respectively. They are reportable test cases—we can copy and paste the codes into bug reports without any manual modification.

3.3.2 Reduction Results of Reported Test Cases

C-Reduce was originally motivated by reducing bug-triggering test cases generated by Csmith. It has been substantially improved to reduce large C++ programs that induce

Table 3.4: Averaged sizes (in bytes) of reduced test cases

		Output Size		**Reduction Ratio**	
Bug Kind	**Original Size**	**Berkeley Delta**	**C-Reduce**	**Berkeley Delta**	**C-Reduce**
Csmith-generated Crash	121,430	2,697	107	107	1,134
Csmith-generated Wrong-code	82,058	8,686	241	9	340
Reported Crash	960,552	34,007	335	28	2,867

```
struct S0
{
   volatile int f4:1
};
static struct S0 a;
void
main ()
{
   printf ("%d\n", a.f4);
}
```

Figure 3.1: The median-sized reduced test case produced by C-Reduce for crash bugs

```
int printf (const char *, ...);
unsigned short *a;
char b;
char *c = &b;
int d;
int
main ()
{
   unsigned short **e = &a, **f = &a;
   d = -(e == f);
   *c = d;
   printf ("%d\n", b);
   return 0;
}
```

Figure 3.2: The median-sized reduced test case produced by C-Reduce for wrong-code bugs

compiler bugs. The newly added transformations deal with certain complex C++ features such as templates, classes, and namespaces, which are obstacles for Berkeley delta. With these transformations that are specific to C++, C-Reduce can effectively reduce megabyte C++ programs into small ones of only several hundred bytes, even for heavily-templated codes. Figure 3.3 shows the results of reducing 10 reported test cases that crashed a variety of GCC and LLVM versions. Among these tests, six are large C++ programs: gcc1–gcc4, llvm1, and llvm4. Figure 3.4 is the median-sized C++ test case produced by C-Reduce from a 2.4M bug-inducing program. Gcc4 was the worst result given by Berkeley delta, which stopped with the output of 287K. It is not uncommon to see that Berkeley delta's outputs remain hundreds of kilobytes, whereas C-Reduce's output was only 549 bytes.

3.4 Discussion

In many cases, C-Reduce is capable of producing small test cases nearly as good as those produced by skilled developers, but extremely experienced programmers can still outperform C-Reduce. C-Reduce can be improved by observing the way that experienced developers conduct test-case reduction. Furthermore, C-Reduce's support for C++ has room for improvement. We occasionally see that C-Reduce fails to create small-enough outputs for some complex C++ codes. For example, C-Reduce needs better support for processing inner classes and typenames. Last, C-Reduce is slow at reducing some large C++ programs, for which C-Reduce can take several hours to finish reduction. Improving the efficiency of C-Reduce on large C++ codes is another piece of future work.

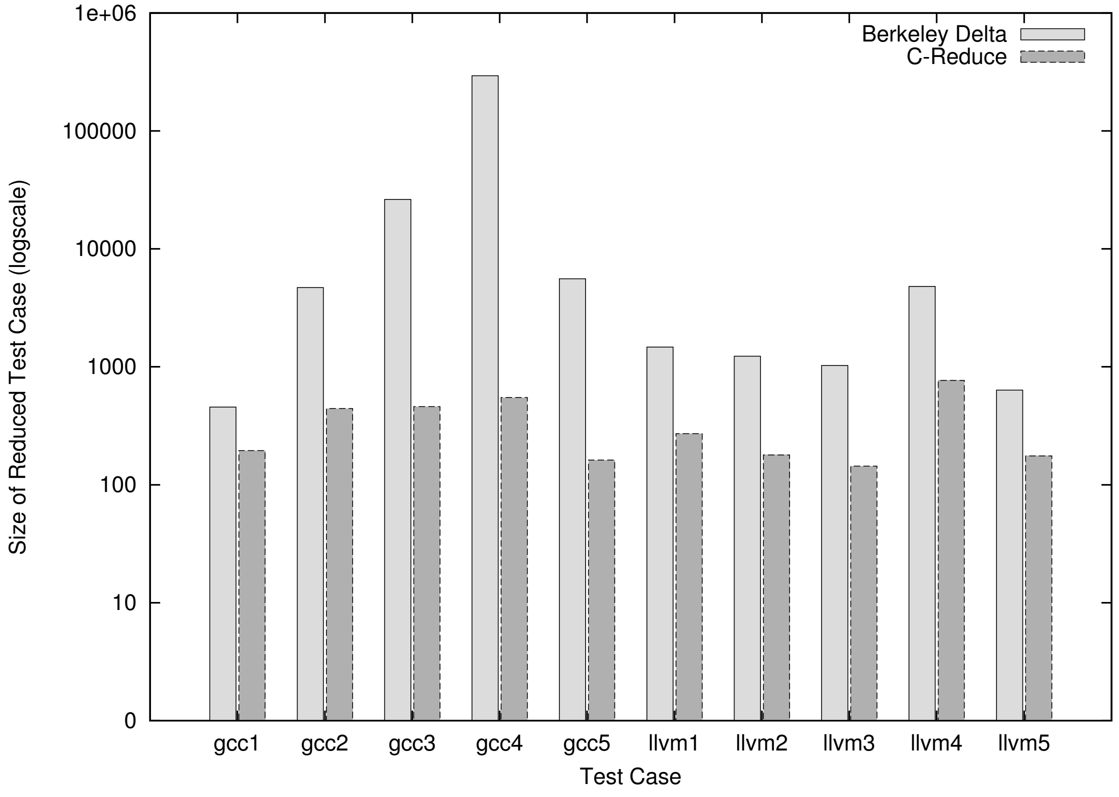

Figure 3.3: Sizes (in bytes) of the test cases reduced by Berkeley delta and C-Reduce

```
template <class T> class A {
public:
  typedef T element_type;
  ~A() { intrusive_ptr_release(px); }
  T *px;
};

template <typename T> struct B {
  friend void intrusive_ptr_release(T *p1) {
    if (p1->ref_count_)
      delete p1;
  }
  int ref_count_;
};

struct C;
struct D;
struct F : B<F> {
  A<F> parent;
  A<C> document;
};
struct C : B<C> {
  A<D> current_section;
};
struct D : B<D> {
  A<D> parent;
};
void start_file() { A<F> a; }
```

Figure 3.4: The median-sized reduced test case produced by C-Reduce for reported bugs triggered by C++ programs

CHAPTER 4

RELATED WORK FOR TEST-CASE REDUCTION

This section surveys the related work for test-case reduction techniques. There is a surprisingly small amount of research about test-case reduction. General reduction approaches are also discussed in this section.

4.1 Test-case Reduction for C Programs

Test-case reduction for C programs has been studied in previous work. Bugfind, developed by Caron and Darnell [16], is able to narrow the cause of a wrong-code bug down to a single C file and search the lowest optimization level where the bug occurs. Bugfind works at the file granularity and only relies on object files to determine whether the program is miscompiled or not and therefore, it can be applied to programs written in languages other than C. The output of Bugfind consists of all files—one or more—that induce the miscompilation.

McKeeman's random C program generator for testing C compilers provides an automated test-case reducer that runs at a finer granularity than Bugfind [50]. McKeeman's test-case reducer has 23 different transformations, including removing statements and declarations, simplifying constants to 1s, deleting balanced braces, etc. The reducer systematically applies all of these transformations on the test under reduction until the test cannot be changed further. It is able to reduce randomly-generated C programs of 500 to 600 lines to a few lines. McKeeman's reducer appears to be similar to C-Reduce's peephole passes. C-Reduce, on the other hand, has a richer set of transformations than those provided by McKeeman's tool. To the best of my knowledge, C-Reduce is the first test-case reducer where compiler-like transformations are used to create reportable bug-triggering test cases for C/C++ programs.

C-Reduce is inspired by delta debugging algorithms, formalized by Zeller and Hildebrandt [76]. In their work, they abstracted the idea of delta debugging and implemented two algorithms, *dd* and *ddmin*. McPeak and Wilkerson developed Berkeley delta [51], a commonly-used delta debugging tool by compiler developers. Berkeley delta is line-based and implements the *ddmin* algorithm. With the help of an external tool, *topformflat*, Berkeley delta is able to to reduce structured input more efficiently than character-based *ddmin*. We created C-Reduce as a result of our dissatisfaction with Berkeley delta's output, which often requires a certain amount of manual reduction effort to produce final reportable tests. Misherghi and Su developed Hierarchical Delta Debugging (HDD) [52]. HDD was designed to be more suitable for reducing structured input and showed its advantage over the original *ddmin* algorithm by producing better reduced output for a number of failure-inducing test cases written in C and XML.

In contrast to Berkeley delta and HDD, C-Reduce exploits deeper knowledge about the language syntax and semantics to conduct reduction, allows nonlocalized alterations to test input, and utilizes more reduction opportunities. As a consequence, C-Reduce produces better output [59].

4.2 Other Test-case Reduction Techniques

Fuzzing tools are often paired with their own reducers that try to minimize the bug-triggering test to reproduce the fault and save human effort. QuickCheck is a lightweight random testing tool for Haskell programs, developed by Claessen and Hughes [18]. The tool was extended to include a "smaller" method, which is used to find a smaller counterexample demonstrating the faulty behavior for the counterexample being found, e.g., by taking a subtree of the original counterexample if it was of a tree-type data structure. Brummayer and Biere proposed a technique to test SMT solvers using grammar-based blackbox fuzzing [12,13]. When a randomly-generated formula triggered a bug in the SMT solver under test, a customized delta debugger was invoked to distill the failure-inducing formula into its essence. The customized delta debugger adapted ideas from HDD [71]. Having the knowledge of formula structures and types benefits the speed of conducting reduction and can arrive at better reduction results.

The CERT Basic Fuzzing Framework, or BFF, is a mutation-based fuzzing tool that is intended for finding bugs in a variety of applications [1]. BFF has been used to uncover a number of bugs in production-quality software systems such as Adobe Reader, Apple QuickTime, and FFmpeg. One of the highlighted features of BFF is its test-case reducer that automatically minimized differences from a seed file [31]. Lithium [62], created by the author of jsfunfuzz [61], is able to reduce jsfunfuzz-generated crash test cases of 3000 lines down to the ones of 3–10 lines in several minutes.

LLVM has a tool, Bugpoint [5], which works on LLVM bitcode and can be used for reducing tests that trigger either crash or wrong-code bugs in the compiler's optimizers or code generators. Given a test input, Bugpoint tries to reduce both the list of LLVM passes and the size of the test. Bugpoint is useful to narrow down the causes of the problems in LLVM's passes.

Lei and Andrews [41] presented an approach to reducing the sequences of method calls of failing test cases based on Zeller and Hildebrandt's delta debugging algorithm. In the work of Leitner et al. [42], static slicing is combined with delta debugging to minimize the sequence of failure-inducing method calls. This later approach was able to minimize failing unit test cases by 96% on average in a case study on the EiffelBase library. It was shown to be more efficient—11 times faster—than doing delta debugging alone based on their case study. JINSI, presented by Burger and Zeller [14], aims for minimizing reproduction of software failures in Java programs. Different from previous delta debugging techniques, which require both successful and failing executions, JINSI only takes a single failing run, where JINSI minimizes the object interactions to the sequence of calls that are relevant to the failure. In their case study of 17 real-life bugs, the combination of delta debugging and dynamic slicing on method calls in JINSI resulted in greater improvements than using only dynamic slicing—the search space was reduced to 13.7% of the dynamic slice.

