RANDOM TESTING OF OPEN SOURCE C COMPILERS

by

Xuejun Yang

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

December 2014

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Xuejun Yang**

has been approved by the following supervisory committee members:

John Regehr	, Chair	**06/06/2014** Date Approved
Ganesh Gopalakrishnan	, Member	**06/06/2014** Date Approved
Matthew Flatt	, Member	**04/03/2014** Date Approved
Eric Eide	, Member	**06/05/2014** Date Approved
Chris Lattner	, Member	**05/12/2014** Date Approved

and by **Ross Whitaker**, Chair/Dean of

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

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

ABSTRACT

Compilers are indispensable tools to developers. We expect them to be correct. However, compiler correctness is very hard to be reasoned about. This can be partly explained by the daunting complexity of compilers.

In this dissertation, I will explain how we constructed a random program generator, Csmith, and used it to find hundreds of bugs in strong open source compilers such as the GNU Compiler Collection (GCC) and the LLVM Compiler Infrastructure (LLVM). The success of Csmith depends on its ability of being expressive and unambiguous at the same time. Csmith is composed of a code generator and a GTAV (Generation-Time Analysis and Validation) engine. They work interactively to produce expressive yet unambiguous random programs. The expressiveness of Csmith is attributed to the code generator, while the unambiguity is assured by GTAV. GTAV performs program analyses, such as points-to analysis and effect analysis, efficiently to avoid ambiguities caused by undefined behaviors or unspecified behaviors.

During our 4.25 years of testing, Csmith has found over 450 bugs in the GNU Compiler Collection (GCC) and the LLVM Compiler Infrastructure (LLVM). We analyzed the bugs by putting them into different categories, studying the root causes, finding their locations in compilers' source code, and evaluating their importance. We believe analysis results are useful to future random testers, as well as compiler writers/users.

To my wife, for the countless weekends with a missing husband.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I thank my advisor, John Regehr, for his support throughout the years. John is the one who brought me into the world of open source compilers. During our first meeting in 2006, he encouraged me to hack GCC, an adventure I promptly embarked on without realizing the huge challenges. While being advised by John, I have learned how to express wild ideas in plain English, and build practical solutions out of them. John has showed me that being a researcher is not in conflict with being a hacker. John took extraordinary effort to correct many expression, grammar, and format errors in early revisions of my dissertation. Without him, the dissertation would never be as polished as it appears now.

I also thank other members of my advisory committee (in no particular order): Ganesh Gopalakrishnan, Matthew Flatt, Eric Eide, and Chris Lattner. They provided many valuable suggestions to my proposal and the final writings. My committee showed a great deal of flexibility when I had to schedule meetings with short notice, and never complained about canceled meetings.

I thank Chris for playing a dual role in our project. First, as the cocreator of LLVM, he provided insight into how compilers work and where the vulnerabilities are; second, he encouraged us to submit LLVM bugs and personally fixed many of them.

Our random program generator borrowed many ideas from previous researchers. I thank the following people for generously sharing their source code with us: William M. McKeeman, Bryan Turner, Christian Lindig, and Eric Eide. Eric's work on volatile-related bugs in compilers directly inspired my research. Many thanks to Eric for his administrative role on Csmith project and guidance on how to be a disciplined programmer.

Portions of the dissertation were previously published in a PLDI 2011 paper titled "Finding and understanding bugs in C compilers." I am grateful to the hard work from coauthors, reviewers, and the shepherd that led to the publishing. That research was primarily supported by an award from DARPA's Computer Science Study Group.

It was a pleasant experience to work closely with Yang Chen, another PhD student advised by John. I appreciate his contribution to Csmith and the personal friendship over the years.

I am thankful to many users of Csmith for reporting bugs or providing feedback. Here is by no means a complete list in no particular order: Pascal Cuoq, Arthur O'Dwyer, Chucky Ellison, Derek M Jones, Paulo J. Matos, Nelson H. F. Beebe, Shakthi Kannan, Andy Lester, Bill Chan, Xavier Leroy.

My last thank goes to the GCC team and LLVM team for promptly fixing (occasionally weird) bugs we reported. By providing strong open source compilers to the community, they have indeed made the world a better place.

CHAPTER 1

INTRODUCTION

Our hypothesis starts from a monkey, symbolizing the randomness, and a compiler. Close examinations of the monkeys, the compilers, and their interactions lead us to believe that random differential testing is an effective way to uncover compiler bugs.

1.1 A monkey

The infinite monkey theorem states that if a monkey hits typewriter keys randomly, it is entirely possible to type the complete works of William Shakespeare given infinite time. On the other hand, with a limited time, even if it is on the order of the age of the universe, it is extremely unlikely for a monkey to exactly reproduce works of Shakespeare [72].

Obviously, time is one of the deciding factors of the monkey's work: giving the monkey a limited time, it would most likely produce garbages; giving the monkey infinite time, it is certainly capable of producing wonderful artifacts.

Practically speaking, when time is constrained, the knowledge of the monkey plays an important role in the outcome. For example, a monkey that knows the vocabulary of an English dictionary certainly has a greater chance to produce one of Shakespeare's works than its peers that do not. If we, as mortals, want the monkey to create something meaningful out of randomness during our life time, providing the monkey with intelligence is essential.

1.2 A compiler

Compilers translate programs written in high level languages into low level representations, e.g., byte code, assembly code, and machine code. High level programming languages are preferred by programmers because they in general provide higher abstractions, scalability, and productivity than their low level counterparts. As the software community moves toward high level programming languages, compilers become increasingly important. Ultimately, programs written in high level languages cannot be executed until they are translated by a compiler or interpreter.

Compilers have direct users (programmers) and indirect users (people who merely execute compiled programs). In either case, users expect compilers to be correct. A bug in a compiler could cause miscompilation of programs, which when executed, could have devastating consequences for their users.

The correctness of C compilers is especially critical because they are used to compile operating systems, part of a trusted computing base. For this reason, C compilers are sometimes included in the trusted computing base as well. C is also used in many safety critical programs. A bug (or a malicious error) in a C compiler is enough to compromise a missile control system and cause a misfire.

There are requirements other than correctness placed on compilers. Compiler users often expect compilers to produce efficient code quickly, i.e., code that utilizes CPU cycles and memory space efficiently. These objectives, sometimes contradictory, lead to increasing complexity of compilers. Sometimes counterintuitive choices have to be made.

Taking LLVM as an example, initially named "Low Level Virtual Machine," the project started in 2000 as a dynamic compilation framework that enables optimizations at compile-time, link-time, or run-time. It became a standalone C compiler with the addition of Clang, a front end for C/C++/Objective-C/Objective-C++ programs, in 2007 [38]. LLVM 1.0 was released in 2003 with 1,267 files and 183,582 lines of code. [1] Over the last 10 years, the source code of LLVM has grown nearly 700% in files, and nearly 800% in LOC. A test suite, which was missing in LLVM 1.0, adds approximately a million lines of source code on top of the 800% growth. For every three lines of compiler source code, approximately two lines of test suite code are needed to validate their correctness. This ratio demonstrates the LLVM team's commitment to delivering high quality compilers, while at the same time highlighting the challenge of validating compiler correctness.

1.3 Putting a monkey and a compiler together - random testing of a compiler

Facing the challenge, we could use a "monkey" to help us validate the correctness of a compiler. The "monkey" is a random program generator, and the random programs are used as test cases of the compiler under test.

McKeeman [44] proposed seven levels of quality for a random program generator. I believe the concept can be extended to all random generators with the following six levels of sophistication:

---

[1] All LOC figures, if not otherwise noted, are reported by cloc 1.58.

- Level 1 are generators without any regard for the programming language. A level 1 generator could simply generate strings of random characters.
- Level 2 are generators that know the tokens in the target language, but cannot properly put them together. For example, a level 2 generator could randomly generate reserved keywords and delimiters in C language in arbitrary orders.
- Level 3 are generators that know the grammar of the language and are capable of generating syntactically correct test cases. The generator sophistication is greatly advanced from the level 2.
- Level 4 are generators that know not only the grammar but also the semantics of the target, and avoid generating problematic code identifiable by scanning nearby code. For example, a level 4 C program generator should never generate " x / 0" unless the handling of divide-by-zero is being tested.
- Level 5 generators have all level 4 capabilities, plus the capability to avoid generating problematic code identifiable by examining runtime behaviors, such as invalid memory references.
- Level 6 generators, on top of level 5 capabilities, know potential vulnerabilities in the software under test (SUT), and adapt their random program generation to reveal the vulnerabilities.

A level 1 or 2 random generator is fairly limited. It produces test cases likely to be rejected by SUT. The test cases probably only validate components inside SUT that are responsible for parsing inputs and rejecting invalid ones. In addition, that space happens to be an easy target of fuzz testing [69], and thus could have been heavily fuzz tested already. Nevertheless, level 1 or 2 generators could be used to find bugs. Some of them are probably unexpected.

While random testing C compilers, we found level 3 and 4 generators are less than desirable. They are likely to exhibit undefined behaviors, such as null pointer dereference. C language standard allows a compiler implementation to translate code with undefined behaviors into anything the compiler writers desire. Given an input with undefined behaviors, the correctness of compilers cannot be expected, let alone reasoned. Nevertheless, level 3 or 4 generators are effective in finding bugs when the compiler is in a fast evolving stage.

While level 5 generators have deep knowledge of the domain, i.e., the programming language, level 6 generators have deep knowledge of the SUT. With this knowledge expressed

in some form in the random programs, the generators are generally more powerful in finding bugs, in terms of both quantity and quality, than lower level generators.

1.4 Adding more compilers to the equation - random differential testing

Random testing has been shown to be effective in testing compilers [4]. For any random testing methodology, the foremost question is how do we know when a random program is miscompiled? A few choices are:

- Looking for error indications, such as crashes, hangs, out-of-memory, and incorrect error/warning messages, during compilation. However, this does not work for silent miscompilations.
- Examining the generated low level code to verify its validity, and comparing it to the source code to validate that they are semantically matched. Program equivalence is undecidable in general. Even if we simplify the problem by using key metrics to represent the semantics, the solution is unlikely to scale to thousands, sometimes millions, of random programs.
- Generating a test oracle while generating a random program. The test oracle is used to determine pass/fail during execution time. A few early random program generators used to generate self-checking code, e.g., assertions, in the random programs. However, either the assertions have to be weakened, or the random programs have to be of limited complexity to be able to make definite assertions. Either way, the ability to find compiler bugs is reduced.
- Using differential testing by adding another compiler implementation to the equation.

Differential testing [44] provides a test oracle. It is based on the idea that if a language specification has multiple implementations, we can 1) compile the same program with different compiler implementations, 2) execute the compiled programs, and 3) compare the results from different executions. If the execution results do not agree with each other, then at least one compiler implementation is wrong unless the input program is invalid. In essence, differential testing gives an oracle that tells something is wrong, but not what is wrong, or how it is wrong.

Optimizing compilers are usually designed with passes of transformations, some of which are optional. A single compiler can be practically considered to be multiple implementations if it provides means to compile programs with different passes.

By combining random testing with differential testing, we can fully automate the compiler-testing process end-to-end. And with the automation, mass random testing becomes possible. The end result is that numerous compiler bugs are found in this fashion in recent years [14, 58].

The test oracle of differential testing is valid only when each test case has just one interpretation. In the case of C compilers, it means the test case has to be free of undefined behaviors and independent of unspecified behaviors. Otherwise, the meaning of the random program becomes ambiguous, the execution result becomes nondeterministic, and the compiler implementations are not required to be consistent on the results.

My thesis statement is that random differential testing is effective in finding compiler bugs. We prove the statement with a sophisticated random program generator - Csmith.

1.5 Challenges

We aim for a level 6 random program generator because we are testing the most well-established C compilers. Some of them have been used for decades by generations of programmers. Their strength has been testified and augmented by widespread academic and industrial usage. We believe only a random generator with the highest sophistication level could be the most successful at finding bugs in these compilers. We believe the goal is accomplished with Csmith.

1.5.1 A random program generator that is both expressive and unambiguous

Before level 6 is reached, we found it is challenging to implement a level 5 generator. Creating a random program generator is similar to teaching a child to speak: At first, they can only mumble, but once they can speak recognizable sentences, the chance of ambiguity increases while they grow their expressiveness. Similarly, creating a random program generator which maintains 100% unambiguity is challenging. When we add the expressiveness, more efforts are required to keep the generator from generating ambiguous code.

C programs could be ambiguous because the language standard stopped short of assigning definite meanings to certain code. Instead, the standard declares that such code have undefined, unspecified, or implementation defined behaviors. C is a flexible low level language that is portable between many different platforms. The ambiguities are purposely left in the standard to recognize the diversity of the platforms that C supports. The

ambiguities are disclosed in the standard so relevant parties, such as compiler writers and compiler users, could handle them properly.