Artho presented Iterative Delta Debugging [8], or IDD, where delta debugging was extended to search the revision history to generate a minimal patch that introduced the defect. Yu et al. [75] evaluated the effectiveness of delta debugging in software evolution, where a number of regression tests from some open source programs were used as the basis of their study. More recently, Zhang [77] proposed SimpleTest, a technique that simplifies the test at the semantic level. Rather than simply removing components from the test, SimpleTest creates a smaller test that preserves certain semantic properties.

CHAPTER 5

TAMING COMPILER FUZZERS

Test-case reduction distills a large failure-inducing test case into its essence that only contains the part relevant to the bug. Creating small test cases is the first step towards making compiler fuzzers more useful—small test cases are easier to debug than their large counterparts. Another problem of using fuzzers to test compilers is that they repeatedly and indiscriminately find bugs that are already known to developers or may not be severe enough to fix right away. An overnight run of a fuzzer may result in hundreds or thousands of failure-inducing test cases, which may trigger only a few bugs. Moreover, these bugs do not uniformly distribute over the test cases—some bugs tend to be triggered much more often than others. It can be time-consuming for fuzzer users to sift through a large collection of failure-inducing test cases to suppress duplicates and identify tests that trigger interesting bugs. We formulate this problem as the *fuzzer taming problem*: given a potentially large number of random test cases that trigger failures, rank them in such a way that diverse, interesting test cases are highly ranked. This section describes our approach to solving the fuzzer taming problem. Most of the text of this Chapter is from our PLDI 2013 paper [17], which is a joint work with Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr.[1]

5.1 Background

A typical workflow for using a fuzzer is as follows:

1. start running the fuzzer against the version of the compiler to be tested;
2. go to work on other things;
3. after a day or two, look through the new failure-inducing test cases, creating a bug

[1]The definitive version was published in *Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, USA, Jun. 2013. http://doi.acm.org/10.1145/2491956.2462173

report for each that is novel and important.

Step 3 can be laborious and unrewarding. We know of several industrial compiler developers who stopped using Csmith not because it stopped finding bugs, but because step 3 became uneconomical. For example, Figure 5.1 shows three reduced Csmith-generated test cases that all trigger crash bugs in GCC 4.3.0. By just looking at the text of these test cases, one would guess that the codes in Figures 5.1(a) and 5.1(b) trigger the same bug, whereas the code in Figure 5.1(c) triggers another one. In this case, it would be easy for the user to decide to report two test cases, e.g., the first and the third ones, to GCC developers. However, manually categorizing failure-inducing test cases can become quite difficult when the number of the tests to be investigated is large. It would be very hard for the user to manually identify the underlying bugs and suppress duplicates for hundreds or thousands of test cases.

One solution to this problem would be to

1. report one bug-triggering test case,
2. wait until the triggered bug has been fixed,
3. run the remaining test cases against the bug-fixed version of the compiler,
4. save tests that still exhibit faulty behaviors in the compiler,
5. go to step (1).

These steps are repeated until there are no more failure-inducing tests. However, in reality, bugs are prioritized by their severity. When resources are limited and deadlines may be inflexible, low-priority bugs can linger unfixed for months or years. For example, in June, 2012, we found more than 2,000 open bugs in GCC's bug database, considering priorities P1, P2, and P3 and considering only bugs of "normal" or higher severity. The median-aged bug in this list was well over two years old. So, this simple report-and-then-wait solution does not work in practice because we do not have control over bug prioritization.

5.2 The Fuzzer Taming Problem

Thus far, little research has addressed the problem of making fuzzer output more useful to developers. In a blog entry, Ruderman, the original author of the *jsfunfuzz* tool, reports using a variety of heuristics to avoid looking at test cases that trigger known bugs, such as turning off features in the test-case generator and using tools like *grep* to filter out test

```
char a;
int b[];
void
fn1 ()
{
   a = 0;
   for (; a < 7; a += 1)
      b[7 + a] = 0;
}
```

(a)

```
int a[][7];
char b;
void
fn1 ()
{
   b = 0;
   for (; b < 7; b += 1)
      a[1][b] = 0;
}
```

(b)

```
char a;
int b;
void
fn1 ()
{
   b = (unsigned) ~a >> 9;
}
```

(c)

Figure 5.1: Three example test cases that crash GCC 4.3.0. (a) The first example test; (b) The second example test; (c) The third example test.

cases triggering bugs that have predictable symptoms [64]. During testing of mission file systems at NASA [27], Groce et al. used hand-tuned rules to avoid repeatedly finding the same bugs during random testing.

We claim that much more sophisticated automation is feasible and useful. We characterize the *fuzzer taming problem* as follows:

> Given a potentially large collection of test cases, each of which triggers a bug, rank them in such a way that test cases triggering distinct bugs are early in the list.
>
> **Sub-problem:** If there are test cases that trigger bugs previously flagged as undesirable, place them late in the list.

Ideally, for a collection of test cases that trigger N distinct bugs (none of which have been flagged as undesirable), each of the first N test cases in the list would trigger a different bug. In practice, perfection is unattainable because the problem is hard and also because there is some subjectivity in what constitutes "distinct bugs." Thus, our goal is simply to improve as much as possible upon the default situation where test cases are presented to users in effectively random order. Our rank-ordering approach was suggested by the prevalence of ranking approaches for presenting alarms produced by static analyses to users [36, 37].

Taming a fuzzer differs from previous efforts in duplicate bug detection [67, 68, 72] because user-supplied metadata is not available: we must rely solely on information from failure-inducing test cases. Compared to previous work on dealing with software containing multiple bugs [33, 47, 57], our work differs in three major parts:

- the methods used: ranking bugs as opposed to clustering that places tests into homogeneous groups and chooses a representative test from each group;
- the kinds of inputs to the machine learning algorithms: we exploit diverse sources of information about bug-triggering test cases, including features of the test case itself, features from execution of the compiler on the test case, and features from the compiler's output, as opposed to just predicates or coverage information; and
- our special goal: we aim for taming a fuzzer, where a large number of randomly-generated test cases exhibit only a few bugs, some of which are triggered much more often than others.

5.3 Our Approach to Taming Compiler Fuzzers

This section describes our approach to taming compiler fuzzers and gives an overview of the tools implementing it. First, I present several important concepts used throughout the rest of this chapter and next two chapters.

5.3.1 Definitions

A *fault* or *bug* in a compiler is a flaw in its implementation. When the execution of a compiler is influenced by a fault—e.g., by wrong or missing code—the result may be an *error* that leads to a *failure* detected by a test oracle. We are primarily concerned with two kinds of failures: (1) compilation or interpretation that fails to follow the semantics of the input source code and (2) compiler crashes. The goal of a compiler fuzzer is to discover source programs—test cases—that lead to these failures. The goal of a *fuzzer tamer* is to rank failure-inducing test cases such that any prefix of the ranked list triggers as many different faults as possible. Faults are not directly observable, but a fuzzer tamer can estimate which test cases are related by a common fault by making an assumption: the more "similar" two test cases, or two executions of the compiler on those test cases, the more likely they are to stem from the same fault [48].

A *distance function* maps any pair of test cases to a real number that serves as a measure of similarity. This is useful because our goal is to present fuzzer users with a collection of highly dissimilar test cases. Because there are many ways in which two test cases can be similar to each other—e.g., they can be textually similar, cause similar failure output, or lead to similar executions of the compiler—our work is based on several distance functions.

Some of our distance functions rely on *features* extracted from various sources relevant to the failure-inducing test case such as test-case text, execution trace of the compiler running on the test case, and the compiler's crash strings. Features capture certain characteristics of the source being described. Each feature can be represented as a $\langle$*name, value*$\rangle$ pair. For example, we can define a feature $\langle$*has_global, value*$\rangle$ to express the appearance of global variables in a test case—feature value "0" means that the test does not have any global variable, whereas "1" means otherwise. Similarly, based on the compiler's profiling data, a feature, $\langle$*function_name, number*$\rangle$ can represent the number of times that a particular function is invoked. A set of features extracted from a source forms a *feature vector*, which summarizes the source.

Clustering refers to a general concept of placing a set of objects into homogeneous groups, i.e., clusters, where objects in the same cluster are more similar to each other than to those in other clusters. Similarity between objects can be measured in various ways, e.g., using the distance between objects computed by certain distance function. Since clustering is a general term, it can be implemented in different algorithms such as K-means [29].

5.3.2 Ranking Test Cases

An obvious approach to tackling the fuzzer taming problem is clustering: given a set of feature vectors extracted from the bug-inducing test cases, apply a clustering algorithm to generate clusters where the test cases in the same cluster are more similar to each other than to those in other clusters and then select one test case from each cluster. Our initial attempt was to use a clustering algorithm [56] to cluster test cases, but we quickly found some drawbacks of using clustering. First, clustering is computationally expensive, especially when feature vectors are large. Second, clustering results can vary a lot depending on the parameters passed to the clustering algorithm. For example, some of the parameters of the clustering algorithm can greatly affect the final number of clusters being generated. Since we have no a priori knowledge about how many distinct bugs are triggered by the test cases, it is hard to determine a "right" set of parameters. Moreover, even if we can tune parameters for one collection of test cases, these parameters could perform badly for other collections.

Rather than relying on a certain clustering algorithm, a better approach to solving the fuzzer taming problem is by ranking, which is based on the following idea:

> **Hypothesis 1:** If we (1) define a distance function between test cases that appropriately captures their static and dynamic characteristics and then (2) sort the list of test cases in *furthest point first* (FPF) order, then the resulting list will constitute a usefully approximate solution to the fuzzer taming problem.

If this hypothesis holds, the fuzzer taming problem is reduced to defining an appropriate distance function. The FPF ordering is one where each point in the list is the one that maximizes the distance to the nearest of all previously listed elements; it can be computed using a greedy algorithm [26]. We use FPF to ensure that diverse test cases appear early in the list. Conversely, collections of highly similar test cases will be found towards the end of the list.

Our approach to ignoring known bugs is based on the premise that fuzzer users will have labeled some test cases as exemplifying these bugs.

Hypothesis 2: We can lower the rank of test cases corresponding to bugs that are known to be uninteresting by "seeding" the FPF computation with the set of test cases that are labeled as uninteresting.

Thus, the most highly ranked test case will be the one maximizing its minimum distance from any labeled test case.

5.3.3 Distance Functions and Features

The fundamental problem in defining a distance function that will produce good fuzzer taming results is that we do not know what the trigger for a generic compiler bug looks like. For example, one C compiler bug might be triggered by a struct with a certain sequence of bitfields; another bug might be triggered by a large number of local variables, which causes the register allocator to spill. Our solution to this fundamental ambiguity has been to define a variety of distance functions, each of which we believe will usefully capture some kinds of bug triggers.