The latest C language standard, C11 [29], designates 58 kinds of unspecified behaviors, 204 kinds of undefined behaviors, and 104 kinds of implementation-defined behaviors [29]. We have to implement mechanisms in the random generator to carefully avoid most of them. Some of them can be avoided by not generating a language construct or feature altogether: e.g., there could never be an array out-of-bound access if the generator produces no arrays. However, limiting the expressiveness would limit the code coverage of compilers, thus limiting the bug-finding power of differential random testing.

These two objectives, being expressive and being unambiguous, sometimes are contradictory. Expressiveness comes with a chance to be ambiguous. For example, adding integer arithmetic introduces signed integer overflow, while adding pointers leads to possible null pointer dereference. For every language construct or feature added, we must add logics in the random generator to avoid undesired behaviors associated with them. These logics sometimes require complicated generation-time program analysis.

1.5.2 Find and understand bugs in open source C compilers

We target our random testing effort mainly toward two open source compilers: GCC and LLVM. They are chosen because they are both well-respected, well-established, and widely used by research community and commercial companies. We want to pick worthy opponents for our random generator.

The GNU Compiler Collection (GCC) is a compiler system from the GNU Project [23]. It is composed of compilers for several programming languages that share a common middle end. LLVM is a relatively younger compiler system. Like GCC, it also supports multiple languages, and targets to many popular platforms.

GCC and Clang/LLVM are two prime examples of strong open source compilers. They have been studied extensively by academic researchers [17] [42], and used or tested by thousands of developers around the world over the years. In addition, both compilers are shipped with large test suites that are diligently used for validation by dedicated teams [19] [43].

Both GCC and LLVM/Clang have evolved over the years to become large software systems due to the demands of supporting multiple languages and architectures and of performing more sophisticated optimizations. As of June 2013, LLVM/Clang is shipped with around 2.8 MLOC, while GCC is shipped with approximately 1.5 MLOC. Nowadays, both compilers are stable enough that it is difficult for a tester to find bugs in them. Creating

a random program generator and finding bugs in GCC and LLVM is understandably more challenging than compilers received less engineering investments.

Our effort does not stop after bugs are found. We report them to compiler teams, and we try to understand the bugs from a nondeveloper's point of view. We study the bugs and identify areas in compilers that are potential targets of random testing, and then try to extend the generator to such areas.

Besides GCC and LLVM, we applied random differential testing on other well-known compilers, most of them being commercial software. We found bugs in those compilers quickly. But the experiment stopped there. Commercial compiler vendors are often not eager to fix bugs we report and/or are unwilling to disclose how the bugs were fixed.

1.6 Evaluation

Our goal, creating a random program generator that is both expressive and unambiguous, is achieved with Csmith. It generates a large set of C language features. Csmith uses GTAV (Generation Time Analysis and Validation) to avoid any ambiguity in the generated code throughout the random generation process.

Our generator is expressive because it supports most C features. Still, we need to show its expressiveness quantitatively. Since there are no concrete metrics to represent expressiveness, we resort to an indirect metric: code coverage on compiler source code. We will say that the expressiveness of a random program generator is correlated to the variety of the programs it generates, which in turn is approximately correlated to the source code coverages on compilers that can be achieved by the random programs collectively.

We believe in the domain of compiler testing, and for the purpose of comparing expressiveness between random program generators, compiler code coverage is a fairly close approximation of expressiveness. After all, the random programs are used to exercise compiler code. A generator that produces random programs capable of exercising more code in the compilers is more expressive than a generator that reaches less compiler code.

In the domain of compiler testing, bug-finding performance is another important metric. Bug-finding performance measures the effectiveness of finding distinct valid bugs. Duplicate bugs and invalid bugs (errors triggered by invalid inputs) are excluded. To add to challenges, we even exclude bugs that are already reported by other parties. Bug-finding performance is directly measurable by counting qualified bugs. However, duplication of bugs could be hard to determine. In practice, we judge whether two bugs are distinct based on the error messages at compile time. For miscompilation bugs which yield no compile-time error messages, we conservatively assume they are duplicates until proven otherwise: e.g., a bug

that is fixed in a given revision is certainly not a duplicate of a bug that still exists in that revision.

We compared Csmith with several previously published random C program generators on both metrics. All those generators had been used to find bugs in C compilers. The most recent one randprog is from our own research group, which had found numerous bugs in GCC and LLVM.

Bug-finding performance is measured in both controlled and uncontrolled environments. In a controlled environment, we are interested to know how good Csmith is at finding compiler bugs within a short period of time, relative to other previously published random C program generators. In an uncontrolled environment, we generate random programs using Csmith, and apply random differential testing to find bugs in well-known compilers including GCC and LLVM/Clang. The validity of the bugs we report is confirmed by the acceptance of compiler teams and the fixes they implement. The uncontrolled experiment lasted more than four years. Most of it was in parallel with our development of Csmith.

1.7 Contributions

We identified six levels of random generators. We argue that to find bugs in well-established compilers, a random generator has to be at the highest levels to be effective. We give a reference implementation of a level 6 generator: Csmith. We propose using generation time analysis and validation (GTAV) to avoid generating invalid test cases. Our implementation of the GTAV is efficient and precise, capable of performing pointer analysis and side-effect analysis using a generic framework. GTAV ensures the random programs are always unambiguous while we add expressiveness to Csmith. In addition to being able to avoid generating ambiguous code, as a level 5 generator would, Csmith is designed to explore the potential vulnerabilities in compilers, as expected from level 6 generators. An example of the vulnerabilities is auto-vectorizer in compilers.

With the criteria we defined in Section 1.6, we found Csmith is more expressive and has better bug-finding performance compared to previously published random C program generators. More impressively, during more than four years of testing, Csmith has found over 400 bugs in GCC and LLVM/Clang. We studied all of the bugs and put them into different categories. Some interesting ones are analyzed to find their root causes. We give our own speculation on why Csmith is able to expose these bugs while other test methodologies failed to do so. Finally, we give suggestions to all parties who have a stake in compilers: the writers, the testers, and the users, on how to improve compiler correctness.

1.8 Thesis organization

This thesis is organized into the following chapters:

- Chapter 1: Introduction - this chapter.
- Chapter 2: Background - some background on software testing; compilers, with emphasis on GCC and LLVM; compiler testing; random testing and its usage in compiler testing.
- Chapter 3: Code generator - the core code generator; its purpose; the design choices; the overall process; the grammatical representation of the generating logic; how randomness influences the process and how we control the randomness.
- Chapter 4: Generation time analysis and validation (GTAV) - the analyses and validation accompanying the code generation; its purpose; its framework; interactions between GTAV and the code generator; examples of analyses and validations; the proof that GTAV is guaranteed to terminate.
- Chapter 5: Evaluation - the uncontrolled experiments: finding bugs in real compilers and reporting them; the controlled experiments: comparing our random generator with previously published random C generators on the metrics of compiler-source-code coverage and bug-finding performance; The relationship between random code size and bug-finding performance.
- Chapter 6: Compiler bug studies - in-depth studies of the compiler bugs found by Csmith; their types; their locations; their importance; their root causes; case studies of a few bugs in GCC and LLVM; suggestions to compiler writes/testers/users.
- Chapter 7: Related work and future directions - recent research making use of Csmith; recent research using random generators for testing compilers/interpreters; What is next for our random program generator?
- Chapter 8: Conclusion.

CHAPTER 2

BACKGROUND

> No service is promised, but we are strongly interested in fixing any bugs; please, please report them.
>
> Leonard H. Tower Jr., GNU C compiler beta test release, 1987

In this chapter, we start with discussing the general practice of software testing, then extend to testing of compilers. Particular interests are paid to open source compilers since they are the SUTs (Systems Under Test) of this research. As examples, testing of GCC and LLVM in practice by their respective teams are presented. The final section covers how random testing of compilers was conducted by previous researches.

2.1 Software testing

Software testing is a critical part of the software development process. It aims to assess or enhance the quality of the software under test (SUT). A SUT typically has documented user requirements and/or design specifications. Software testing evaluates the software against the documents. If they are satisfied, the SUT is considered high quality. In practice, software testing is conducted through bug finding: any behavior while executing the SUT that deviates from the requirements/specifications is considered a bug [73].

ISTQB [31] states that "Software defects usually originate from a programming error (mistake), and end with an execution that produces wrong results (failure). Not all defects will necessarily result in failures, e.g., defects in dead code. A single defect may result in a wide range of failure symptoms" [31].

Broadly speaking, test oracles other than requirements/specifications can be used to detect software bugs. Contracts, competing products, previous versions, user expectations, relevant standards, applicable laws, or other criteria could serve as test oracles [33].

Software testing is as old as software development. Over the years, many methodologies have evolved around it. The web page [73] provides a summarization of them. The following are the major ones:

- Functional testing: Functional testing tries to verify a specification or functionalities of the code. Specifications are usually documented in the user requirements and design specifications, and used as reference during testing;

- White-box testing vs Black-box testing vs Gray-box testing: Black-box testing verifies and validates the software without any knowledge of internal implementation. It views the SUT as a "black-box," and the testing is usually specification-based. White-box testing verifies and validates the software with knowledge of the internal implementation, e.g., data structures and algorithms. The testing focuses on internal interactions of the SUT. Gray-box testing applies black-box level tests with knowledge of the internal implementation;

- Unit testing vs Integration testing: Unit testing, a.k.a. component testing, verifies the functionality of an encapsulated component in the SUT. A component (unit) is usually defined at function or class level. The tests are usually written by developers of the component(s). Integration testing is a follow-up to component testing. It tries to verify the interfaces between components, making sure the integrated product works according to a specification. Integration testing aims to find defects in mismatched interfaces and misinteractions between components (units);

- Regression testing: Regression testing focuses on finding software regressions, incidences where software functionality that was previously working correctly stops working due to recent changes;

- Security testing: Security testing focuses on finding security loopholes that enable system intrusion by hackers. It is essential for software that processes confidential or sensitive data;

- Error testing: Error testing gives invalid inputs to the software, and validates that its error handling is graceful and expected.

Our random differential testing of compilers employs functional testing, integration testing, and regression testing. We try to improve its bug-finding performance by adopting as much gray-box testing as possible: we applied our knowledge of compiler workflow in the creation of the random generator, hoping to explore vulnerabilities in the compilers.

2.2 Compilers

A compiler is a software that transforms source code written in high level programming languages into low level representations. Some examples of high level languages include C, C++, and Java. The low level representations can be in the form of assembly language, machine code, or byte-code. The primary functionality of a compiler is to translate programs written in more programmer-friendly languages into programs that are (almost) ready for execution [67].

2.2.1 History of compilers

Software in early days were mostly written in assembly languages. The languages make programming a lot easier than writing machine code because assembly languages produce more reusable and relocatable programs. But assembly languages are not portable. A program written in an assembly language has to be modified or rewritten if the program is to be executed on a different computer hardware architecture [71].

The need to write portable programs, meaning code once and compile/execute everywhere, led to the invention of higher level programming languages, and consequently, compilers. Around the end of the 1950s, machine-independent programming languages were proposed. Subsequently, several experimental compilers were developed. In 1952, Grace Hopper developed the first compiler for the A-0 programming language [71].

With the benefits of being able to reuse software on different architectures significantly outweighing the cost of writing a compiler, compilers became a mainstream development tool. However, early computers came with little memory, a limitation that posed challenges to compiler writers. Most compilers were written in assembly languages for performance reasons. In 1962, Tim Hart and Mike Levin at MIT wrote the first self-hosting compiler, which was written in Lisp and could be used to compile its own source code [71].

C has become a wide-spread general purpose programming language since its creation by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs. One of the first C compilers was Portable C Compiler (pcc) written by Stephen C. Johnson of Bell Labs in the mid-1970s. GNU C Compiler (gcc) was released in 1987 and became an influential and popular C compiler [16] largely due to its openness and adherence to the language standard. Since its first release in 2003, LLVM/Clang has attracted a lot of attentions as an open source compiler. It is widely adopted by many research and commercial projects due to its well-designed architecture, cutting edge optimizations, and flexible license terms [40].

2.2.2 Structure of a compiler

The functionality of a compiler includes (but is not limited to) [67]:

- Validating the syntactical correctness of the programs, and optionally providing feedbacks to users;
- Generating correct and efficient low level code through translations and/or optimizations;
- Providing runtime organization, e.g., calling conventions, based on the language and/or the architecture;
- Outputting the low level code according to specifications of assemblers, linkers, or virtual machines.

A compiler typically consists of three main components: the front end, the middle end, and the back end. Correspondingly, a compilation undergoes the above three phases. The Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University [35] promoted the division of the compilation processes into phases. This approach makes it possible to piece together front ends (tailored to different languages), back ends (targeted to different architectures), and a shared middle end. Together, they form a compiler suite that supports multiple languages. Practical examples of this approach are GCC and LLVM.

The front end checks whether the language syntax of the program is correct. Semantic errors, such as divide-by-zero, could also be checked if possible. Type checking is performed statically. If there are any errors, they are reported, and the compiler stops. Otherwise, the front end generates an intermediate representation (IR) of the source code for processing by the middle end. Major components in the front end typically include preprocessor, lexical analyzer, syntax analyzer, and semantic analyzer [67].

For optimizing compilers, the middle end is where most of the code optimization takes place. The optimization is often implemented as multipass transformations applied to the IRs. Typical optimizing transformations involve removing useless (code has no effect) or dead (unreachable during execution) code, propagating constant values, reusing previously computed results, etc. Program analysis is sometimes required before applying transformations [67].

The back end is responsible for translating the IR produced by the middle end into low level code. Major activities involve instruction selection/scheduling and register allocation. Instruction selection chooses target-specific instructions in the IR. Register allocation

assigns processor registers (hardware registers) with limited numbers to the unlimited pseudo-registers used in the IR. The back end optimizes the target code based on the hardware configurations, e.g., instruction scheduling could recognize that there are multiple execution units and reorder instructions accordingly to maximize parallelism [67]. The output from the back end is typically a program in an assembly language which is then translated into machine code by an assembler.

2.3 GCC

2.3.1 History

The GNU Compiler Collection (GCC) is a suite of compilers developed and published by the GNU Project. GCC supports several programming languages, including C, C++, and Java. GCC has been shipped with many modern Unix-like computer operating systems, most notably Linux, as the default compiler [70]. It has been ported to more than 60 processor families including the most widely used IA-32 (x86) and ARM [24].

GCC 1.0 was released in 1987 by Richard M. Stallman, the founder of Free Software Foundation (FSF), with only a front end for C programs. Later additions of more front ends enabled it to compile programs written in other languages such as C++, Objective-C, Objective-C++, Fortran, Java, Ada, and Go [20]. With more languages supported, the full name was properly changed from "GNU C comppiler" to "GNU Compiler Collecton." GCC is licensed under the GPL, a license that gives programmers freedom to modify the compiler provided they distribute the modified source code. As an open source project, GCC has played an important role in the growth of free software [70].

In 1997, several developers advocated and formed EGCS (Experimental/Enhanced GNU Compiler System) [22]. They intended to combine several GCC forks into a single project. Because the official project was tightly controlled by the FSF, the EGCS group created a separate project for the merged forks. It underwent considerably more active development than the official project. In 1999, FSF officially halted their own development, merged the EGCS branch into the official project, and appointed the EGCS project owners as the GCC steering committee members [70]. As of 2014, GCC is maintained by a varied group of programmers from around the world, under the direction of the steering committee.

2.3.2 System overview

Internally, GCC consists of a collection of compilers for different languages, e.g., cc1 for C language. Externally, the driver program gcc interprets command arguments, decides

which compiler to use for an input file depending on the language used, optionally runs the assembler on its outputs, and then optionally runs the linker to produce an executable [70].

The front end is language-specific. A parser processes the source code written in a supported programming language, and produces an abstract syntax tree (AST or tree for short). The tree representations for different languages could be different. The front end needs to convert the different representations into GENERIC, a language-independent representation. Subsequently, a gimplifier lowers the GENERIC form into the simpler SSA-based GIMPLE form, which is consumed by the middle end [70]. Static single assignment (SSA) is a representation that allows only one assignment per variable. Many static analyses are greatly simplified with SSA.

The middle end of GCC performs language-independent and architecture-independent analyses and transformations on code represented in GIMPLE form. The middle end employs a pass manager to apply various transformations to the IR. The passes include the standard transformations, such as loop optimizations, dead code elimination, and common subexpression elimination [70]. The middle end transforms the code into RTL (Register Transfer Language) form for the back end.

GCC's back end depends on machine description files, which are usually provided by the architecture developers who want to port GCC. The machine description specifies target-specific information, e.g., the endianness, word size, and calling conventions. The machine description files are taken into consideration when building GCC for the target architecture. In the back end, RTL code is mapped to machine instructions and hard registers, both specified in machine description files. During register allocation, initially-assigned pseudo-registers are assigned to hardware registers. Sometimes not all pseudo-registers can be assigned due to the limited numbers of hard registers. A follow-up pass "reloading" accommodates the unassigned pseudo-registers by "spilling" them to the stack [70].

2.3.3 GCC testing

GCC ships with several test suites to help maintain compiler quality. Most language front ends in GCC have test suites, including C, C++, Ada, Java, and Go. Some optimizations, such as link-time optimization and profile-directed optimizations, also have their own test suites [19].

The C language testsuite includes a "gcc.dg" subsuite and a "gcc.c-torture" subsuite. The former is used to test particular features of the C compiler, while the latter consists of code fragments which have historically broken the C compiler. The "torture" test cases are

compiled with multiple optimization options. They also contain tests to ensure that certain optimizations occur for particular inputs.

A test harness is implemented to automatically build GCC and compile its test suite as well as SPEC CPU 2K/2006 with various optimization options. The harness runs on Linux/ia32 and Linux/Intel64. Several volunteers perform regular builds and regression test runs. All regressions are sent to the gcc-regression mailing list. GCC developers have published instructions on how to run the test suites on user machines, and have asked users to submit their own test results to the list.

In addition to test suites, GCC developers use a few selected applications, including Blitz++ and Boost, to verify the functionality of the compiler. Users are encouraged to test-compile applications important to them once the compiler is installed. The applications can be as large as Linux kernel or GCC itself [19].

2.4 LLVM and Clang

2.4.1 History

The LLVM (initially stands for "Low Level Virtual Machine," an acronym that no longer applies) project were started in 2000 by Vikram Adve and Chris Lattner as a research project at the University of Illinois at Urbana Champaign. It was designed to provide an infrastructure that enables optimizations at compile-time, link-time, or even run-time [40].

Apple became involved in the LLVM project in 2005 by recruiting Lattner and investing resources on the project. Since then, LLVM has become a primary development tool on Apple's operating systems [1] while it remains to be an open source project. It is also extensively used by teams outside Apple as the framework to perform code analysis and transformation, in addition to the original purpose of implementing virtual machines. LLVM provides an infrastructure which includes the IR, the debugger, and the runtime library. A great set of tools are built on top of the infrastructure.

LLVM initially used GCC's front end to generate a language-independent IR then convert it to LLVM IR (intermediate representation), which became the representation worked on by various analyses and optimizations. The IR is used as input by LLVM's back end to produce target code. The compiler resulted from integration of GCC's front end and LLVM is named "llvm-gcc."

However, piggybacking on GCC leads to several problems [49]: Apple's software makes heavy use of Objective-C, which was placed low on the priority list by GCC developers. Also, GCC cannot be integrated smoothly into Apple's developing environment (IDE). Finally, GCC is GPL version 3 licensed, which requires developers who distribute extensions for GCC

or its modifications, such as llvm-gcc, to publish their source code. LLVM's license [38] is more liberal and permits inclusion of the source code into proprietary software.

Apple chose to drop the GCC front end and develop its own. This project resulted in Clang, a language front end for C/C++ and Objective-C/C++. Clang makes LLVM completely independent of GCC. Compared to the GCC front end, Clang provides better integration with an IDE [49] and more expressive diagnostics [38]. Clang was open sourced in 2007.

2.4.2 System overview

LLVM serves as the middle end and back end of a complete compiler system. It takes outputs of any front end as input, as long as the outputs are in, or can be converted into, LLVM intermediate representation (IR). LLVM applies optimizations on the intermediate representations before it converts them into machine-dependent target-specific assembly code. Joined with Clang, LLVM/Clang forms a complete compiler system, capable of compiling programs written in C/C++ or Objective-C/C++ to code executable on various target architectures including x86, x64, PowerPC, and ARM.

LLVM IR is a language-independent RISC-like instruction set and type system [41]. Each instruction is in SSA form. Its type system consists of basic types, such as integers or floats, and five derived types: pointers, arrays, vectors, structures, and functions. Unlike GCC, LLVM uses only one IR across all compiling phases. LLVM IR allows code to be compiled and/or optimized statically or just-in-time (JIT). The simplicity and versatility of LLVM IR attract many projects to be conducted using it as the code representation [42].

LLVM has well-designed architecture and modularization. Rather than producing a monolithic executable that is difficult to reuse its components, LLVM produces a set of reusable libraries. It allows an LLVM user to pick and choose which libraries to use in a compilation. It even enable users to write their own libraries and integrate them with LLVM.

2.4.3 LLVM testing

The LLVM testing infrastructure [43] contains two major categories of tests: regression tests and whole programs (test suite). The regression tests are contained inside the LLVM repository and are validated before every commit. The regression tests are incomplete programs consisting of small pieces of code that each test a specific feature of LLVM or trigger a specific bug in LLVM. They are usually written in LLVM IR. Each regression test

case has a "RUN" command that tells the testing utility which commands to execute and which outputs to verify.

The test suite contains whole programs generally written in high level languages such as C or C++. These programs are compiled using several methods and executed, then the outputs are compared. The compilation methods include native compiler (the golden implementation), LLVM C back end, LLVM JIT, and LLVM native code generation. A whole program consists of either single-source or multiple-source files. The latter can be as large as the ones in SPEC 95 and SPEC 2000 benchmark suites [43].

2.5 Compiler correctness and testing

Compiler correctness is the subject of an investigation or engineering effort that attempts to show that a compiler behaves according to its language specification. The effort can be applied at compiler development time (writing compilers using formal methods), or after development time, e.g., subjecting a compiler to extensive testing [68].

Compiler errors, even small ones, could cause great harm to users: specifications are violated, safety is compromised, and productivity is lost. Firstly, software written in high level programming languages depends on the language specification. A miscompilation could violate the contract(s) in the specification without alerting the compiler user. Secondly, a compiler error could alter the behaviors of a program so that a safety requirement is no longer met after compilation even though the source program is safe. Thirdly, when a compiler crashes, the developers have to spend time on finding a workaround rather than producing new code. Even more damage could be done to their productivity when the developers are tempted to hunt for "ghost" bugs in their code while in fact the bug is in a compiler.

Ways to reason about compiler correctness include model checking, formal verification, and provably-correct-compiler generation [12]. The CompCert project [34] gives a formal proof to the compiler transformations. Aided by the Coq proof assistant, they can prove that the observable behaviors of an executable code produced by the compiler match the observable behaviors of the C program. However, the correctness is only formerly verified on transformations during translating a C abstract syntax tree into an assembly abstract syntax tree. Certain optimizations are avoided by CompCert because their correctness is difficult to verify.

The input spaces to many SUTs are infinite, thus testing with all possible inputs is infeasible [7]. The input to compilers are programs, which obviously have an infinite space; therefore, exhaustive functional testing of compilers is impossible. Some compiler defects

are only triggered with rare conditions. In the absence of exhaustive testing, finding these bugs is difficult.

The difficulty of validating compiler correctness brings business opportunity to companies like Perennial and Plum Hall, both of which provide commercial compiler compliance validation suites [51] [52]. Compiler vendors use the test suites to certify or validate their conformance to language standards, or simply to find compiler bugs. However, the effectiveness of the test suites has never been scientifically verified.

2.6 Random testing of compilers

The history of random testing of compilers dates back half a century. With the emergence of compilers in 1950s, Sauder [56] built a test case generator for COBOL in 1962. Hanford [27] at IBM implemented a program generator based on "dynamic syntax" for PL/1 in 1970 that is capable of generating syntactically valid programs. Purdom [53] in 1972 proposed an efficient algorithm for generating sentences based on grammars, and used it to test the correctness of parsers.

Burgess and Saidi [5] designed an automatic generator for testing Fortran compilers. The test cases were designed to contain specific features that optimizing compilers frequently exploit. Test cases are self-checking. To be able to predict the values, the code generator restricts assignment statements to be executed only once. They found four bugs in two Fortran77 compilers.

In 1998, McKeeman [44] coined the term "differential testing." His work resulted in DDT, a program generator that conforms to the C standard at various levels: from the least conforming (random characters) to the most conforming (generated code statically conforms to the standard, and detected dynamic violations by postgeneration analysis), and used them to find numerous bugs in C compilers. The tool generates many programs containing undefined and unspecified behaviors, and some of them can be detected through dynamic checking. Mckeeman's paper contains the first acknowledgment we are aware of that it is important to avoid undefined behavior when generating C programs. It also emphasized the need for test case reduction, and implemented a reduction tool composed of iterative applications of small heuristic transformations.

Lindig [37] used randomly generated C programs to find several compiler bugs related to calling conventions. His tool, Quest, is type-directed and can be extended using short Lua scripts. Quest creates complex data structures, and passes them to a called function where assertions ensure that the values were passed properly. He reported random testing

is comparable in effectiveness with specification-based testing from prior work. This tool uncovered 13 previously unknown bugs in mature open source and commercial C compilers.

Sheridan [58] used a random generator to find bugs in ANSI C compilers. The tool constructs expressions from a list of constants and the principal arithmetic types, and prints out the results of the operations. A source file is generated for each operator. This tool was able to find two bugs in GCC, one bug in SUSE Linux's version of GCC, and five bugs in CodeSourcery's version of GCC for ARM. It is reported that failure to avoid undefined behavior such as signed integer overflows caused some false positives.