A subsequent question in using distance functions is what are the "appropriate" features passed to various distance functions? There is no obvious answer to this question. Many features can be abstracted to represent compiler bugs due to the high diversity of bugs. Furthermore, it is difficult to foresee which features would be important to discriminate one bug from others. For example, test cases that trigger the same bug can have high textual likelihood, and then text-based features are suitable for identifying these test cases or can lead to very similar execution traces in the compiler being tested and therefore, features related to the compiler execution path are likely the right choice. Nevertheless, it is necessary to investigate different feature metrics—separately and together—in terms of their effectiveness on bug discrimination.

5.3.3.1 Levenshtein Distance

The Levenshtein distance [45], or edit distance, between two strings is the smallest number of character additions, deletions, and replacements that suffices to turn one string into the other. For every pair of test cases, we compute the Levenshtein distance between the following, all of which can be treated as plain text strings:

- the test cases themselves;
- the output of the compiler as it crashes, if it exists; and
- the output of Valgrind [54] on a failing execution (if any).

Computing Levenshtein distance requires time proportional to the product of the string lengths, but the constant factor is small (a few tens of instructions), so it is reasonably efficient in practice.

5.3.3.2 Euclidean Distance

Many aspects of failure-inducing test cases, and of executions of compilers on these test cases, lend themselves to summarization in the form of feature vectors, which we can use to compute the Eclidean distance between each of them. Given two n-element vectors v_1 and v_2, the Euclidean distance between them is

$$\sqrt{\sum_{i=1..n} (v_1[i] - v_2[i])^2}$$

Features can be composed of pure tokens appearing in test cases. For example, Figure 5.2 shows a reduced test case that triggers a crash bug in GCC 4.3.0. Lexing this code gives 13 tokens, and a feature vector based on these tokens contains 13 nonzero elements. The overall vector contains one element for every token that occurs in at least one test case, but which does not occur in every test case, out of a batch of test cases that is being processed by the fuzzer tamer. The elements in the vector are based on the number of appearances of each token in the test case.

For some bugs, lexical features can be used to reliably rule out test cases that cannot trigger that bug. However, a token-wise feature vector only captures the content in the test case but ignores its structure and meaning. In other words, we may need to characterize the syntactic rules and semantics encoded in the test case. For this purpose, we implemented a C-Feature extractor in Clang front-end. The tool currently extracts 45 features including

- basic syntactic components such as types, statement classes, and operator kinds;
- syntactic rules specific to aggregates such as packed structs and structs with bit-fields;
- obvious divide-by-zero operations; and

```
int a;
const volatile long b;
void
main ()
{
   a = b;
}
```

Figure 5.2: A test case that has 13 tokens

- some kinds of infinite loops that can be detected statically.

In addition to constructing vectors from test cases, we also constructed feature vectors from compiler executions. For example, the *function coverage* of a compiler is a list of the functions that it executes while compiling a test case. The overall feature vector for function coverage contains an element for every function executed while compiling at least one test case, but that is not executed while compiling all test cases. As with token-based vectors, the vector elements are based on how many times each function executed. We created vectors based on the following items:

- functions covered,
- lines covered,
- tokens in the compiler's output as it crashes (if any), and
- tokens in output from Valgrind (if any).

Function and line coverage data were obtained using Gcov [3]. In the latter two cases, we used the same tokenization as with test cases (treating output from the execution as a text document), except that in the case of Valgrind we abstracted some nonnull memory addresses to a generic ADDRESS token. The overall hypothesis is that most bugs will exhibit some kind of dynamic signature that will reveal itself in one or more kinds of feature vectors.

5.3.3.3 Normalization

Information retrieval tasks can often benefit from *normalization*, which serves to decrease the importance of terms that occur very commonly and hence convey little information. Before computing distances over feature vectors, we normalized the value of each vector element using *tf-idf* [65]; this is a common practice in text clustering and classification. Given a count of a feature (token) in a test case or its execution (the “document”), the tf-idf is the product of the term-frequency (tf) and the inverse-document-frequency (idf) for the token. Term-frequency is the ratio of the count of the token in the document to the total number of tokens in the document. (For coverage we use number of times the entity is executed.) Inverse-document-frequency is the logarithm of the ratio of the total number of documents and the total number of documents containing the token: this results in a uniformly zero value for tokens appearing in all documents, which are therefore not included in the vector. We normalized Levenshtein distances by the length of the larger of the two strings, which helped handle varying sizes for test cases or outputs.

CHAPTER 6

FUZZER TAMING EXPERIMENTS

To evaluate our work, we needed a large collection of reduced versions of randomly generated test cases that trigger compiler bugs. Moreover, we required access to ground truth: the actual bug triggered by each test case. This section describes our experimental setup and the results of applying our approach to taming fuzzers. Most of the text of this Chapter is from our PLDI 2013 paper [17], which is collaborated with Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide and John Regehr.

6.1 Test Cases

We chose to test GCC 4.3.0 running on Linux on x86-64. Our selection of this particular version was based on several considerations. First, the version that we fuzzed had to be buggy enough that we could generate useful statistics. Second, it was important that most of the bugs revealed by our fuzzer had been fixed by developers. This would not be the case for very recent compiler versions. Also, it turned out not to be the case for GCC 4.0.0, which we initially started using and had to abandon since maintenance of its release branch—the 4.0.x series—terminated in 2007 with too many unfixed bugs.

We used the same collection of Csmith-generated test cases as in our experiments for C-Reduce: 2,501 crash test cases and 1,298 wrong-code test cases for GCC 4.3.0. After reduction, some previously different tests became textually equivalent; this happens because C-Reduce tries quite hard to reduce identifiers, constants, data types, and other constructs to canonical values. For crash bugs, reduction produced 1,797 duplicates, leaving only 704 different test cases. Reduction was less effective at canonicalizing wrong-code test cases, with only 23 duplicate tests removed, leaving 1,275 tests to examine. In both cases, the typical test case was reduced in size by two to three orders of magnitude, to an average size of 128 bytes for crash bugs and 243 bytes for wrong-code bugs.

6.2 Establishing Ground Truth

Perhaps the most onerous part of my work involved determining ground truth: the actual bug triggered by each test case. Ground truth provides the identification of each test case in terms of the bug triggered by the test case and determines whether test cases trigger the same bug or not—test cases with the same ground truth are considered to trigger the same bug in the compiler being tested. Ground truth is fundamental to evaluating our work because it is the basis for interpreting the results. Thus, ground truth must be accurate: the established ground truth for a bug must carry only the information for this bug. For example, our evaluation would be compromised if we falsely considered some test cases to trigger the same bug even if they did not.

It is infeasible for nondevelopers to examine the execution of a complex software artifact such as GCC to identify the actual bug for each of the thousand bug-inducing test cases. Instead, our goal was to create, for each of the 46 total bugs that our fuzzing efforts revealed, a patched compiler fixing only that bug. At that point, ground-truth determination can be automated: for each failure-inducing test case, run it through every patched version of the compiler and see which one changes its behavior. We only partially accomplished our goal. For a collection of arbitrary bugs in a large application that is being actively developed, it turns out to be very hard to find a patch fixing each bug and only that bug.

For each bug, we started by performing an automated forward search over the revision history to find the patch that fixed the bug. In some cases, this patch met the following conditions:

- it was small;
- it clearly fixed the bug triggered by the test case, as opposed to masking it by suppressing execution of the buggy code; and
- it could be backported to the version of the compiler that we tested.

In other cases, some or all of these conditions failed to hold. For example, some compiler patches were extraordinarily complex, changing tens of thousands of lines of code. Moreover, these patches were written for compiler versions that had evolved considerably since the GCC 4.3.0 versions that were the basis for our experiments.

Although we spent significant effort trying to create a minimal patch fixing each compiler bug triggered by our fuzzing effort, this was not always feasible. Our backup strategy for assessing ground truth was first to approximately classify each test case based on the revision of the compiler that fixed the bug that it triggered, and second to manually inspect each test case in order to determine a final classification for which bug it triggered, based on our understanding of the set of compiler bugs.

6.3 Bug Slippage

When the original and reduced versions of a test case trigger different bugs, we say that *bug slippage* has occurred. Slippage is not hard to avoid for bugs that have an unambiguous symptom (e.g., "assertion violation at line 512"), but it can be difficult to avoid for silent bugs such as those that cause a compiler to emit incorrect code. Although slippage is normally difficult to recognize or quantify, these tasks are easy when ground truth is available, as it is here.

Of our 2,501 unreduced test cases that caused GCC 4.3.0 to crash, almost all triggered the same (single) bug that was triggered by the test case's reduced version. Thirteen of the unreduced test cases triggered two different bugs, and in all of these cases the reduced version triggered one of the two. Finally, we saw a single instance of actual slippage where the original test case triggered one bug in GCC leading to a segmentation fault, and the reduced version triggered a different bug, also leading to a segmentation fault. For the 1,298 test cases triggering wrong-code bugs in GCC, slippage during reduction occurred 15 times.

6.4 Evaluating Effectiveness using Bug Discovery Curves

Figures 6.1–6.4 present the primary results of applying our approaches to taming a compiler fuzzer. All results are shown using *bug discovery curves*. A discovery curve shows how quickly a ranking of items allows a human examining the items one by one to view at least one representative of each different category of items [55, 70]. Thus, a curve that climbs rapidly is better than a curve that climbs more slowly. Here, the items are test cases and categories are the underlying compiler faults. The top of each graph represents the point at which all faults have been presented. As shown by the y-axes of the figures, there are 11 GCC crash bugs and 35 GCC wrong-code bugs in our study.

Each of Figures 6.1–6.4 includes a baseline: the expected bug discovery curve without any fuzzer taming. We computed it by looking at test cases in random order, averaged over