In 2008, Eide and Regehr [14] used random differential testing to find errors in C compilers' implementations of the volatile qualifier. All 13 compilers they tested were found to miscompile accesses to volatile objects. Their implementation, randprog, forms the basis of Csmith, but Csmith supports a much richer set of the C language specification and features.

Zhao et al. [78], in 2009, created an automated program generator for testing an embedded C++ compiler. Their tool allows general test requirements, such as which compiler modules to test, to be specified. A program template is then constructed based on test requirements, and used to drive the program generation. They used GCC as the reference to check the correctness of the tested compiler. They reported greatly improved statement coverage in the tested modules, and found several previously unknown compiler bugs.

CHAPTER 3

CODE GENERATOR

> The two go hand in hand like a dance: chance flirts with necessity, randomness with determinism. To be sure, it is from this interchange that novelty and creativity arise in Nature, thereby yielding unique forms and novel structures
>
> Eric Chaisson, Epic of Evolution: Seven Ages of the Cosmos

There are many ways to devise a random program generator. In this chapter, we present our objectives and design choices when we create Csmith. They are followed by in-depth discussions of the random programs and the generation process.

3.1 The objectives

Our goal is to design and implement a random C program generator. The random programs are used as test cases for compilers. To be effective in testing, the generator should operate at the most sophisticated level, level 6, as defined in Section 1.3. It should have sufficient expressiveness. And it should express without ambiguity. Our knowledge of compiler internals should be incorporated into the generator, and increase its bug-finding performance. The terms used in the statement are defined as:

- Random program: A program that is the product of a nondeterministic process which is partly or fully automated.
- Test case for compilers: A program, valid or not, that is fed into compilers.
- Random program generator: A software that fully automates the nondeterministic production of random programs.

- Expressiveness: The variety of the programs produced by a random program generator. It is indirectly measured by the compiler code coverage achieved by those random programs collectively.

- Unambiguity: Each program has a single interpretation. For C programs, this means they are free of undefined behaviors, and free of dependence on unspecified behaviors. Ambiguities caused by implementation-defined behaviors are tolerated because they only matter when differential testing is performed across platforms, an endeavor we carefully avoided.

- Bug-finding performance: The ability to find distinct, valid, and previously unknown bugs.

3.2 Design choices

3.2.1 Maximizing expressiveness while maintaining unambiguity

Random testing of compilers creates an extra level of indirection: We have to create a random program generator, which in turn creates random test cases. And finally, the test cases are consumed by compilers. As shown in Figure 3.1, the random generator has to explore the space of C programs (Space 1), and use them to explore the space of compiler source code and compiler executions (Space 2).

Since both Space 1 and Space 2 are unlimited, we have to choose on which areas in Space 1 we need to focus so we can maximize coverage on Space 2. The question is tricky because there is generally no definite mapping from a C program to the compiler code that it exercises. For example, we can declare two variables in the source code, but we are not sure whether the code will trigger an optimization in the register allocation that coalesced the variables into one register. The code, in its original form or in an IR form, undergoes multiple transformation passes before reaching the pass that performs register allocation. Any of the upstream operations could alter the reachable paths downstream.

An easier question perhaps is: Which areas in Space 1 should we exclude so we can maximize coverage on Space 2? Intuitively, we want to avoid invalid programs because they explore a small portion of Space 2 that deals with syntactically invalid inputs. Invalid programs, while occupying most portions in Space 1, impact little in Space 2.

Even after excluding all syntactically invalid programs, Space 1 is still prohibitively large. To increase bug-finding performance per test-case, we need to further narrow the space and focus on unambiguous programs. A program is ambiguous when it contains anything which

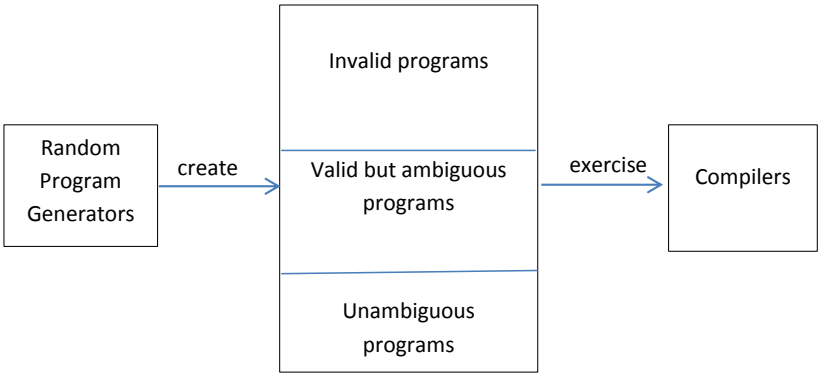

Figure 3.1: Chain of actions from random program generators to compilers

gives the compilers the freedom to interpret (and translate) the code in more than one way. Ambiguous programs are harmful to differential testing because we can no longer rely on disagreements among compilers as test oracles. During our experiments, we found that ambiguous random programs usually lead to false-positive bug reports quickly. Such a bug report would be rejected by compiler developers, and we as bug reporters lose credibility, or even worse, sometimes suffer embarrassment.

3.2.2 Finding a test oracle

A test oracle determines whether a test case has failed or passed. The oracle we use is: if an unambiguous random program is compiled by two or more compiler implementations, and the compiled programs, when executed, yield different effects, then at least one of those implementations is wrong. The rationale behind such an oracle is:

1. A compiler implementation is correct only if it translates a valid program from source language to target language while preserving the meaning of the program;
2. The concrete meaning of a compiled program is represented by the execution effects;
3. An unambiguous program has only one meaning;
4. Therefore, the execution effects from different compiler implementations should be identical if conditions 1 and 2 and 3 are all satisfied.

5. The converse-negative proposition is: if 4 is false, then either 1 or 2 or 3 is false. Since 2 is infallible, therefore 1 must be false when 3 is guaranteed, which means one of the compiler implementations is wrong.

In order to use this oracle, the programming language, for which the compilers are implemented, must meet two conditions: 1) the language is standardized; and 2) there is more than one compiler implementation for that language. Fortunately, C language meets the conditions. It has been standardized since 1989, and there exist multiple widely used C compilers. Some C compilers, such as Clang and GCC, provide more than one implementation from the point of view of differential testing. Both can compile C programs at various optimization levels, and different optimization levels trigger different execution paths in the compilers.

To compare execution effects, i.e., the meaning of the programs, we have to represent the effects as observable and verifiable values. Specifically, we consider two kinds of execution effects: 1) the checksum of global scalar variables; and 2) sequence of read/write accesses to volatile objects. I/O operation is another kind of execution effect which is generally missing in our random programs. However, the checksums in 1) are printed out so their differences become visible to the external world. We also exclude local memory states in comparison because they are not observable at the end of the execution and they could vary across compiler implementations.

In an unlikely scenario, when all compiler implementations err on the same program and produce executables with an identical buggy behavior, differential testing would fail. The "blind spot" is an inherent weakness of differential testing. However, given the variety between compiler implementations, we think the likelihood is low. In addition, the weakness can be remedied by adding more compiler implementations.

Besides the oracle provided differential testing, a more obvious oracle we use is the exit code of the compiler process. Typically, the exit code is zero for a successful compilation. If the compiler crashes, the exit code is nonzero, and an error message is usually provided. Since compilers should always handle failures gracefully (though technically they are allowed to do anything, including crash, when presented with undefined code), therefore crashes should always reveal compiler errors. In fact, both GCC and LLVM customarily print a message after a crash inviting the user to submit a bug report.

3.2.3 Validating intermediate states vs. final states

It seems more bugs could be found by not only validating the program's final state (the state immediately before the program's termination), but also intermediate states during the execution. It is possible that a variable is being written twice, and the first write, but not the second one, is miscompiled by a compiler. The second write would hide the wrong-code bug if we only validate the final states. Applying differential testing on intermediate states would uncover the bug.

In addition to computing and printing a checksum at the end of the execution, Csmith could compute and print a checksum at other execution points. However, checking intermediate states could potentially alter the compiler behaviors, and hide some bugs. Additionally, we would require more complicated comparison logics for execution effects. Evaluation order could lead to different sequences of effects from two correct executions.

We determined the cons outweigh pros while designing Csmith. For a compiler error that happens on a write and then gets covered up by a subsequent write, we think the randomness is likely to work its way to expose the bug eventually.

3.2.4 Grammar-based generation vs. AST-based generation

Many random program generators use production rules (described in grammars) to drive the generation. Such a generator typically starts from a nonterminal symbol, and expands it according to production rules. If more than one production rule applies, one of them is selected randomly. If a chosen production rule generates additional nonterminal symbols, the newly generated nonterminals are added to a work-set to be expanded later. Production iterates until the work-set is empty.

A grammar-based random generator has many advantages: the generated programs are syntactically valid, not difficult to construct, and the generator can be retargeted to a new language by changing the production rules. However, production rules carry limited context sensitivity. Even for context-sensitive grammars, the context is limited and only described by symbols. Without semantic annotations on production rules, it is difficult to perform program analyses during a grammar-based random generation. Adding semantic annotations cancels the benefits offered by grammar-based random generation, that is, easy to construct, and easy to retarget to different languages.

On the other hand, the Abstract syntax tree (AST) is a natural choice for semantic annotations. Compiler front ends often construct ASTs from the source code, then annotate them with semantics. During an AST-based random generation, the generator has a global view of the of all AST nodes already produced, as well as their semantic implications. The

global view enables generation time program analysis. After random generation, printing the abstract syntax tree as a source program is straightforward.

We decided to adopt AST-based random generation because we intend to perform generation time analysis to avoid ambiguity in generated code. Another reason is that the predecessor of Csmith, randprog, was implemented with AST-based random generation.

3.2.5 Avoiding false positives vs. postgeneration trimming

Random programs with ambiguity often cause compilers to produce different result. The compilers are correct even though our test oracle tells us one of them is wrong. Thus, we have a false positive.

When the random generator produces ambiguity, eliminating false positives in our random testing process is challenging, if not impossible. In the past, we have had to either 1) wait for a compiler writer to tell us that the test case in our bug report is undefined, or 2) manually reduce the random program to a point that we can see the ambiguity with naked eyes. Both situations are unpleasant.

Furthermore, once a false positive is identified, we have to slow down our random testing. It is critical to fix the bug which caused ambiguity before reporting new bugs. Otherwise, another false positive is almost surely coming.

Instead of avoiding ambiguity in the generator, we could add a postgeneration trimming pass that filters out ambiguous test cases. There are three arguments against this idea: 1) static analysis cannot detect all kinds of ambiguity; 2) even for the ambiguities that could be detected, the computation cost is very high; 3) when a random generated test case is excluded, the CPU cycles used to generate it are wasted.

With the arguments in mind, we think it is more profitable to avoid generating ambiguities in the first place. It takes more effort to develop the generator, and the generation performance is worse compared to an unguarded generator. However, bug-finding performance per test-case is greatly improved.

3.3 The shape of a randomly generated program

Not including comments, each program generated by Csmith has four sections:

1. Header file inclusions, type declarations, and function prototypes;

2. A set of randomly generated global variables, each initialized to a randomly chosen constant value. The types of the global variables are randomly chosen, with or without *const* or *volatile* qualifier(s);

3. A set of randomly created functions. Each function (except for *func_1*, see Section 3.4 for more details) accepts a number of arguments and returns a value. A function contains a block as the body. A block in turn contains a sequence of statements. Expressions are constituents of statements, while variables and constants (literals) are constituents of expressions. Variables have either a global scope or a scope limited to one of the blocks;

4. A small amount of runtime support, including the program's main function. The tasks of the main function include invoking *func_1* and computing a checksum based on values of the global objects. A global object is included in the computation if it is a scalar variable, a scalar field of a structure/union, or a scalar member of an array.

Csmith adopts AST-based random generation. The generation order is typically from top nodes (the functions) to bottom nodes (the variables), and from left nodes (the first randomly generated function *func_1*) to right nodes (the leaf functions). As shown in Figure 3.2, the height of the trees is five, from level 1 (function) to level 5 (variables and constants). The width of the trees has no limit, even though there are some pragmatically determined limits. For example, at function level, the width is to 10 by default. That is, by default, Csmith generates at most 10 random functions (not counting *func_1* and *main*). Csmith allows users to configure the limits.

Csmith builds two more pieces for each random program: user-defined data types and the *main* function. They are not shown on the above tree.

- User-defined data types: which are prebuilt before the tree construction. Csmith supports integer types ranging from char (either signed or unsigned) to int64 (either signed or unsigned). In addition to built-in types, structures, unions, arrays, and pointers are randomly generated.

  The notable missing ones are floating points and function pointers. Structure types and union types are created with random fields, which could be an integer type, another structure/union type, or a pointer type. Pointer types are derived from all other supported types. Array types are not explicitly created. Array variables are represented by combining a meta-variable, representing the whole array, and an index.

- Main function: which is postbuilt after the tree construction, filled with a call to checksum initialization routine, a call to *func_1*, and statements to compute and print checksum based on global values.

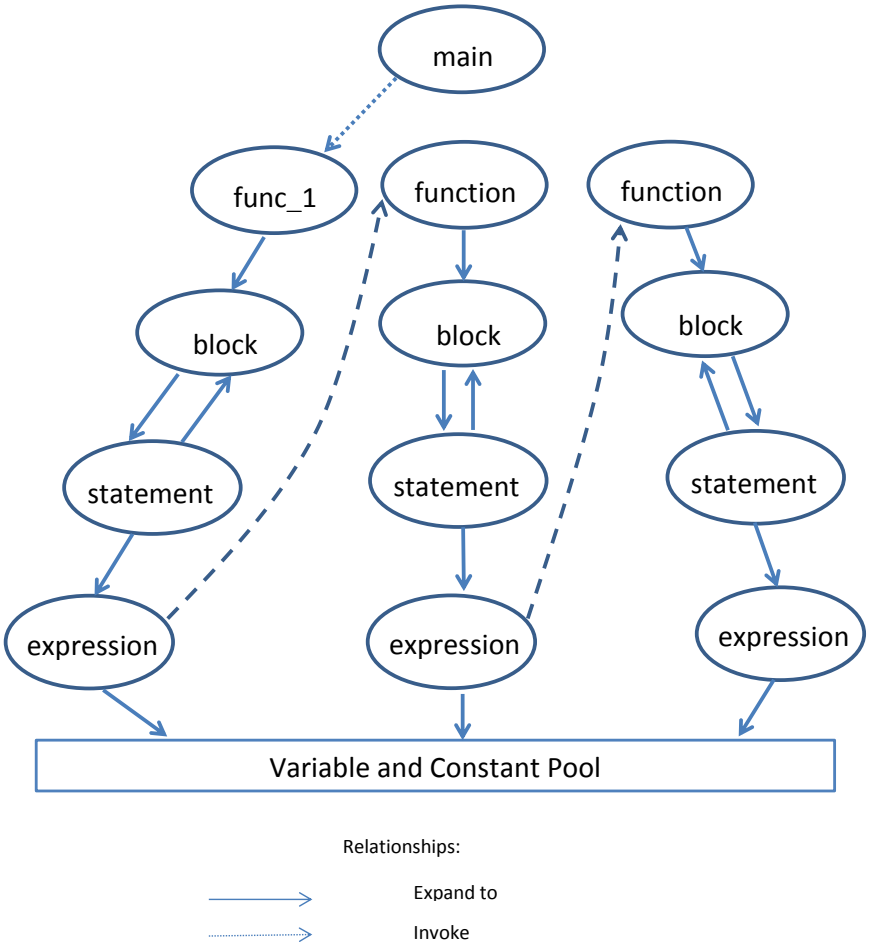

Figure 3.2: Hierarchy and relationships of program constructs in a Csmith abstract syntax tree

The generated functions are nonrecursive. However, due to the possibility of infinite loops, there is no guarantee that random programs generated by Csmith always terminate.

3.4 Overview of the random generation process

The random generation process follows a top-down build-on-demand fashion. The tree construction starts from *func_1*. *func_1* is unique in several ways: 1) it is the starting point of the tree construction; 2) it is invoked only by the main function, and is invoked only once; 3) it does not have parameters.

Functions, including *func_1*, are expanded into a header and a body. The header is expanded into a list of parameters, while the body is represented by a block. A block is the container for a list of statements and a list of variables. A block forms either the body of a function or a constituent of a complex statement, such as true-block and false-block in an if-else-statement.

Csmith builds a block by expanding it into a list of statements. When generating a statement, Csmith has 10 types to select from:

1. Assignment: includes simple assignments and compound assignments;
2. Block: a nested block providing a storage scope for statements and variables;
3. For-statement: a for-loop with a header and a body, which is a block;
4. Invocation: a function call statement;
5. If-else-statement: a branch with a condition, a true-block, and a false-block;
6. Return-statement: a simple return statement;
7. Continue-statement: a control-flow statement jumping to the loop header, available only in the block of a for-statement;
8. Break-statement: a control-flow statement jumping out of the loop body, available only in the block of a for-statement;
9. Goto-statement: an unconditional jump to a location designated with label; and
10. Array-statement: a special for-statement that traverses one or more arrays and computes based on the values of array members.

A statement is expanded into expressions or blocks or their combinations. For example, an if-statement can be expanded to a true-block, a false-block, and a condition, which is an expression. The bi-directional expansion rules between blocks and statements are represented in Figure 3.2.

When generating an expression, Csmith has eight types to select from:

1. Constant: a literal for a representable type, i.e., integers, structures, unions, and pointers (only NULL is allowed);
2. Variable: a memory location which can be accessed via a declared variable, its address, or a dereference of a pointer variable;

3. UnaryExpression: an expression composed of an unary operator and another expression;

4. BinaryExpression: an expression composed of a binary operator and two other expressions;

5. CallExpression: a function call;

6. AssignmentExpression: an assignment embedded in another expression or a statement. An Assignment operator is a binary operator. But assignments have a unique effect on program analyses, thus they deserve a type separated from other binary expressions;

7. CommaExpression: an expression composed of a comma operator and two other expressions. Program analyses for comma expressions are different from other binary expressions, so we created a dedicated type for it;

8. LHS: an expression appears only on the left-hand side of an assignment. It is similar to Variable type, but with more limited production rules.

Csmith assigns weighted probabilities for each type of statement or expression. When creating a statement or expression, one type is chosen randomly based on weights. If the chosen types have nonterminal constituents, the production rules for the constituents will be invoked. For example, when a for-statement is chosen, the following steps are carried out:

- Choose an induction variable
- Choose an initial value for the induction variable
- Choose a final value for the induction variable
- Choose a step for the induction variable
- Create the loop body

While expanding an expression to a function call, Csmith takes the opportunity to use an already fully created function (as long as type checking passes), or to create a new function. When the second choice is taken, Csmith halts the generation of the current function, and switches to the generation of the new function. After its random generation is complete, Csmith continues to work on the caller function.

Once the abstract syntax tree is constructed, Csmith dumps out the tree as a C program while adding supporting pieces to make the program compilable, executable, and testable. These pieces include a *main* function, header file inclusions, macro definitions, function declarations, and user-defined types.

3.5 The generation grammar of Csmith

Csmith uses AST for random generation. The tree nodes, their types and production rules, are implicitly encoded in Csmith's source code. For the sake of simplicity and readability, the implicit production rules are translated into context-free grammars, and presented in Appendix A. Csmith also has context-sensitive generation rules which are beyond the expressive power of context-free grammars. Those rules are discussed in the next section.

There are complete lexical grammars for C89 [32], C99 [30], and C11 [29]. However, the grammars were designed from the point of view of language specifications, not from the point of view of generators. Csmith does not claim to fully cover the lexical grammars, but it represents a large subset of them. Some indications of good coverage can be obtained from the coverage of keywords and operators: out of 33 keywords in C99, Csmith supports 22. Out of 47 operators in C99, Csmith supports 41. C11 was not finalized when we developed Csmith.

We selected the above set of production rules to support in Csmith based on the following beliefs:

- We believe it is more valuable to generate expressive random programs, i.e., programs exercise as much compiler source code as possible. This means key language constructs, including loops, branches, jumps, and function calls, are indispensable;
- There are production rules that merely serve as syntax sugars, which help little in testing compilers, and thus should be ignored by a random generator. For example, although for-loops, while-loops, and do-while-loops appear different syntactically, compilers tend to normalize all of them into a canonical form in an early stage;
- Grammar rules bearing similar semantics should be grouped together. This is for the benefit of generation time program analyses. For example, "++" and "–" operators, typically considered as unary operators by most lexical grammars, should be considered as assignments from the point of view of program analysis;

- Some random choices are unnecessary, such as variable names. Csmith produces standard variable names: a single letter type identifier followed by a sequence number. Although it is possible that compilers make mistakes parsing extremely long variable names, that kind of bug is not targeted by Csmith;

- It pays to dedicate production rules and generation logic to interesting language constructs. An interesting construct is one able to increase compiler code coverage. Csmith generates array-statement, a special kind of loop that traverses arrays and reads from or writes into array members. We believe the resulting loops can better exercise compilers, which tend to aggressively apply loop optimizations, such as vectorization.

3.6 Other considerations of context sensitivity: checks, balances, and biases

Context-free grammars can only describe half of Csmith. The other half is encoded in various context-sensitive analyses and validations during random generation. There are checks, balances, and biases throughout the generation. Checks are hard directions that must be enforced. Balances and bias are soft directions and often configurable. Biases are the preferences we assign to choices. Checks and balances provide determinism to ensure randomness does not get out of control and result in undesired chaos.

In Csmith, checks are mainly used to ensure type safety and avoid ambiguity in the C language. Most ambiguities are caused by undefined behaviors or dependency on unspecified behaviors. The formal definitions are provided in the latest C programming language standard ISO/IEC 9899:2011 (referred to as C11 thereafter) [29] are:

Undefined behaviors: "behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements."

Unspecified behaviors: "use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance."

Implementation-defined behaviors: "unspecified behaviors for which each implementation documents how the choice is made."

Many implementation-defined choices, such as the number of bits in a byte, and the endianness, are invariant on a given architecture. Csmith allows implementation-defined behaviors because we limit differential testing to the same platforms. In other words, we do

not compile and run a random program on x86, then compare the results with the results obtained on PowerPC.

3.6.1 Type check

Compilers perform type checking, and reject programs which fail the checking. To ensure that random programs are compilable, type checking rules in Csmith are generally equivalent or stricter than those found in compilers.

To reduce their number, production rules in Csmith do not specify type requirements. Rather, the instantiation process imposes requirements on types. For example, when an assignment is instantiated, the type of RHS (Right-hand Side) must match that of LHS (Left-hand Side). When a function call is instantiated, the type of an actual parameter should match that of the formal parameter, and the return type must match what is expected by the call site. When an integer operation is being instantiated, each operand must be one of the integer types.

All integer types in Csmith are defined with no ambiguity of their storage size. Csmith supports "int8_t," "int16_t," "int32_t," and "int64_t" which are defined in system header stdint.h. Csmith does not generate language-allowed "int" which has different sizes on different platforms. Even though we do not run differential testing across platforms, we prefer to avoid generating "int" altogether because it leaves the chance of ambiguity.

In Csmith, type matching is defined as type convertibility:

- Type T is convertible to itself (self-evident);
- Any integer type is convertible to another integer type [29, §6.3.1.8];
- For integer types T_1 and T_2, T_1* is convertible to T_2* if T_1 and T_2 have equal size (signed to unsigned and vice versa); and
- All other type conversions are prohibited.

Why does Csmith impose the restriction in pointer type conversions while C11 does not? If a pointer is allowed to be converted to a pointer to a data storage wider than the current points-to data, a read through the second pointer would introduce garbage bits, and thus ambiguity.

Structure types and union types are not convertible to any other types. The alignment of fields inside structures or unions are not specified by C11. Because of the uncertainty over memory layout on different platforms, we stay away from the ambiguity by not allowing

any type conversion for structures and unions. Additionally, Csmith reads from or writes to structure/union fields with no assumption about how the fields are laid out. The layouts are usually unspecified by C standard.

On top of the above type checking rules, Csmith does not allow explicit type casting. Type casting could be used to bypass the type-matching rules encoded in Csmith, and thus should be avoided.

3.6.2 Value-range check

In the following case, Csmith does value-range checking while applying a production rule:

- While generating the second operand of a shift operation, Csmith limits the value between 0 and one less than the bit-size of INT. Shifting an expression by a negative number or by an amount greater than or equal to the width of the promoted expression is undefined in C11 [29, §6.5.7].
- While generating a constant for a pointer type, Csmith limits the value to 0 (NULL)
- While generating an Array Statement, Csmith limits the value range for the induction variable to between 0 and the length of the underlying array(s)

3.6.3 Qualifiers check

Csmith supports two type qualifiers: *const* and *volatile*. Checks on qualifier compatibility are required when a pointer is assigned to another pointer, or when a pointer is used as an actual parameter of a function. Nonpointer variables may have qualifiers but their compatibility checking is waived: a volatile int can be assigned to an plain int, and vice versa. The reasoning is that after assignment, the value exists in a new storage, and the qualifier(s) of new storage do not interfere with the qualifiers of old storage.

For a multilevel pointer, there could be qualifiers at each level of indirection. For example, an integer pointer with three levels of indirection can be qualified as *int * const * volatile * const volatile*. If a pointer is assigned to another pointer with different qualifiers, while both of them are now pointing to the same storage, the program can now access the storage through the second pointer in a way incompatible with the qualifiers of the first pointer, for example, by assigning a *const int** to an *int** then modifying the storage through the second pointer.