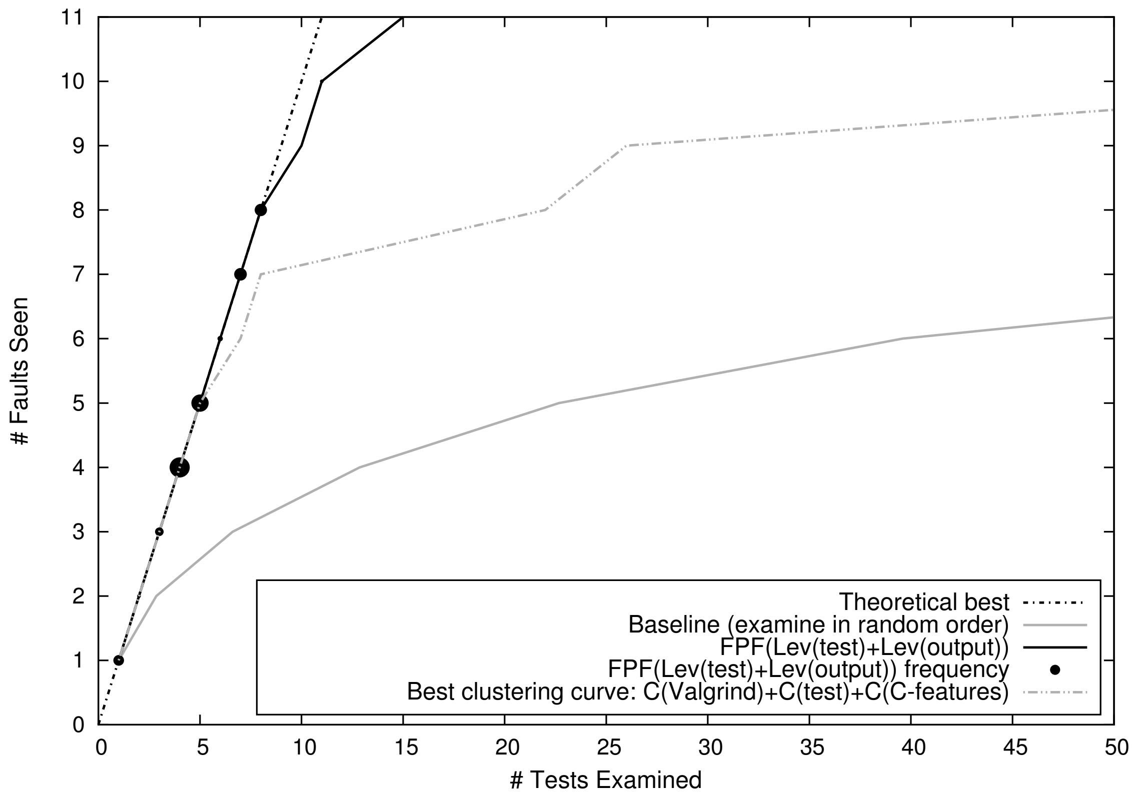

Figure 6.1: GCC 4.3.0 crash bug discovery curves, first 50 tests

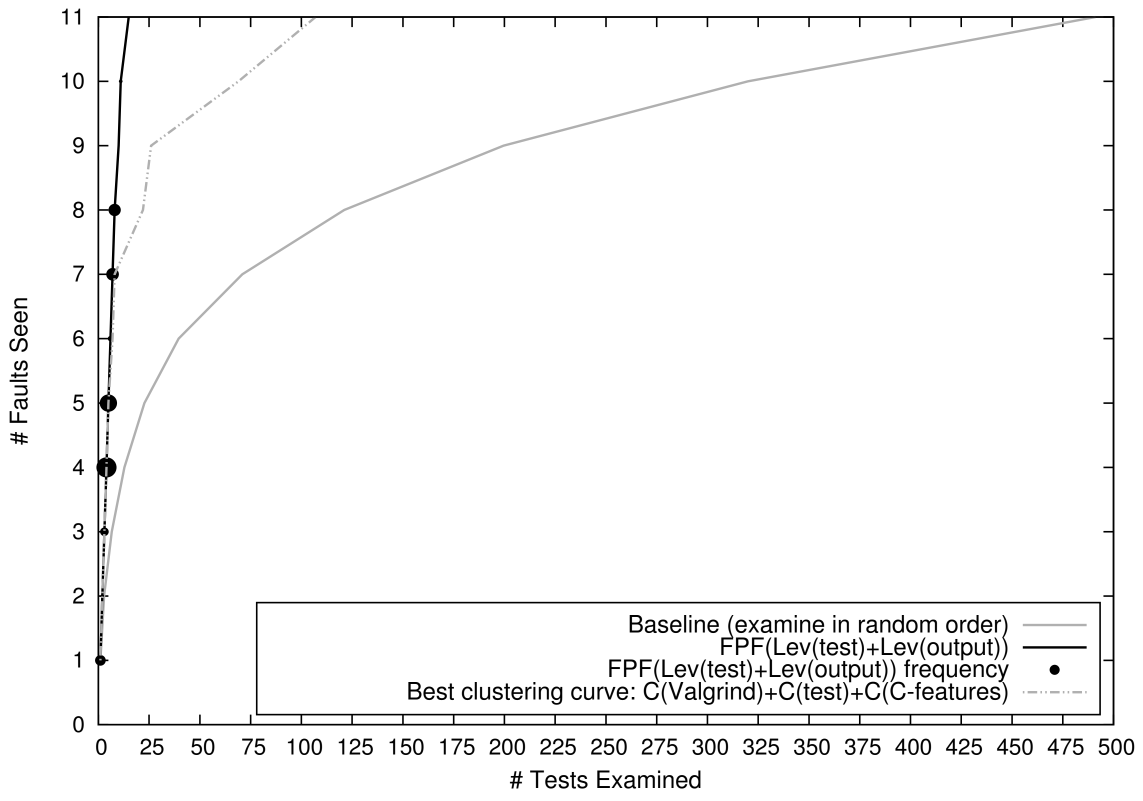

Figure 6.2: GCC 4.3.0 crash bug discovery curves, all tests

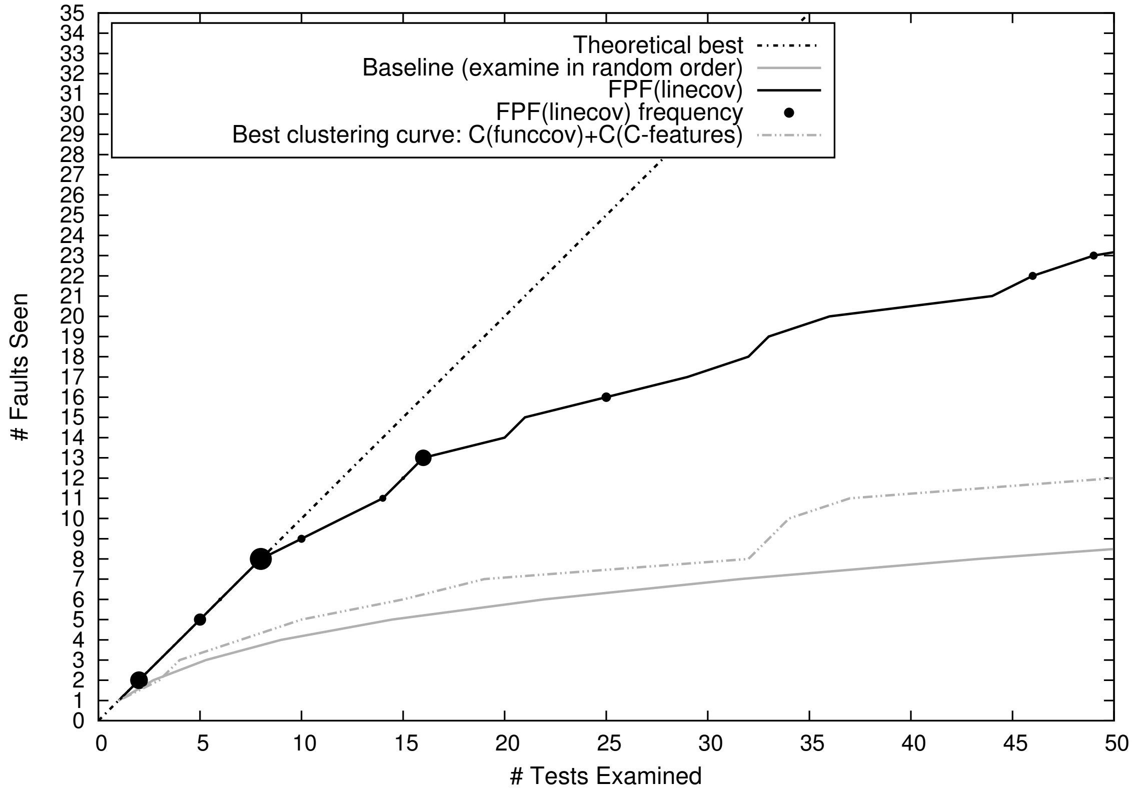

Figure 6.3: GCC 4.3.0 wrong-code bug discovery curves, first 50 tests

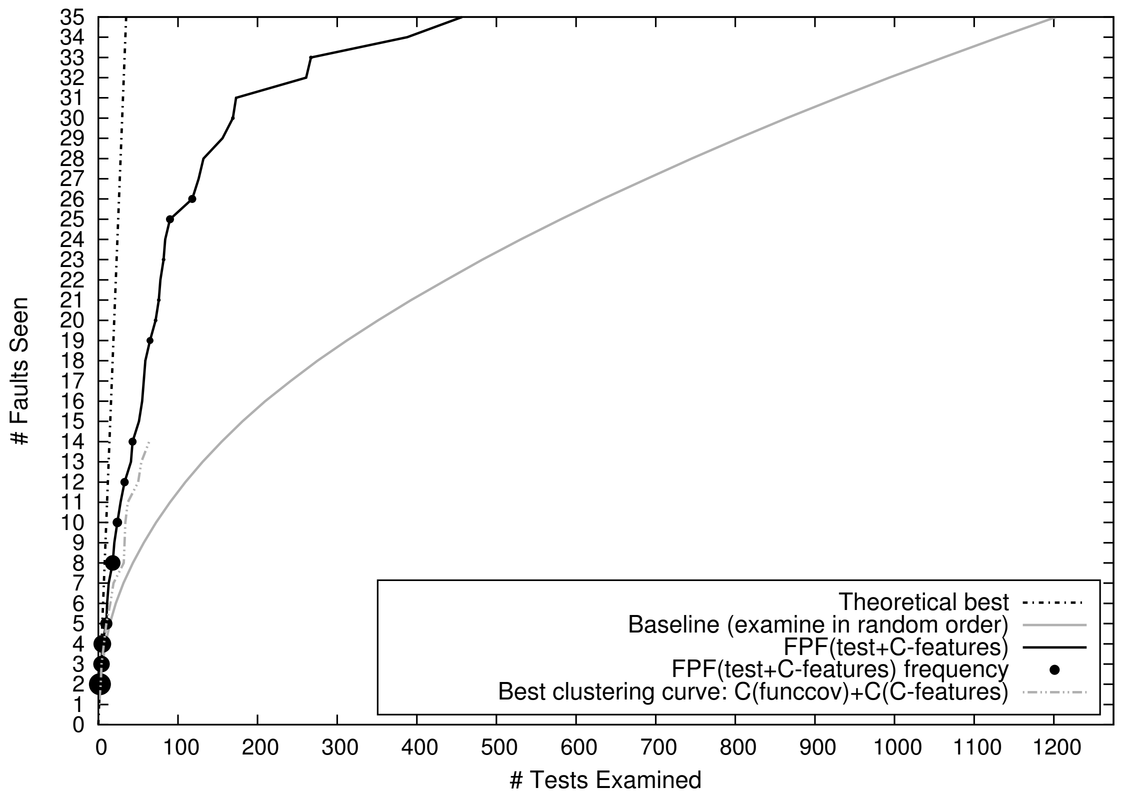

Figure 6.4: GCC 4.3.0 wrong-code bug discovery curves, all tests

10,000 orders. We also showed the theoretical best curve where for N faults each of the first N tests reveals a new fault.

In each graph, we showed in solid black the first method to find all bugs. For GCC crash bugs, this method is also the method with the best area under both curves: the 50 top-ranked and all tests. For GCC wrong-code bugs, it is almost the best for the first 50 tests (and, in fact, discovers the same number of bugs as the curve with the best area); however, for all test cases, the first method to find all bugs has a bad climb curve among all of our methods. For this best curve, we also showed points sized by the log of the frequency of the fault; our methods do not always find the most commonly triggered faults first. Finally, each graph additionally shows the best result that we could obtain by ranking test cases using clustering instead of FPF, using X-means to generate clusterings by various features, sorting all clusterings by isolation and compactness, and using the centermost test for each cluster.