C11 [29, §6.5.16.1] states that for pointer assignment, "both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left

has all the qualifiers of the type pointed to by the right." This is not sufficient for some cases. Listing 3.1 is a typical example: the program was trying to modify c even though it is declared as constant. C11, nevertheless, allows conversion at line 3 while marking the pointer dereference at line 4 undefined [29, §6.7.3]: "If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined."

Listing 3.1: Conversion of const-qualified pointers

```
1 char const c = 'x';
2 char *p;
3 char const **pc = &p;    // disallowed by Csmith
4 *pc = &c;                // undefined in C11
5 *p = 'y';
```

C++, on the other hand, imposes sufficient checking on the assignment. The example program is compiled successfully by gcc but is rejected by g++. Qualifier checking in Csmith is more aligned with C++ because the checking in C++ is more bulletproof. The formal rule is defined as:

1. Two pointer types are **similar** if: 1) they have the same level of indirections, and 2) they have the exact same underlying type T;

2. **Qualifier strength** is 2 for *volatile const*; 1 for *const* or *volatile*; and 0 if there is no qualifier;

3. Conversion from pointer type T_1 to T_2 is possible if and only if:

- T_1 and T_2 are **similar**;
- At the last two indirection levels (further away from the storage), **Qualifier strength** of T_2 is greater than or equal to that of T_1; and
- At other indirection levels (if there are more than two indirections), T_1 and T_2 must have equal **Qualifier strengths**.

Some examples of valid conversions and invalid conversions are:

Valid	Invalid
short * → short volatile *	short const * → short *
char volatile ** → char volatile * const *	char ** → char const **
long ** → long * const * volatile	char *** → volatile char ***

3.6.4 Unsafe arithmetic check

Many arithmetic operations, such as divide by zero, are undefined or unspecified in C11. Csmith needs to ensure the generated C programs do not have such operations, or if they

do, the ambiguity is removed. Here is the list of unsafe arithmetic operations we tried to avoid:

- divide by zero
- modulo by zero
- signed integer overflow, which could be caused by addition, multiplication, subtraction, division, modulus, and negation,
- shift by a negative number of bits
- shift by more bits than exist in the operand
- left shift a negative number

Csmith avoids them by statically inserting a runtime check for every potentially unsafe operation. For example, a division operator is replaced with a wrapper *safe_div*. The wrappers are implemented as either macros or functions largely based on code from CERT program [59]. Listing 3.2 shows an example wrapper function.

Listing 3.2: Safe_div wrapper

```
int32 safe_div(int32 si1, int32 si2)
{
    if ((si2 == 0) ||
        ((si1 == INT_MIN) && (si2 == (-1)))) {
        return si1;
    }
    else {
        return (si1 / si2);
    }
}
```

3.6.5 Pointer check

The dangling pointer reference problem arises when a pointer outlives its points-to object, and is used after the points-to object is dead. C11 [29, Annex J2] states that using "The value of a pointer to an object whose lifetime has ended" is undefined. Both dereference and read of the pointer are considered usage of the pointer. According to C11 [29, 6.2.6.1], "Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined." Furthermore, "An

automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it." A dangling pointer is a trap representation; thus, reading it (except for assigning it to another pointer) causes undefined behavior. On the contrary, reading a null pointer is defined.

An object's life ends when it becomes out-of-scope or its memory is reclaimed. Csmith does not generate memory deallocations; therefore, an object lives until the block in which it is declared ends its scope, causing a pointer of the object to be dangling.

The null pointer dereference problem occurs when an attempt is made to read or write through a pointer with a NULL value. According to C11 [29, §6.5.3.1], "If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined." In a footnote, it states that "Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime."

Csmith avoids generating all three kinds of pointer-related undefined behaviors: 1) null pointer dereferences; 2) dangling pointer dereferences; 3) read of dangling pointers. The avoidance is made possible by generation-time points-to analysis which is detailed in the next chapter.

3.6.6 Array out-of-bound access check

C11 [29, Annex J2] states that accessing an array with "an array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression a[1][7] given the declaration int a[4][5]) (6.5.6)" is undefined. Csmith avoids generating out-of-bound array accesses by dealing with two types of array accesses differently:

- Array accesses within a loop: Csmith makes sure the arrays are always indexed with induction variables. An array index is constructed by adding an optional offset to the induction variable. The index always has a value range between 0 and one less than the size of the array. The value of the induction variable is confined to a range with the offset value taken into account. The induction variable is not allowed to be modified inside the loop body; and

- Array accesses outside of loops: Csmith either uses a constant if the constant is proven to be less than the array size, or applies a modulus to the index expression, to ensure the index is within range.

3.6.7 Use before initialization check

C11 states that using a variable before its initialization is undefined. Csmith enforces every variable to be initialized at the declaration site. This is not necessary for global variables. They are zero-ed out by default. Csmith nevertheless generates initial values for them to introduce more variety.

3.6.8 The program size balance

Theoretically there is no upper bound on the size of a random program generated by Csmith. The width of an abstract syntax tree is not limited; therefore, the sizes of the programs have no limit either. But in practice, we found that generating random programs that are too large is counter-productive to compiler testing. Generating them is slow. In most cases, we can reduce a compiler error-inducing random program to a small test case with only a few lines. This indicates size is not an important factor in discovering compiler bugs.

On the other hand, we also found in practice that limiting random program sizes limits the expressiveness, i.e., the ability to excise compiler source code. Csmith aims to preferentially generate random programs with median sizes by giving an upper bound at various levels: the number of random functions, the number of statements in each block, the depth of nested blocks, the complexity of expressions, etc. In many cases, the upper bound is "soft," meaning it can be occasionally exceeded due to other generation constraints. For example, if a function body has four (the limit of number of statements in a block) statements already, Csmith allows a fifth statement, a return-statement, to be added.

3.6.9 The biases

While making a random choice, Csmith does not give equal opportunities to all options. Instead, it usually assigns different weights to different options. The weights are pragmatically determined. For example, while selecting a variable to construct an expression, Csmith favors a variable from the variable pool rather than creating a new one. The bias is imposed based on the belief of the following cause-effect chain: more new variables lead to less variable dependencies, thus lead to more dead variables and/or dead code, thus require less complex program analysis by compilers, thus lead to less compiler bugs found.

Some biases are configurable through the command line or a configuration file. Table 3.1 lists the default weighted probabilities for the 10 production rules of statements. Blocks and calls have zero probability because they are not generated as standalone constructs. Rather, they are the byproducts of other kinds of statements.

Table 3.1: Weights of different kinds of statements

Production rule	**Weight**	**Comment**
Block	0	byproduct of functions, for-statements, etc.
Call-statement	0	byproduct of expressions
Return-statement	5	A return-statement ends the generation of a function. Thus higher probability leads to shorter functions.
Continue-statement	0 or 5	only available inside for-loops
Break-statement	0 or 5	only available inside for-loops
Goto-statement	0 or 5	can be disabled by command line options
If-else-statement	15	keep high to increase program complexity
For-statement	15	keep high to increase program complexity
Array-statement	0 or 10	can be disabled by command line options
Assignment-statement	varies ($\geq$ 40)	takes remaining probabilities (total 100%)

As shown in the table, Assignment is the most favored option, followed by branches and loops. We also assign relatively high weight to array-statements because we are interested in finding bugs in loop optimizations. A certain statement type can be disallowed completely by Csmith with a command line option, effectively reducing its probability to zero, and the extra probabilities would be allocated to assignment-statement.

Table 3.2 lists the default weights for eight production rules of expressions. Call-expression is the most favored type. However, while trying to generate a call-expression, and if no existing function that has a matching return type and no new function is allowed due to the limit on the maximum numbers of functions, Csmith generates a binary or unary expression instead. Of those two, binary expressions are heavily favored over unary expressions. A certain expression type could be disallowed completely by Csmith with a command line option, effectively reducing its weight to zero. LHS has a zero probability because it is not generated as a standalone construct.

To make it flexible to disable/enable certain kind(s) of expressions, the weights are used to calculate the actual probabilities during the random generation. For example, if only call-expression and constant-expression are enabled, their respective probabilities are their weights divided by the sum of weights, 87.5% and 12.5% specifically.

Csmith uses a slightly different weight table when generating expressions as actual function parameters. The weight of call-expressions is lowered to avoid creating call chains

Table 3.2: Weights of different kinds of expressions

Production rule	**Weight**	**Comment**
Call-expression	70	effectively 0 when no new function can be generated and no existing function matches the required type
Constant-expression	10	useful in testing constant folding, constant propagation, etc.
Variable-expression	20	useful in creating define-use chains
Unary/Binary expression	vary	when call-expression is impossible, 95% chance to be binary and 5% to be unary
Comma-expression	0 or 10	can be disabled by command line options
Assignment-expression	0 or 10	embedded assignments that can be disabled by command line options
LHS	0	byproduct of assignments

that are beyond a reasonable length.

Table 3.3 lists the default weighted probabilities for storage scopes of variables. The global scope is slightly favored because the final checksum is computed based on global values. The more global variables that are created and modified, the more variance the checksum value would have.

Do the above tables represent the best probability distributions? Certainly not. We assign the weight based on intuition and empirical results. One important empirical result is the number of compiler bugs found within a period of time. But that number is heavily influenced by timing, compilers under test, and luck. We do not claim the chosen probability distributions are the best, but we think they are sufficiently good for our compiler testing

Table 3.3: Weighted probabilities of variable storage scopes

Scope	**Probability (%)**	**Comment**
global	35	useful in increasing variety of checksums and creating interprocedural use-define chains
local	30	a block randomly chosen from blocks visible at current generation point.
parameter	30	if a formal parameter is available (based on type). Otherwise use local scope.
new	5	necessary for on-demand generation, but the probability should be kept low

purpose.

3.7 Exceptions to the general AST construction order

Csmith generally constructs an abstract syntax tree from top to bottom and from left to right. A tree node is generated first, then validated against a partially generated tree, and committed to the tree finally. A committed node becomes part of the tree. It becomes generally untouchable by later random generations. The rationale is that Csmith works on the tree incrementally. Any part of the tree, once committed, should stay constant. This principle ensures that Csmith keeps making progress. A deviation from the principle may lead to termination problems. The assumption is that everything that passes validation before commitment can also pass the validation later, regardless of the new nodes committed to the tree. The assumption is true for most cases.

An exception must be made while generating forward jumps. Csmith supports goto-statements which jump to almost anywhere in the same function. A forward jump targets a destination that is downstream in the control flow. A seemingly impossible problem for the generator is: how could it jump to some code that has not been generated? Csmith solves the problem by relaxing the restriction on modifying previously generated nodes. It creates a forward jump backward: finding the destination first (usually the latest generated statement), and then inserting a goto-statement somewhere upstream.

Another exception may have to be made while generating loops. Example 4.1 in the next chapter shows that loops have the potential to break the premise that everything validated before commitment stays valid after the commitment. If the premise is broken, Csmith has to backtrack, i.e., delete committed node(s), in order to pass the validation.

CHAPTER 4

GENERATION TIME ANALYSIS AND VALIDATION

> Analysis brings no curative powers in its train; it merely makes us conscious of the existence of an evil, which, oddly enough, is consciousness.
>
> Henry Miller, The cosmological eye

A unique feature of Csmith is that it has a fairly sophisticated generation-time analyzer and the analysis results are actively used throughout the random generation process. GTAV (Generation Time Analysis and Validation) is an integral part of Csmith. The analysis is performed on code that are generated but not yet committed, as well as on code already committed if necessary. The validation is the final guard against any code that is potentially ambiguous.

4.1 The objectives

The objectives are to 1) establish a framework enabling efficient and incremental program analyses, 2) perform various kinds of program analyses within the framework, and 3) feed the results back to the code generator to guide and validate the random program generation. The terms used above are defined as:

- Program analyses: a collection of analyses performed on a program, including, but not limited to, pointer analysis and side-effect analysis;
- Analysis framework: a generalized framework in which the analysis results are represented and carried forward with a unified data structure. Specialized analyses can be built within the framework. To "plug into" the framework, an analysis needs to provide **transfer functions** operating on a set of **program constructs**;

- Program construct: any node representable on an abstract syntax tree, including functions, blocks, various types of statements, various types of expressions, variables, and constants. An single instance of a program construct is called a node;
- Transfer function: a function to establish the relationship between inputs and outputs of a program construct;
- Incremental analysis: an analysis that is performed during or immediately after the generation of a program construct. It avoids reanalyzing previously validated constructs as much as possible;

4.2 The necessity and challenges of generation time analysis and validation (GTAV)

Why do we invest efforts on program analyses while developing a random generator? The act of a random program generation involves making random decisions while generating code, and the decisions are mostly determined by random numbers directly or indirectly. Analyzing code determined by random numbers seems to make little sense.

The reason is that we have to use GTAV to avoid ambiguity at generation time for the purpose of differential testing. Considering the list of undefined behaviors and unspecified behaviors in C11, an avoidance or detection mechanism without program analysis is virtually impossible. Unless we are content with limited expressiveness, GTAV is imperative.

Implementing GTAV is far from trivial. The target being analyzed is a partial program. Many whole program analysis techniques are no longer applicable. Furthermore, incremental analysis means every new program construct requires a new round of GTAV. There could be tens of thousands of constructs in a random program. GTAV has to be efficient and scalable.

4.3 GTAV framework

We developed a GTAV framework which is general enough to allow us to perform various kinds of program analyses in which we are interested. The framework consists of data representations and control representations of the analyses, and the transfer functions at various program construct levels.

4.3.1 Data representations and control representations

All program analyses results are represented as “facts.” For points-to analysis, a fact is a pointer and a list of locations that are possibly pointed to by the pointer. For effect

analysis, a fact is a list of variables that are read or written. Facts are associated with program constructs. A construct has a set of facts as inputs and a set of facts as outputs. Each function has a fact manager, which oversees all input fact sets and output fact sets for constructs in the function.

Csmith does not maintain a control flow graph, but derives control flows by analyzing statement types, and their sequences in blocks. Specifically, there is a control edge between 1) two consecutive statements in the same block unless the first one is a jump; 2) a jump statement (goto/break/continue) and its target; 3) the last statement in a block and the statement following the block in its parent block; 4) the last statement in a callee function and the statement or expression following the call site.

Within a statement with multiple subexpressions, Csmith uses sequence points [1] to derive control flows. It assumes no edges between unsequenced expressions. For two expressions with a clear sequence point in-between, which is defined by the C language standard, Csmith recognizes that there is a control flow edge from the first expression to the second.

Transfer functions produce output fact sets while taking input fact sets as input. The other input to a transfer function is the context, a bag holder for data that might help GTAV, such as the call chain leading to the current function being analyzed.

4.3.2 Pre-commit GTAV

During random program generation, Csmith keeps a set of facts as the running facts accounting for the already generated (and committed) code and their executions. At any generation point, the code generator is given a type, and it generates a type-matched node tentatively. A node is basically an instance of constructs on the AST depicted by Figure 3.2. After the node is generated, and before it is committed to the tree, the node is analyzed using the current running facts as inputs. The transfer function performs both analyses and validation at the same time. Both operations are expensive, so we want to do them in one pass. If the validation fails at one point, the remaining analyses are skipped. This round of GTAV is called **pre-commit GTAV**. The context it takes as the other input is called **generation context**.

If **pre-commit GTAV** fails, the node is discarded, and the current running facts stay unchanged. If it passes, the node is committed to the tree, with current running facts

[1] C11 [29, §5.1.2.3] defines that "sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B."

updated to the outputs from the transfer function. The previous running facts are saved as inputs to the node, and the current running facts are saved as outputs to the node.

4.3.3 Post-commit GTAV

After a node is committed to the tree, Csmith has to revisit it for analyses and validation when needed. The revisit is typically required while generating function calls, loops, and jumps. The revisit is necessary because the inputs to the transfer function, either the facts or the context, have changed; thus, the last round analyses results cannot be used for validation anymore. This round of GTAV is called **post-commit GTAV**.

If post-commit validation fails on a committed node, the node is not discarded; rather, the calls/loops/jumps that triggered the revisit are discarded or modified to be able to pass the validation. If the offending construct is discarded, Csmith moves to generate the next node. If the offending construct is modified, a new round of GTAV is necessary.

4.3.4 Transfer function for programs

We implement a transfer function at each level of the abstract syntax tree (see Figure 3.2). The topmost level is the program, which remains incomplete until the generation finishes. The input fact set to a program is empty $\emptyset$. The output fact set from a program is almost the same as the output fact set of *func_1*. The *main* function is ignored by GTAV because it is just a wrapper for *func_1*. Because facts are only used during the random program generation, not after, in practice Csmith pays no attention to the output fact set of the program.

4.3.5 Transfer function for functions

4.3.5.1 Pre-commit GTAV

All constructs (except for types) in Csmith are generated on demand. A function is generated only when an expression requires a function call, and there are no existing functions with a matching return type.

The input fact set to the new function is the current running facts at the generation point when the demand arises. Some adjustments are needed to pass the facts from call site to callee. We refer to the adjustments as the caller-to-callee handshake. The handshake modifies the input fact set by 1) pruning irrelevant facts, e.g., facts concerning objects not accessible in the callee; and 2) adding facts concerning the parameters.

The problem is that when we generate a function, the actual parameter could be unknown. How do we derive the facts regarding parameters? Csmith solves the problem

by generating the actual parameters before generating the function body. This particular generation order is aligned with control flow, where the function parameters are evaluated before function body.

As the call chain grows, the number of the facts could grow too large and degrade the performance. The pruning is necessary to keep the list size within a manageable range. How does Csmith know which objects are accessible in a function which has not even been created? The accessible objects in a function include 1) global objects, 2) its parameters, and 3) objects indirectly accessible through a global or a parameter pointer.

GTAV follows the control flow and visits statements in the function. After the last statement of the function, the output fact set is passed back to the call site. A callee-to-caller handshake would take place, which is almost the opposite of a caller-to-callee handshake: 1) adding back facts pruned during the corresponding caller-to-callee handshake, and 2)removing facts concerning function parameters.

4.3.5.2 Post-commit GTAV

A function could be revisited by GTAV after its creation. When generating function calls, Csmith may choose to invoke an existing function instead of generating a new one, thus triggering a round of post-commit GTAV.

To increase precision, GTAV performs path-sensitive analyses, which may require a series of functions, if they are invoked by the same call chain, to be revisited, leading to very expensive GTAV operations. In practice, I have observed Csmith being tremendously slow while generating programs with long call chains. To mitigate the problem, Csmith caches the input fact set and output fact set and uses them to bypass unnecessary reanalyses.

By definition, if a function is invoked with identical inputs at two call sites, the outputs should be identical as well. Applying to transfer functions, if the input fact sets are identical between two rounds of GTAV to the function, and the context (see Section 4.3.1 for details) is similar enough (i.e., the key properties of the contexts match each other), the output fact set should be identical. By avoiding unnecessary reanalyses, Csmith is able to improve efficiency.

If a reanalysis is necessary, post-commit analyses are carried out in the identical way to pre-commit analyses: caller-to-callee handshake, function body traverse, and ending with callee-to-caller handshake. Each round of GTAV for a function corresponds to a specific execution path leading to the function. Therefore, the program analyses performed by Csmith are path-sensitive.

4.3.6 Transfer functions for statements

The input fact set to a statement is the union of the output fact set from all its predecessors in the control flow graph. The transfer functions are different for different types of statements, as shown in Table 4.1. The same bypassing mechanism we used on functions is used for post-commit GTAV of statements.

4.3.7 Transfer function for loops

Csmith generates two kinds of loops: for-loops which are for-statements with standard headers and jump-loops which are formed by back-edge jumps, e.g., goto/continue statements. There is no restriction on the size of the loop body, nor is there a limit on the level of loop nesting.

4.3.7.1 Transfer function for for-loops

Transfer function for loops traverses the loop body iteratively until a fixed-point is reached. The first iteration uses the preloop running fact set as the input, and applies transfer functions to the header, and then to the loop body. After the body is analyzed/-

Table 4.1: Type-specific transfer functions for statements

Statement type	**Transfer function description**
Block	Apply transfer functions to its child statements following CFG edges
Call-statement	Apply transfer functions to actual parameters, then to callee function body
Return-statement	Apply transfer functions to returned expression and merge the output fact set to the function's aggregated output fact set
Continue-statement	No change on input fact set, merely carries them into the CFG successor, i.e., the loop header
Break-statement	No change on input fact set, merely carries them into the CFG successor, i.e., the statement immediately follows the loop
Goto-statement	No change on input fact set, merely carries them into the CFG successor, i.e., the labeled target. If goto forms a back edge, loop analyses is be required, as described in Section 4.3.7
If-else-statement	Apply transfer functions to the if-condition, fork the outputs as inputs to both true-block and false-block, then merge the outputs from both true-block and false-block as the final output fact set of the statement
Assignment-statement	Apply transfer functions to RHS then LHS
For-statement	See transfer function for loops in Section 4.3.7
Array-statement	See transfer function for loops in Section 4.3.7

validated, the output fact set is used as the input fact set of the next iteration, starting from the header again. The process repeats until the input fact set to the header is identical to the input fact set to the header during the previous iteration.

While generating a for-loop, statements in the body are committed as they pass pre-commit GTAV. After the loop body generation is complete, Csmith performs a final post-generation analysis/validation. If the validation fails, some of the previously committed statements have to be removed from the loop body to satisfy GTAV. This is called backtracking. Listing 4.1 demonstrates why backtracking is difficult to avoid.

Listing 4.1: Example of loop analysis and backtracking

```
1  int i, j, k;
2  int* p1 = &i;
3  int* p2 = &j;
4  int* p3 = &k;
5
6  for (...)
7  {
8        p3 = p2;
9        *p3 = 0;
10       p2 = p1;
11       p1 = NULL;
12 }
```

It is not until the third iteration $p3$ becomes $NULL$ and a null pointer dereference occurs because of that. The culprit is line 11, which introduces a null pointer. To be able to avoid generating line 9, we would need to perform points-to analysis and use-define chain analysis, and the analyses must compute a fixed point during the generation of every loop statement. I tried that route and found the performance was greatly degraded. The current approach, generating loop statements with optimism and using only one fixed-point computing at the end to validate the loop, proved to be more scalable.

Table 4.2 shows how the fixed-point computing evolves through iterations over the above loop. GTAV fails during iteration three. Instead of throwing out the whole loop body, backtracking tries to minimize the loss by removing the statement that causes the validation to fail, along with all following statements in the loop body. In this example, Csmith deletes statements 9, 10, 11. Note this is not an optimal rescue solution. Removing statement 11 is sufficient to saving the loop. This is achievable by repeatedly removing the last statement from the loop body. The process stops when just enough statements are removed so that the analyses/validation passes. However, this iterated process could require multiple fixed-point computing. I chose the current solution for performance reasons.

Table 4.2: Iterations of points-to analysis

Location	**Iteration 1**	**Iteration 2**	**Iteration 3**
Before line 6	{p1→{i}, p2→{{j}}, p3→{k}}	{p1→{NULL}, p2→{i}, p3→{j}}	{p1→{NULL}, p2→{NULL}, p3→{i}}
After line 6	{p1→{i}, p2→{j}, p3→{k}}	{p1→{NULL}, p2→{i}, p3→{j}}	{p1→{NULL}, p2→{NULL}, p3→{i}}
Before line 8	{p1→{i}, p2→{j}, p3→{k}}	{p1→{NULL}, p2→{i}, p3→{j}}	{p1→{NULL}, p2→{NULL}, p3→{i}}
After line 8	{p1→{i}, p2→{j}, p3→{j}}	{p1→{NULL}, p2→{i}, p3→{i}}	{p1→{NULL}, p2→{NULL}, p3→{NULL}}
Before line 9	{p1→{i}, p2→{j}, p3→{j}}	{p1→{NULL}, p2→{i}, p3→{i}}	{p1→{NULL}, p2→{NULL}, p3→{NULL}}
After line 9	{p1→{i}, p2→{j}, p3→{j}}	{p1→{NULL}, p2→{i}, p3→{i}}	GTAV failed! Skip
Before line 10	{p1→{i}, p2→{j}, p3→{j}}	{p1→{NULL}, p2→{i}, p3→{i}}	GTAV failed! Skip
After line 10	{p1→{i}, p2→{i}, p3→{j}}	{p1→{NULL}, p2→{NULL}, p3→{i}}	GTAV failed! Skip
Before line 11	{p1→{i}, p2→{i}, p3→{j}}	{p1→{NULL}, p2→{NULL}, p3→{i}}	GTAV failed! Skip
After line 11	{p1→{NULL}, p2→{i}, p3→{j}}	{p1→{NULL}, p2→{NULL}, p3→{i}}	GTAV failed! Skip

4.3.7.2 Transfer function for jump-loops

Loops formed by back-edge jumps require fixed-point computing similar to for-loops. Csmith allows the target location to be almost anywhere in the same function; thus, the implicitly formed loop could partially or fully overlap with other loops, complicating the fixed-point computing.

In Listing 4.2, the implicit loop formed by the goto-statement spans lines 3-9; while the for-loop spans lines 6-11. Results from fixed-point computing for one loop could be "unfixed" by the fixed-point computing for the other loop. It seems we have to answer these questions for analyzing overlapping loops: Which loop should we start with to find a fixed point? How to evaluate the effect of overlapping on fixed-point computing? How to determine whether all overlapping loops have reached their collective fixed point?

Instead of addressing these thorny questions individually, Csmith takes a different approach: it considers that all statements involved in overlapping loops form a container loop and applies fixed-point computing to it. All inner loops reach their individual fixed points when the container loop reaches its fixed point.

For the example in Listing 4.2, the container loop spans statements 3-11. At the beginning of each iteration, GTAV determines if it reached the fixed point by considering both inner loops. If both reached a fixed point, no more iterations are needed.