The following items summarize the mapping from the abbreviations used in those figures to the actual feature metrics:

- test: test-case text,
- output: the compiler's crash string,
- funccov: functions covered,
- linecov: lines covered,
- C-features: features extracted using the Clang-based feature detector.

Furthermore, distances between test cases were computed using Levenshtein distance if "Lev" is explicitly stated in the figures; otherwise, Euclidean distance was used. For example, "FPF(Lev(test)+Lev(output))" is involved with the following steps:

1. compute Levenshtein distance between each test case using test-case text;
2. compute Levenshtein distance between each test case using the compiler's crash string;
3. for each pair of test cases, produce a new distance by adding their corresponding distances generated by the previous two steps;
4. apply FPF on the distances obtained from step (3).

6.5 Are These Results Any Good?

Our efforts to tame fuzzers would have clearly failed had we been unable to significantly improve on the baseline. On the other hand, there is plenty of room for improvement: our bug discovery curves do not track the "theoretical best" lines in Figure 6.3 for very long. For GCC crash bugs, however, our results are almost perfect.

Perhaps the best way to interpret our results is in terms of the value proposition they create for compiler developers. If a GCC team member randomly examines 15 reduced crash test cases, he or she can expect them to trigger six different bugs. In contrast, if the developer examines the first 15 of our ranked tests, he or she will see all 11 distinct bugs: a noticeable improvement. Similarly, the developer can only see five distinct wrong-code bugs if he or she looks through the reduced wrong-code test cases in a random order, whereas the developer is able to see 12 distinct bugs by examining the first 15 of our ranked tests.

6.6 Selecting a Distance Function

In Section 5.3 we described a number of ways to compute distances between test cases. Since we did not know which of these would work, we tried all of them individually and together, with Figures 6.1–6.4 showing our best results. Since we did not consider enough case studies to be able to reach a strong conclusion such as "fuzzer taming should always use Levenshtein distance on test-case text and compiler output," this section analyzes the detailed results from our different distance functions, in hopes of reaching some tentative conclusions about which functions are and are not useful.

6.6.1 Crash Bugs

GCC crash bugs were our easiest target: there are only 11 crash outputs and 11 faults. Even so, the problem is not trivial, as the faults and outputs do not correspond perfectly—two faults have two different outputs, and there are two outputs that are each produced by two different faults.

For crash bugs, the best distance function to use as the basis for FPF, based on our case studies, is the normalized Levenshtein distance between test cases plus normalized Levenshtein distance between failure outputs. Our tentative recommendation for bugs that (1) reduce very well and (2) have compiler-failure outputs is: use normalized Levenshtein distance over test-case text plus compiler-output text and do not bother with Valgrind output or coverage information. Given that using Levenshtein distance on the test-case text plus compiler output worked so well for both of these bug sets, where all faults had meaningful failure symptoms, we might expect using output or test-case text alone to also perform acceptably. In fact, the results for Levenshtein distance functions based on test-case text alone were uniformly mediocre at best.

Every distance function increased the area under the curve for examining less than 50 tests by a factor of four or better compared to the baseline. Clearly there is a significant amount of redundancy in the information provided by different functions. Using compiler output plus C-features performed nearly as well as the best distance function, suggesting that the essential requirement is compiler output combined with a good representation of the test case, which may not be satisfied by a simple vectorization: vectorizing the test case plus output performed badly for GCC. All but five of the 63 distance functions we used were able to discover all bugs within at most 90 tests: a dramatic improvement over the baseline's 491 tests. Valgrind output alone performed extremely poorly in the long run—the only metric that was worse than the baseline. Valgrind was of little value because most failures did not produce any Valgrind output. The other four poorly performing methods all involved using vectorization of the test case, with no additional information beyond Valgrind output and/or test-case output.

In summary, for crash bugs, failure output alone provides a great deal of information about a majority of the faults, and test-case distance completes the story.

6.6.2 Wrong-code Bugs

Wrong-code bugs in GCC were the trickiest bugs that we faced: their execution does not provide failure output and, in the expected case where the bug is in a "middle end" optimizer, the distance between execution of the fault and actual emission of code (and thus exposure of failure) can be quite long.

For these bugs, the best method to use for fuzzer taming was less clear. Figures 6.5 and 6.6 show the performance of all methods that we tried, including a table of results sorted by area under the curve up to 50 tests and the number of test cases to discover all faults. It is clear that code coverage (line or function) is much more valuable here than

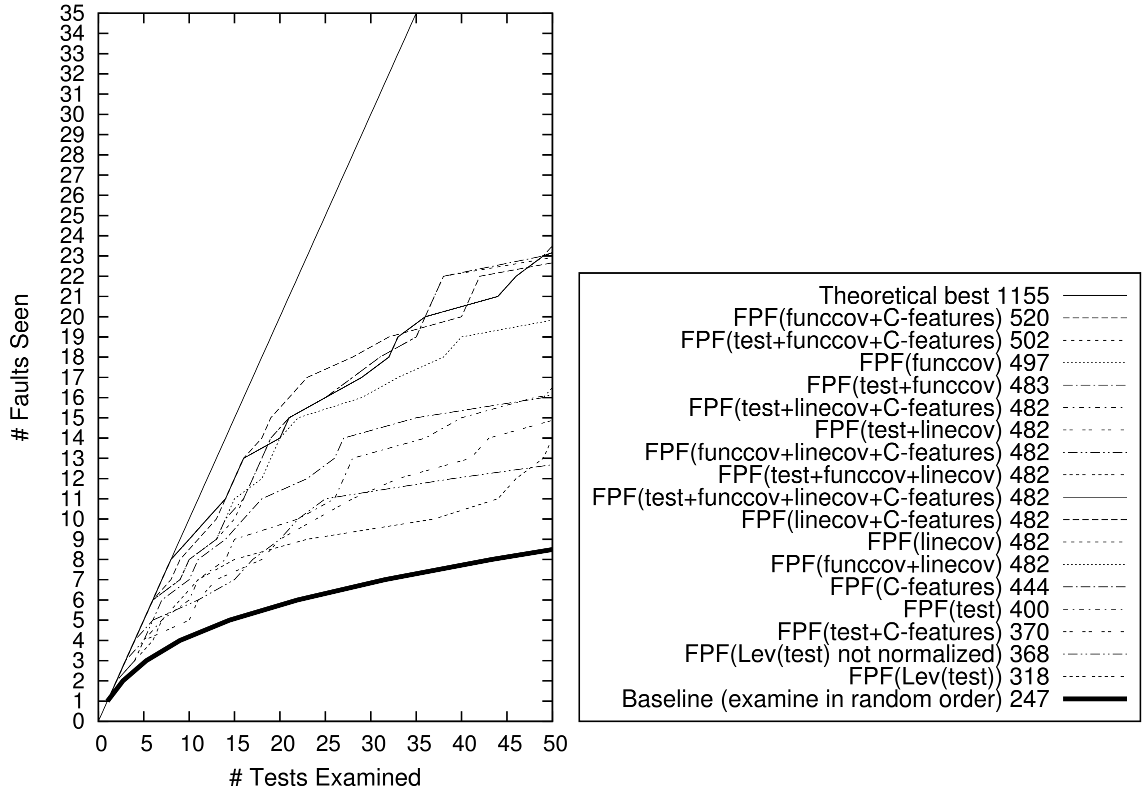

Figure 6.5: All bug discovery curves for GCC 4.3.0 wrong-code bugs, the 50 top-ranked test cases

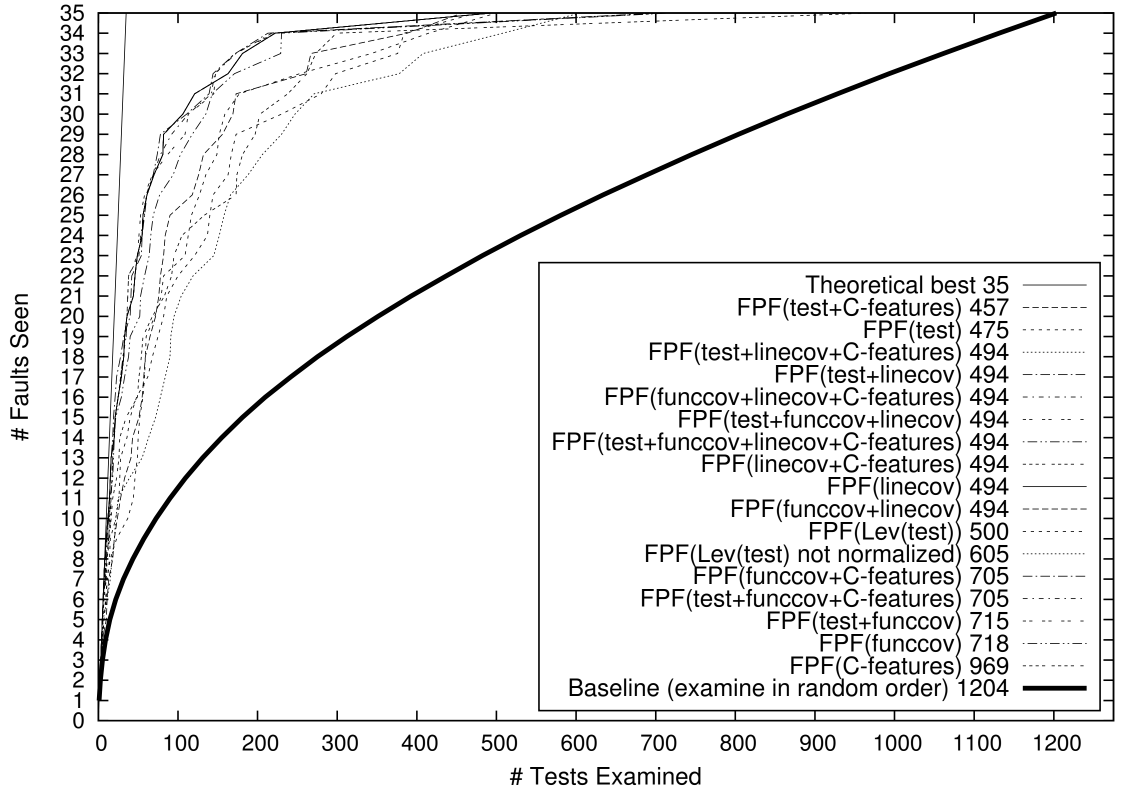

Figure 6.6: All bug discovery curves for GCC 4.3.0 wrong-code bugs, all test cases

with crash bugs, though Levenshtein distance based on test case alone performs well in the long run (but badly initially). Perhaps most importantly, given the difficulty of handling GCC wrong-code bugs, all of our methods perform better than the baseline in terms of ability to find all bugs and provide a clear advantage over the first 50 test cases. We do not wish to overgeneralize from a few case studies, but these results provide hope that for difficult bugs, if good reduction is possible, the exact choice of distance function used in FPF may not be critical.

Nevertheless, with our best method, i.e., FPF(test+C-features), the GCC developer is able to see all 35 distinct bugs by examining 457 test cases. Furthermore, with FPF(linecov), our best method for the short run, he or she can see 12 distinct bugs by examining the first 15 test cases—it probably matters more to the developer.

6.7 Avoiding Known Faults

In Section 5.3.2, we hypothesized that FPF could be used to avoid reports about a set of known bugs; this is accomplished by lowering the rankings of test cases that appear to be caused by those bugs. Figures 6.7 and 6.8 show averaged bug discovery curves for our crash and wrong-code bugs where half of the bugs were assumed to be already known, and five test cases (or fewer, if five were not available) triggering each of those bugs that were used to seed FPF. This experiment models the situation where, in the days or weeks preceding the current fuzzing run, the user has flagged these test cases and does not want to see more test cases triggering the same bugs. The curve is the average of 100 discovery curves, each corresponding to a different randomly chosen set of known bugs.

The topmost bug discovery curve in each figure represents an idealized best case where all test cases corresponding to known bugs are removed from the set of test cases to be ranked. The second curve from the top is our result. The third curve from the top is also the average of 100 discovery curves; each corresponds to the case where the five (or fewer) test cases for each known bug are discarded instead of being used to seed the FPF algorithm, and then the FPF algorithm proceeds normally. This serves as a baseline: our result would have to be considered poor if it could not improve on this. Finally, the bottom curve is the "basic baseline" where the labeled test cases are again discarded, but then the remaining test cases are examined in random order.

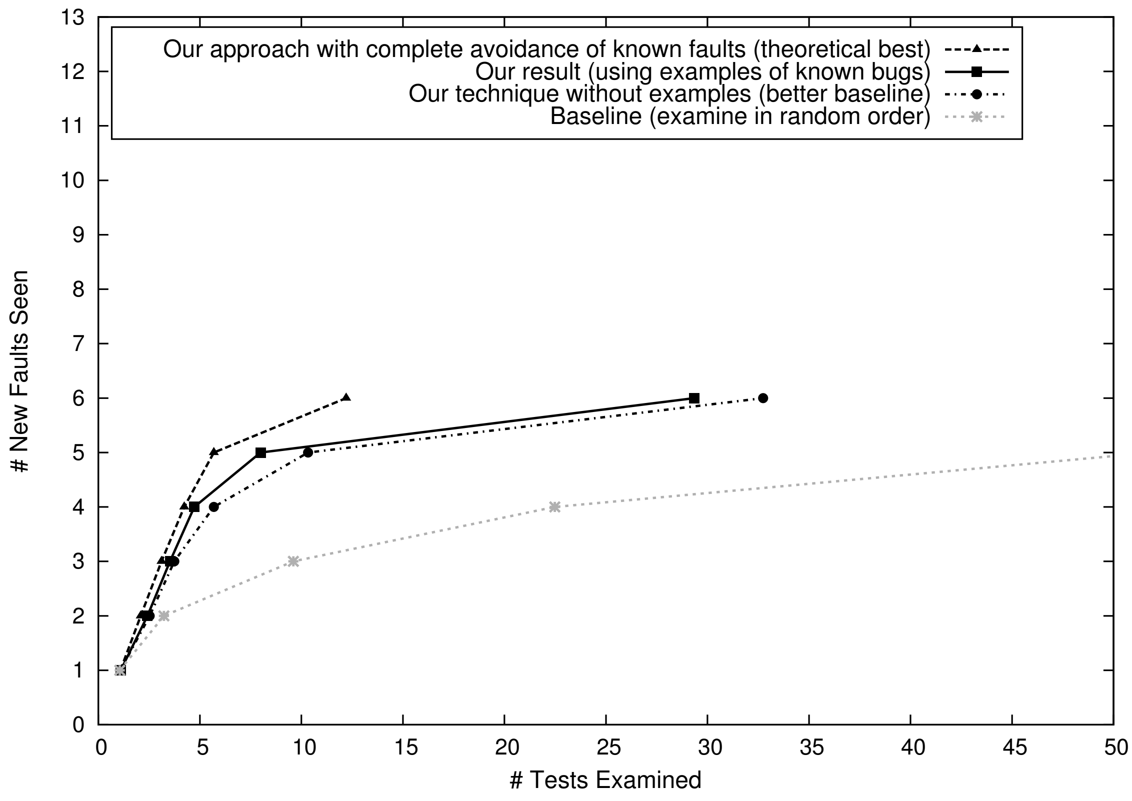

Figure 6.7: Avoiding known crash bugs in GCC 4.3.0

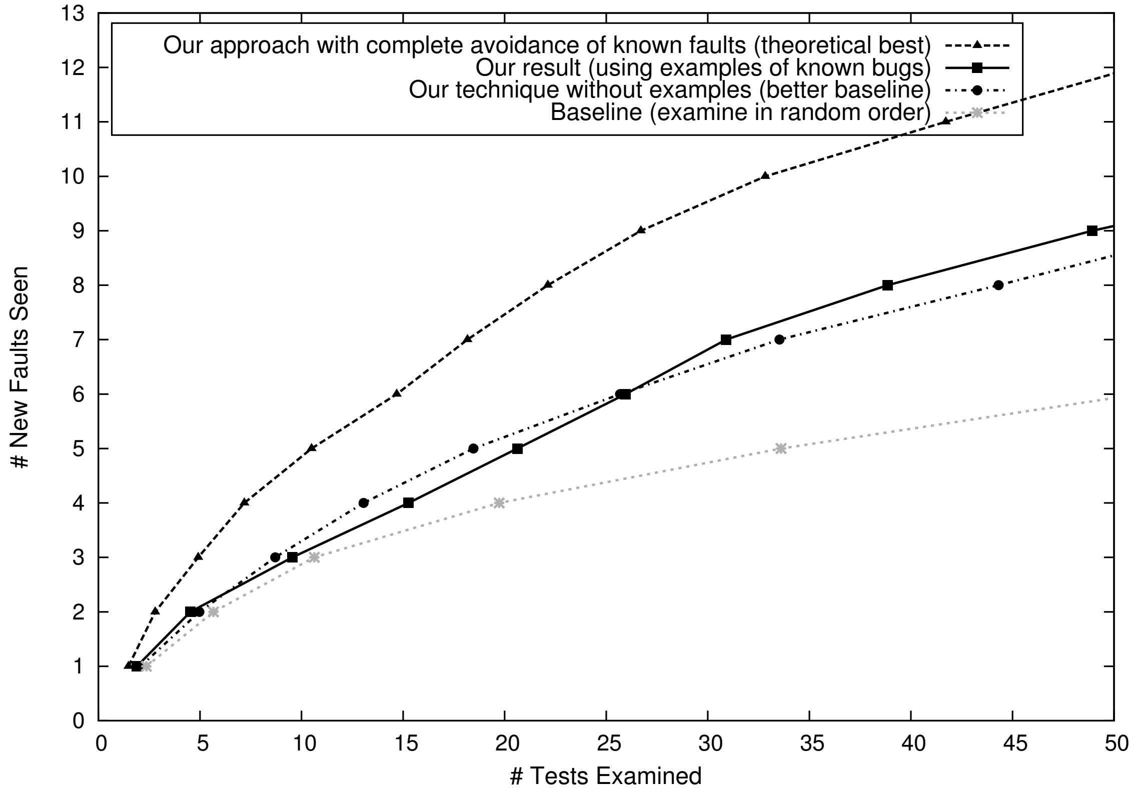

Figure 6.8: Avoiding known wrong-code bugs in GCC 4.3.0

As can be seen, our current performance for crash bugs is reasonably good, but not quite as good for wrong-code bugs. I speculate that classification, rather than clustering or ranking, might be a better machine-learning approach for this problem if better results are required.

6.8 Clustering as an Alternative to Furthest Point First

The problem of ranking test cases is not, essentially, a clustering problem. On the other hand, if the goal were simply to find a single test case triggering each fault, an obvious approach would be to cluster the test cases and then select a single test from each cluster, as in previous approaches to the problem [24,57]. The FPF algorithm we use is itself based on the idea of approximating optimal clusters [26]; we simply ignore the clustering aspect and use only the ranking information.

Our initial approach to taming compiler fuzzers was to start with the feature vectors described in Section 5.3.3.2. Instead of ranking test cases using FPF, we then used X-means [56] to cluster test cases. A set of clusters does not itself provide a user with a set of representative test cases, however, nor a ranking (since not all clusters are considered equally likely to represent true categories). Our approach therefore followed clustering by selecting the member of each cluster closest to its center as that cluster's representative test. We ranked each test by the quality of its cluster, as measured by compactness (whether the distance between tests in the cluster was small) and isolation (whether the distance to tests outside the cluster was large) [70]. This approach appeared to be promising as it improved considerably on the baseline bug discovery curves.

We next investigated the possibility of independently clustering different feature vectors, then merging the representatives from these clusterings [66], and ranking highest those representatives appearing in clusterings based on multiple feature sets. This produced better results than our single-vector method, and it was also more efficient, as it did not require the use of large vectors combining multiple features. This approach is essentially a completely unsupervised variation (with the addition of some recent advances in clustering) of earlier approaches to clustering test cases that trigger the same bug [24]. Our approach is unsupervised because we exploit test-case reduction as a way to select relevant features, rather than relying on the previous approaches' assumption that features useful in predicting failure or success would also distinguish failures from each other.

However, in comparison to FPF for all three of our case studies, clustering was (1) significantly more complex to use, (2) more computationally expensive, and (3) most importantly, less effective. The additional complexity of clustering should be clear from our description of the algorithm, which omits details such as how we compute normalized isolation and compactness, the algorithm for merging multiple views, and (especially) the wide range of parameters that can be supplied to the underlying X-means algorithm.

Table 6.1 compares runtimes, with the time for FPF including the full end-to-end effort of producing a ranking and the clustering column only showing the time for computing clusters using X-means, with settings that are a compromise between speed and effectiveness. (Increased computation time to produce "more accurate" clusters in our experience had diminishing returns after this point, which allowed up to 40 splits and a maximum of 300 clusters.) Computing isolation and compactness of clusters and merging clusters to produce a ranking based on multiple feature vectors adds additional significant overhead to the X-means time shown if multiple clusterings are combined, but we have not measured this time because our implementation is highly unoptimized Python (while X-means is a widely used tool written in C). Because the isolation and compactness computations require many pairwise distance results, an efficient implementation should be approximately equal in time to running FPF. If a curve relies on multiple clusterings, its generation time is (at least) the sum of the clustering times for each component. Note that because X-means expects inputs in vector form, we were unable to apply our direct Levenshtein-distance approach with clustering, but we include some runtimes for FPF Levenshtein to provide a comparison.

That clustering is more expensive and complex than FPF is not surprising; clustering has to perform the additional work of computing clusters, rather than simply ranking items by distance. That FPF produces considerably better discovery curves, as shown in Figures 6.1–6.4, *is* surprising. The comparative ineffectiveness of clustering is twofold: the discovery curves do not climb as quickly as with FPF, and (perhaps even more critically) clustering does not ever find all the faults in many cases. In general, for almost all feature sets, clustering over those same features was worse than applying FPF to those features. The bad performance of clustering was particularly clear for GCC wrong-code bugs: Figure

Table 6.1: Runtimes for FPF versus clustering, where time is in seconds

Program / Feature	**FPF**	**Clustering**
GCC crash bugs / output	0.08	0.71
GCC crash bugs / Valgrind	0.09	0.75
GCC crash bugs / C-Feature	0.10	1.95
GCC crash bugs / test	0.14	15.12
GCC crash bugs / funccov	1.37	162.22
GCC crash bugs / linecov	18.70	2,021.08
GCC crash bugs / Lev. test+output	75.07	N/A
GCC wrong-code bugs / C-Feature	0.49	4.26
GCC wrong-code bugs / test	0.72	67.72
GCC wrong-code bugs / funccov	4.12	1,046.07
GCC wrong-code bugs / linecov	60.60	7,127.42
GCC wrong-code bugs / Lev. test	667.21	N/A

6.9 shows all discovery curves for GCC wrong-code, with clustering results shown in gray. Clustering at its "best" missed 15 or more bugs, and in many cases performed much worse than the baseline, generating a small number of clusters that were not represented by distinct faults. In fact, the few clustering results that manage to discover 20 faults also did so more slowly than the baseline curve. Our hypothesis as to why FPF performs so much better than clustering is that the nature of fuzzing results, with a long tail of outliers, is a mismatch for clustering algorithm assumptions. FPF is not forced to use any assumptions about the size of clusters and so is not "confused" by the many single-instance clusters.

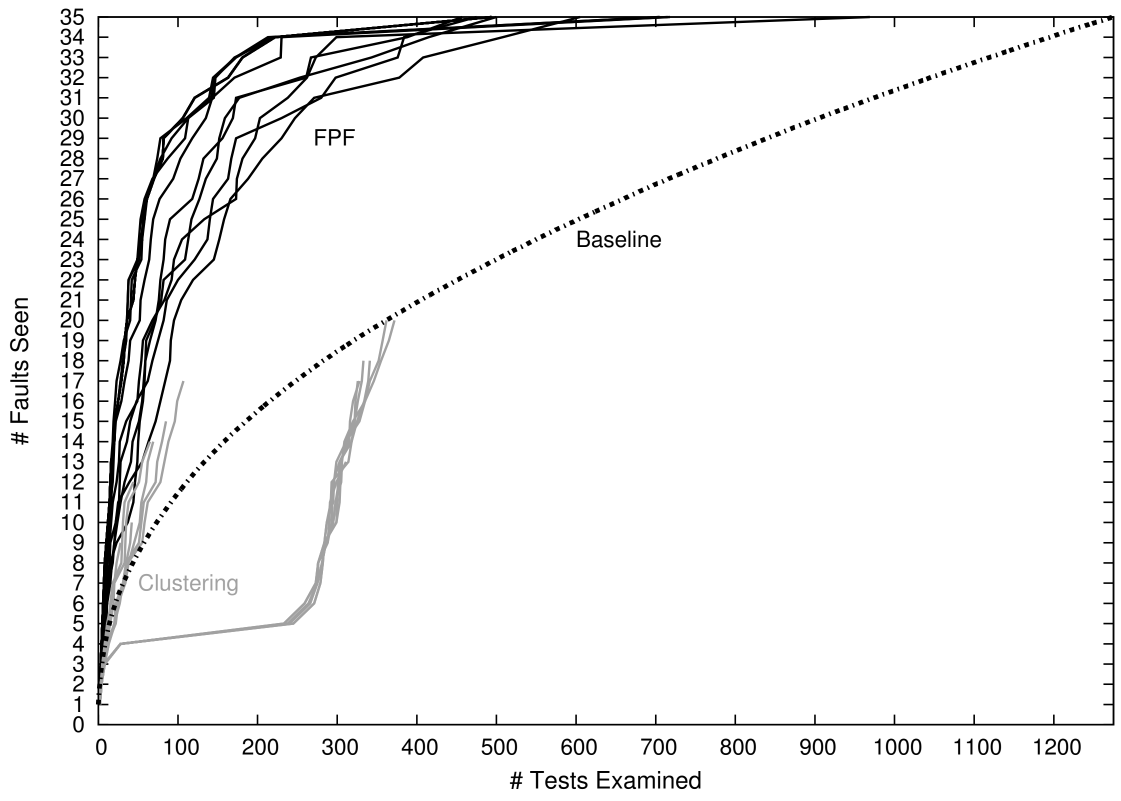

Figure 6.9: GCC 4.3.0 wrong-code bug clustering comparison

CHAPTER 7

RELATED WORK FOR TAMING COMPILER FUZZERS

A great deal of research related to fuzzer taming exists. This section surveys closely related techniques. Some related areas such as fault localization are too large to do more than summarize the high points.

7.1 Software Failure Clustering

Previous work focusing on the core problem of "taming" sets of redundant test cases differs from ours in a few key ways. The differences relate to our choice of primary algorithm, our reliance on unsupervised methods, and our focus on randomly generated test cases.

First, the primary method used was typically clustering, as in the work of Francis et al. [24] and Podgurski et al. [57]. In their work, test cases with the same cause were grouped into the same cluster. In the work of Podgurski et al. [57], features were selected under supervision among profiling data, e.g., function coverage. The selected feature vectors, which were likely to be more relevant to causing the failures, were then used by the underlying clustering algorithms to generate clusters, each cluster representing a group of test cases with the same cause. The user can report to developers one failure-inducing test case from each cluster. Francis et al. [24] later extended these techniques by using tree-based techniques to refine the generated clusters.

Clustering at first appears to reflect the core problem of grouping test cases into equivalence classes by underlying fault. However, in practice the user of a fuzzer does not usually care about the tests in a cluster, but only about finding at least one example from each set with no particular desire that it is a perfectly "representative" example. The core problem that we address is therefore better considered as one of *multiple output identification* [23] or rare category detection [23,70], given that many faults will be found by a single test case out of thousands. This insight guides our decision to provide the first evaluation in terms of discovery curves—the most direct measure of fuzzer taming capability we know of—for this problem. Our results suggest that this difference in focus is also algorithmically useful, as clustering was less effective than our (novel, to our knowledge) choice of FPF.

One caveat is that, as in the work of Jones et al. on debugging in parallel [33], clusters may not be directly useful to users, but might assist fault localization algorithms. Jones et al. provided an evaluation in terms of a model of debugging effort, which combines clustering effectiveness with fault-localization effectiveness. This provides an interesting contrast to our discovery curves: it relies on more assumptions about users' workflow and the debugging process and provides less direct information about the effectiveness of taming itself. In our experience, sufficiently reduced test cases make localization easy enough for many compiler bugs that discovery is the more important problem. Unfortunately, it is hard to compare results: cost-model results are only reported for SPACE, a program with only around 6,200 LOC, and their tests included not only random tests from a simple generator but 3,585 user-generated tests. In the event that clusters are needed, FPF results for any k can be transformed into k clusters with certain optimality bounds for the chosen distance function [26].

Furthermore, our approach is completely unsupervised. There is no expectation that users will examine clusters, add rules, or intervene in the process. We therefore use test-case reduction for feature selection, rather than basing it on classifying test cases as successful or failing [24,57]. Because the number of distinct bugs found by fuzzers follows a power law, many faults will be represented by far too few tests for a good classifier to include their key features; this is a classic and extreme case of class imbalance in machine learning. Reduction remains highly effective for feature selection, in that the features selected are correct for the reduced test cases, essentially by the definition of test-case reduction.

Finally, our expected use case and experimental results are based on a large set of failures produced by large-scale random testing for complex programming languages implemented in large, complex, modern compilers. Most previous results in failure clustering used human-reported failures or human-created regression tests, e.g., GCC regression tests [24, 57], which are essentially different in character from the failures produced by large-scale fuzzing, and/or concerned with much smaller programs with much simpler input domains [33, 48], i.e., examples from the Siemens suite. Liblit et al. [47] in contrast directly addressed scalability by using 32,000 random inputs (though not from an existing industrial-strength fuzzer for a complex language) and larger programs (up to 56 KLOC), and noted that they saw highly varying rates of failure for different bugs. Their work addresses a somewhat different problem than ours—that of isolating bugs via sampled predicate values, rather than straightforward ranking of test cases for user examination—and did not include any systems as large as GCC.

7.2 Fault Localization

Fault localization refers to the techniques to identify the locations where bugs occur in a software system. Fault localization has a long research history and therefore, I only survey some closely related ones.

In early work [73], Whalley presents *vpoiso*, an automated tool to isolate faults in a compiler. vpoiso performs a binary search on a sequence of optimization transformations to localize the first erroneous transformation. When another reference compiler is given, vpoiso is also able to isolate faults in nonoptimization transformations. Renieris and Reiss [60] proposed their techniques, *Set Union*, *Set Intersection*, and *Nearest Neighbor*, all of which operate on coverage data. By measuring the similarity of the coverage data between passed and failed test cases, Nearest Neighbor shows the best effectiveness in terms of the ability to localize bugs, compared to two-set-based models.

Zeller and Hildebrandt [76] presented an approach to isolating a failure cause by examining the difference between the program state of a successful run and the program state of a failing run. Using the Delta Debugging algorithm, this approach is able to automatically narrow down the program states to a small set that is relevant to a failure. Later on, Cleve and Zeller [19] extended the earlier work to another technique, called *Cause Transitions*, which also relies on Delta Debugging to locate the program points that are likely the causes of failures.

Liblit et al. [46, 47] described techniques to isolate bugs in deployed software systems via program sampling. In their framework, the program under test is instrumented with predicates that trace the program execution in a random fashion. The sampled data are then collected and analyzed for locating failures in the program. SOBER, proposed by Liu et al. [49], is another predicate-based statistical bug-localization method. In contrast to Liblit's work, SOBER computes predicates in both failing and successful executions and considers that predicates with more abnormal evaluations in failing executions are more likely to be relevant to the fault.

Jones et al. [34, 35] described the tarantula technique, which computes the *suspiciousness* of each statement in the program under test in terms of the likelihood of causing the fault. Tarantula favors statements being primarily covered in failing runs. These statements get higher suspiciousness values and hence are considered to be more relevant to the cause of the fault.

Our work on taming compiler fuzzers shares a common ultimate goal with fault localization: reducing the cost of manual debugging. Taming compiler fuzzers, on the other hand, targets a narrow problem: preventing compiler fuzzer users from being overwhelmed by a large amount of bug-inducing test cases generated by the fuzzer. Localization may support fuzzer taming and fuzzer taming may support localization. A central question is whether the payoff from keeping summaries of successful executions—a requirement for many fault localizations—provides sufficient improvement to pay for its overhead in reduced fuzzing throughput.

CHAPTER 8

CONCLUSION

Random testing, or fuzzing, has emerged as an important way to test compilers and language runtimes. Despite their advantages, however, fuzzing creates a unique set of challenges when compared to other testing methods. First, bug-triggering test cases produced by fuzzers are often large and make it hard to debug the faults triggered by these random tests. Second, fuzzers indiscriminately and repeatedly find test cases triggering bugs that have already been found and that may not be economical to fix in the short term. Third, fuzzers tend to trigger some bugs far more often than others, creating needle-in-the-haystack problems for engineers who are triaging failure-inducing outputs generated by fuzzers. This dissertation addresses these challenges and improves the utility of a fuzzer by (1) advancing the techniques to automatically generating reportable test cases and (2) taming the fuzzer by suppressing duplicate test cases.

Previous test-case reducers based on delta debugging failed to produce test cases sufficiently small to be directly inserted into compiler bug reports. C-Reduce, our new test case reducer, is capable of producing test cases for C/C++ compilers nearly as good as those produced by skilled developers. The main challenge for C-Reduce is that highly structured inputs make reduction difficult. C-Reduce navigates complex input spaces by exploiting rich and domain-specific transformations, which are beyond the capabilities of basic delta-debugging searches. For 4,799 bug-inducing test cases generated by Csmith, C-Reduce produces outputs that are, on average, more than 735 times smaller than the unreduced tests and more than 30 times smaller than those produced by Berkeley delta. In addition to reducing randomly generated C programs, C-Reduce is able to effectively reduce general C/C++ test cases. We also evaluated C-Reduce on 10 C/C++ programs independently reported by others (not generated by Csmith): C-Reduce was able to turn 960 KB of codes into 335 bytes, on average, whereas Berkeley delta's outputs were of 34 KB.

The second contribution of this dissertation is to characterize the fuzzer taming problem and demonstrate that the problem can be effectively solved using techniques from machine learning to rank test cases in such a way that interesting tests are likely to be highly ranked. Bug-triggering test cases are ranked using the furthest point first technique based on diverse sources of information about the tests. If our rankings are good, fuzzer users will get most of the benefit of inspecting every failure-inducing test case discovered by the fuzzer for a fraction of the effort. For example, a user inspecting test cases that cause GCC 4.3.0 to emit incorrect object code will see all 35 bugs 2.6 times faster than one inspecting tests in random order. The improvement for test cases that cause GCC 4.3.0 to crash is even higher: 32$\times$, with all 11 bugs exposed by only 15 test cases.

8.1 Future Work

In Section 6.3, we discussed bug slippage, a symptom where the reduced version of a test case triggers a different bug than the one triggered by the original test case. One indication of bug slippage is that test-case reducers can uncover new bugs. Thus, an interesting direction would be to investigate the idea of turning a test-case reducer into an effective fuzzer. One hypothesis behind this reducer-as-fuzzer idea is that reduction may explore execution paths in the system under test that may not be covered by the test cases generated by the fuzzer. Furthermore, some static analysis techniques or feedback such as coverage information from the system under test may be used to guide the reducer for generating variants that would cover more code space in the system, in hopes of uncovering more bugs.

One of the distance functions that we used to distinguish test cases is edit distance. We could adapt other distance metrics, especially tree distance, which naturally fits the structure of our test inputs, i.e., C programs. Our coverage feature vectors are based only on failing test cases. Inspired by the tarantula technique [34,35], we can investigate whether we could benefit from using successful test cases. Analyzing the coverage data from both failing and successful runs might produce better feature vectors because some insignificant coverage information could be trimmed off by the analysis.

REFERENCES

[1] CERT basic fuzzing framework. http://www.cert.org/vuls/discovery/bff.html.

[2] The CompCert C compiler. http://compcert.inria.fr/compcert-C.html.

[3] gcov—a test coverage program. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[4] A guide to testcase reduction. http://gcc.gnu.org/wiki/A_guide_to_testcase_reduction.

[5] LLVM Bugpoint tool: design and usage. http://llvm.org/docs/Bugpoint.html.

[6] Report a bug in a Keil product. http://www.keil.com/support/bugreport.asp.

[7] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Proceedings of the 23rd International Conference on Automated Software Engineering* (L'Aquila, Italy, Sept. 2008), pp. 19–28.

[8] Cyrille Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer 13*, 3 (June 2011), pp. 223–246.

[9] Abhishek Arya and Cris Neckar. Fuzzing for security, Apr. 2012. http://blog.chromium.org/2012/04/fuzzing-for-security.html.

[10] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Proceedings of the 14th International Symposium on Formal Methods* (Hamilton, ON, Canada, Aug. 2006), pp. 460–475.

[11] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience 25*, 12 (Dec. 1995), pp. 1315–1310.

[12] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, QC, Canada, 2009), pp. 1–5.

[13] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th international conference on Theory and Applications of Satisfiability Testing* (Edinburgh, United Kingdom, July 2010), pp. 44–57.

[14] Martin Burger and Andreas Zeller. Minimizing reproduction of software failures. In *Proceedings of the the 2011 International Symposium on Software Testing and Analysis* (Toronto, ON, Canada, July 2011), pp. 221–231.

[15] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. In *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation* (Edmonton, AB, Canada, Sept. 2009), pp. 123–124.

[16] Jacqueline M. Caron and Peter A. Darnell. Bugfind: A tool for debugging optimizing compilers. *SIGPLAN Notices 25*, 1 (Jan. 1990), pp. 17–22.

[17] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation* (Seattle, WA, USA, 2013), pp. 197–208.

[18] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming* (Montreal, QC, Canada, Sept. 2000), pp. 268–279.

[19] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA, May 2005), pp. 342–351.

[20] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, United Kingdom, Aug./Sept. 2009), pp. 281–286.

[21] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th International Conference on Embedded Software* (Atlanta, GA, USA, Oct. 2008), pp. 255–264.

[22] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages* (Philadelphia, PA, USA, Jan. 2012), pp. 533–544.

[23] Shai Fine and Yishay Mansour. Active sampling for multiple output identification. In *Proceedings of the 19th Annual Conference on Learning Theory* (Pittsburgh, PA, USA, June 2006), pp. 620–634.

[24] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering* (Washington, DC, USA, Nov. 2004), pp. 451–462.

[25] GCC Team. GCC bugs, 2012. `http://gcc.gnu.org/bugs/minimize.html`.

[26] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science 38* (1985), pp. 293–306.

[27] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering* (Minneapolis, MN, USA, May 2007), pp. 621–631.

[28] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA, July 2012), pp. 78–88.

[29] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics) 28*, 1 (1979), pp. 100–108.

[30] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Security '12: Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA, USA, Aug. 2012), pp. 445–458.

[31] Allen Householder. Well there's your problem: Isolating the crash-inducing bits in a fuzzed file. Technical Report CMU/SEI-2012-TN-018, Software Engineering Institute, Carnegie Mellon University, 2012.

[32] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 9899:TC3: Programming Languages—C*, 2007. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[33] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom, July 2007), pp. 16–26.

[34] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering* (Long Beach, CA, USA, Nov. 2005), pp. 273–282.

[35] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, FL, USA, May 2002), pp. 467–477.

[36] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *Proceedings of the 12th International Conference on Static Analysis Symposuim* (London, United Kingdom, Sept. 2005), pp. 203–217.

[37] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th International Conference on Static Analysis Symposuim* (San Diego, CA, USA, June 2003), pp. 295–315.

[38] Chris Lattner. What every C programmer should know about undefined behavior, May 2011. `http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html`.

[39] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization* (Palo Alto, CA, California, Mar. 2004), pp. 75–86.

[40] Chris Lattner and Misha Brukman. How to submit an LLVM bug report, 2012. `http://llvm.org/docs/HowToSubmitABug.html`.

[41] Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2005), pp. 267–276.

[42] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the 22nd International Conference on Automated Software Engineering* (Atlanta, GA, USA, 2007), pp. 417–420.

[43] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd Symposium on Principles of Programming Languages* (Charleston, SC, USA, Jan. 2006), pp. 42–54.

[44] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), pp. 107–115.

[45] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 8 (1966), pp. 707–710.

[46] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, CA, USA, June 2003), pp. 141–154.

[47] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, USA, June 2005), pp. 15–26.

[48] Chao Liu and Jiawei Han. Failure proximity: A fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, OR, USA, Nov. 2006), pp. 46–56.

[49] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: Statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference held joint with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal, Aug. 2005), pp. 286–295.

[50] William M. McKeeman. Differential testing for software. *Digital Technical Journal 10*, 1 (Dec. 1998), pp. 100–107.

[51] Scott McPeak and Daniel S. Wilkerson. Delta, 2003. `http://delta.tigris.org/`.

[52] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 30th International Conference on Software Engineering* (Shanghai, China, May 2006), pp. 142–151.

[53] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation* (Seattle, WA, USA, June 2013), pp. 187–196.

[54] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (San Diego, CA, USA, June 2007), pp. 89–100.

[55] Dan Pelleg and Andrew Moore. Active learning for anomaly and rare-category detection. In *Advances in Neural Information Processing Systems 17* (Dec. 2005), pp. 1073–1080.

[56] Dan Pelleg and Andrew W. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conference on Machine Learning* (Stanford, CA, USA, June/July 2000), pp. 727–734.

[57] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, OR, USA, May 2003), pp. 465–475.

[58] John Regehr. A guide to undefined behavior in C and C++, July 2010. `http://blog.regehr.org/archives/213`.

[59] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation* (Beijing, China, June 2012), pp. 335–346.

[60] Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th International Conference on Automated Software Engineering* (Montreal, QC, Canada, Oct. 2003), pp. 30–39.

[61] Jesse Ruderman. Introducing jsfunfuzz. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`.

[62] Jesse Ruderman. Introducing Lithium, a testcase reduction tool. `http://www.squarefree.com/2007/09/15/introducing-lithium-a-testcase-reduction-tool/`.

[63] Jesse Ruderman. Mozilla bug 349611. `https://bugzilla.mozilla.org/show_bug.cgi?id=349611` (A meta-bug containing all bugs found using jsfunfuzz.).

[64] Jesse Ruderman. How my DOM fuzzer ignores known bugs, 2010. `http://www.squarefree.com/2010/11/21/how-my-dom-fuzzer-ignores-known-bugs/`.

[65] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM 18*, 11 (Nov. 1975), pp. 613–620.

[66] A. Strehl and J. Ghosh. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *The Journal of Machine Learning Research 3* (2003), pp. 583–617.

[67] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 26th International Conference on Automated Software Engineering* (Lawrence, KS, USA, Nov. 2011), pp. 253–262.

[68] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd International Conference on Software Engineering* (Cape Town, South Africa, May 2010), pp. 45–54.

[69] John-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation* (San Jose, CA, USA, June 2011), pp. 295–305.

[70] Pavan Vatturi and Weng-Keen Wong. Category detection using hierarchical mean shift. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France, June/July 2009), pp. 847–856.

[71] Andreas Vida. *Random Test Case Generation and Delta Debugging for Bitvector Logic with Arrays*. Master's thesis, Johannes Kepler University, Linz, Austria, 2008.

[72] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany, May 2008), pp. 461–470.

[73] David B. Whalley. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems 16*, 5 (Sept. 1994), pp. 1648–1659.

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation* (San Jose, CA, USA, June 2011), pp. 283–294.

[75] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *Journal of Systems and Software 85*, 10 (Oct 2012), pp. 2305–2317.

[76] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (Feb. 2002), pp. 183–200.

[77] Sai Zhang. Practical semantic test simplification. In *Proceedings of the 35th International Conference on Software Engineering, NIER track* (San Francisco, CA, USA, May 2013), pp. 1173–1176.

[78] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Symposium on Principles of Programming Languages* (Philadelphia, PA, USA, Jan. 2012), pp. 427–440.

[79] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation* (Seattle, Washington, USA, June 2013), pp. 175–186.

[80] Jianzhou Zhao and Steve Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs* (Kyoto, Japan, Dec. 2012), pp. 27–42.