Listing 4.2: Example of overlapping loops

```
1       int i, j, k;
2       int* p1 = &i;
3   label:
4       int* p2 = &j;
5       int* p3 = &k;
6       for (i = 0; i < 3; i++) {
7           if (i < 2) {
8               goto label;
9           }
10          p2 = p3;
11      }
```

4.3.8 Transfer function for expressions

The input fact set to an expression is the output fact set from the previously analyzed expression/statement immediately before a sequence point. Table 4.3 lists different transfer functions for different types of expressions.

4.4 Points-to analysis and validation within GTAV framework

4.4.1 Abstract variables

A points-to fact is defined as $p \rightarrow \{l_1, l_2...l_n\}$ where p is a pointer, and l_i is a location. If the points-to set has only one member, then it is a must points-to. If the set has more than one member, then the locations are may points-to. In Csmith, every accessible location is treated as an **abstract variable**, which is defined as one of the following:

- an actual variable in the C program being generated;
- a field in a structure or union variable; or

Table 4.3: Type-specific transfer functions for expressions

Expression type	**Transfer function description**
Constant	Do nothing
Variable	Validate pointer reference and union field read if applicable; validate and update the read of the location(s) represented by the variable
Unary Expression	Apply transfer function to the operand
Binary Expression	Apply transfer functions to the first operand, then the second operand
Call Expression	Apply transfer functions to the parameters, then to the function body
Assignment Expression	The same as assignment-statement
Comma Expression	Apply transfer functions to the first operand, then the second operand
LHS	Validate pointer reference and union field read if applicable; validate and update the write of the location(s) represented by the variable; validate and update the read of the location(s) represented by intermediate dereferenced pointers

- a special points-to location such as **NULL** or **GARBAGE**. $NULL$ is reserved for null pointers, and $GARBAGE$ represents a location that has not been initialized or has been deallocated.

Abstract variables do not differentiate array members. In other words, abstract variables are field sensitive while array-member insensitive.

The design choice represents a compromise between performance and precision. If we want to reason about the individual members of the arrays, we must reason about the index value of every array access. This is impossible without value range analysis. On the other hand, representing the structure or union fields as individual abstract variables enables field-sensitive program analysis. The cost of field-sensitive analyses is manageable.

When a points-to location is dead, the pointer becomes a dangling pointer. The live range of an abstract variable is defined as: 1) $NULL$ and $GARBAGE$ are always alive; and 2) for other abstract variables, life starts when it is declared, and life ends when the scope of its storage block ends.

Representing all locations with abstract variables gives GTAV a couple of benefits. Merging points-to facts becomes a standard set-join operation, and the sizes of points-to

objects do not have to be explicitly stored as they are implied by the types of abstract variables.

4.4.2 The lattice

Points-to analysis in Csmith represents the points-to set of each pointer as a lattice. The lattice is basically formed by the set of all abstract variables. This set constitutes a bounded (the bound on the height of the lattice is the number of abstract variables) lattice $(L, \vee, \wedge, \top, \bot)$ where:

L is the set of all abstract variables

$\top = \emptyset$

$\bot = \{av_1, av_2, ...av_n\}$ where n is the number of abstract variables

$a \vee b = a \cup b$ for any two sets in the lattice

$a \wedge b = a \cap b$ for any two sets in the lattice

4.4.3 Validation of pointer references

4.4.3.1 Invalid pointer references

Csmith supports pointer references in many ways: reading, writing, dereferencing the pointer and then reading/writing the pointed-to content. Csmith also supports multilevel indirections of pointers and multilevel dereferences. A pointer reference is undefined, and thus should be avoided by Csmith if it constitutes a dangling pointer reference or a null pointer dereference.

Dangling pointer reference: C11 [29, §6.2.4] designates that the behavior is undefined when "an object is referred to outside of its lifetime," and when "the value of a pointer to an object whose lifetime has ended is used." The lifetime of an object depends on its storage duration. Global variables are objects with static storage duration, and thus have a lifetime that is the entire execution time of the program, as do static local variables. Allocated objects have a life time spanning between the memory allocation and memory deallocation. An object with automatic storage (the default for local variables) dies when the scope of its declaring block ends.

Csmith (version 2.1) does not generate allocated objects; therefore, automatic objects are the primary concern. A dangling pointer is a pointer which takes the address of an automatic object, and is still alive when the automatic object dies. The use of such a pointer is commonly called dangling pointer reference, and is undefined.

Null pointer dereference: Null pointer dereference is also considered undefined in C11, which states in Section 6.5.3.2, "If an invalid value has been assigned to the pointer,

the behavior of the unary * operator is undefined." And immediately in footnote 102, it states "Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime."

A program generator can easily avoid null pointer dereference by inserting a check for nullness before pointer dereference. There are undesired complications such as 1) the condition check is cumbersome, especially for complex expressions with multiple pointer dereferences; and 2) protecting pointer dereferences with nullness checks could impact the behavior of the compiler. Since Csmith already performs points-to analysis to be able to avoid dangling pointer references, using the same results to avoid null pointer dereferences incurs no additional cost.

4.4.3.2 Avoiding invalid pointer references

With points-to analysis, invalid pointer references are determined by Csmith when:

- a dereference of p and $NULL$ is in the points-to set of p, or
- a dereference of p and $GARBAGE$ is in the points-to set of p, or
- a read of p and $GARBAGE$ is in the points-to set of p. Note writing to p is valid.

Dereferencing a multilevel indirect pointer is more complicated. Take the expression $**p$, for example; Csmith needs to check points-to sets of both p and $*p$ to validate they do not contain $NULL$ or $GARBAGE$. Furthermore, if $**p$ is read as a pointer, its own points-to set should not contain $GARBAGE$.

Let $*^n p$ denote an expression in which pointer p is dereferenced n times, and n is a nonnegative integer. Let $PS(*^n p)$ be the function that returns the points-to set of $*^n p$. Then the PS function can be generalized to a multilevel indirection pointer as:

1. for special locations $NULL$ and $GARBAGE$, $PS(NULL) = PS(GARBAGE) = GARBAGE$;
2. when n = 0 (the base case), $PS(p)$ is the points-to set of p;
3. when n = 1, $PS(*p)$ is $\forall$ q $\in PS(p)$: $\bigcup PS(q)$;
4. generalizing the rule derived from the base case to n = 1: when n = i, $PS(*^i p)$ is $\forall$ q $\in PS(*^{i-1} p)$: $\bigcup PS(q)$.

The formal validation for expression $*^n p$ can be expressed in inductive rules:

1. when n = 0 (the base case), writing to p is always valid and reading from p is valid if $GARBAGE \neq p$;

2. when n = 1, writing to $*p$ is valid if ($NULL$ or $GARBAGE$) $\notin PS(p)$; reading from $*p$ is valid if writing to $*p$ is valid and $GARBAGE \notin PS(*p)$;

3. Generalizing the inductive rule from the base case to n = 1: when n = i, writing to $*^i p$ is valid if ($NULL$ or $GARBAGE$) $\notin PS(*^{i-1} p)$; and reading from $*^i p$ is valid if writing to $*^i p$ is valid and $GARBAGE \notin PS(*^i p)$.

While generating a type T expression, Csmith considers all variables with T or its derived types such as $T*$, $T**$, etc. GTAV determines which pointers are valid candidates, i.e., valid after proper levels of dereference according to the above validation. For example, pointer p of type $int16*$ is a qualified candidate for generating a to-be-modified expression of type $int32$, as long as p is not a null or dangling pointer.

Post-commit points-to analysis is identical to pre-commit points-to analysis. However, Csmith acts differently when the analysis/validation fails. At generation time, the invalid pointer reference is discarded, while a failed post-commit points-to analysis/validation would cause Csmith to discard the construct that triggered the revisit. See Section 4.3.3 for details.

4.4.3.3 Opportunistic validation of pointer references

Csmith was originally designed to generate only valid pointer references to serve the purpose of random differential testing of compilers. However, we find the random programs would be valuable for testing tools performing pointer-related static analysis, if we relax the restriction. The basic idea is to have Csmith purposely generate a few invalid pointer references in a random program, and challenge a static analyzer to find such references.

To fulfill the goal of testing program analyzers, Csmith provides a command line option that symbolizes the desired chance of invalid pointer references. If the command line option is 30%, then there is a 30% chance that Csmith would ignore failed GTAV while generating pointer references. Csmith also generates a marker when such an invalid pointer reference is generated.

4.4.4 Validation of points-to analysis itself

Points-to analysis is used to validate pointer references, but how do we validate the points-to analysis itself? Points-to analysis is not trivial for programs with unrestricted

pointer references. While Csmith simplifies the problem by disallowing certain pointer operations, such as the pointer arithmetics, the analysis remains challenging and error-prone.

We learned that fact the hard way. While trying to support pointers in Csmith, we found numerous bugs in points-to analysis and a great amount of effort was spent to find and fix them. Those bugs had led to either crashes in Csmith or false-positive bug reports.

It became clear that we needed to implement a validation mechanism for our points-to analyzer. This is achieved by inserting assertions about points-to facts in the random programs. For example, if the points-to set for pointer p is $\{v_1, v_2\}$, the assertion would be $assert(p == \&v_1 || p == \&v_2)$. With these assertions, we can easily spot bugs in points-to analysis because they lead to execution-time assertion failures.

However, the assertions could not fully express the points-to analysis results for two reasons: 1) The assertions are static, so they have to be context insensitive. Even though Csmith is capable of context-sensitive points-to analysis, it has to merge points-to analysis results obtained under various contexts to yield the static assertions. Consequently, the precision and the validation strength are decreased; 2) The C language syntax does not allow certain points-to facts to be asserted. For example, if pointer p in a function points to a local variable v in the caller function, the assertion $assert(p == \&v)$, while semantically correct, would be rejected by a compiler because v is not in scope. In addition, the special location $GARBAGE$ is impossible to express in an assertion.

Despite the above two limitations, the assertions proved to be very helpful in finding bugs in the points-to analysis. The assertions are disabled by default after the points-to analysis in Csmith becomes stable enough.

4.5 Side-effect analysis within GTAV framework and the evaluation order problem

C11 [29, §5.1.2.3] states that "Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects." Csmith simplifies side-effect analysis by focusing on three primary effects: read/write on volatile variables, read of nonvolatile variables, and write of nonvolatile variables.

Side-effect analysis is essentially engineered to solve the evaluation order problem. C language only provides a list of sequence points and a mandate that an expression before a sequence point must be evaluated before an expression after the same sequence point. C11 leaves the evaluation order of subexpressions in a full expression to be mostly unspecified. It states, "The order in which subexpressions are evaluated and the order in which side effects

take place, except as specified for the function-call (), &&, ||, ? :, and comma operators (6.5)" is unspecified. It [29, §6.5.2.2] clearly reiterates that "the order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call" is also unspecified. The only specified evaluation orders, as mentioned in C11 [29, §6.5], are:

- The function parameters must be evaluated before the function body,
- For expression $E1$ && $E2$, $E1$ is always evaluated first. If $E1$ is evaluated to False, $E2$ is not evaluated,
- For expression $E1$ || $E2$, $E1$ is always evaluated first. If $E1$ is evaluated to True, $E2$ is not evaluated,
- For expression $E1?E2 : E3$, $E1$ is always evaluated first. Then $E2$ or $E3$ is evaluated based on the result of $E1$,
- For Expression "$E1$, $E2$", $E1$ is always evaluated first. Then $E2$ is evaluated no matter what.

The ambiguity in evaluation orders is not necessarily a bad design. Compilers are likely to take advantage of it and rearrange orders between operations to achieve the best performance. But it is harmful to Csmith because it introduces ambiguity, also because the following behavior is undefined in C11 [29, Annex J2]: "Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored (6.5)." Listing 4.3 shows an example of ambiguous code due to unspecified evaluation orders.

Because the evaluation orders between two sequence points are unspecified, a modifying operation on an object could happen either before or after another reading/modifying operation of the same object, causing the outcome to be indeterministic. If two volatiles are accessed between the same two sequence points, their access orders are not deterministic either. Such indeterminism is called the evaluation order problem. It leads to failure of random differential testing.

Csmith solves the evaluation order problem by detecting conflicts in unordered operations. Two unordered operations on a same location are determined to be in conflict if 1) one of them is read and the other one is write (read-and-write conflict) or 2) both are write operations (write-and-write conflict).

Listing 4.3: Example of ambiguity due to unspecified evaluation orders

```
void main()
{
    int a[2] = {0, 0};
    int x = 0;
    a[x] = ++x;
    // could print 0 1 or 1 0
    printf("%d %d", a[0], a[1]);
}
```

4.5.1 The algorithm and the lattice

To avoid dependency on unspecified evaluation orders, we implement in Csmith a side-effect analysis and validation of abstract variables. The algorithm is detailed below:

- Create an empty side-effect context C after a sequence point,
- Before the generation reaches another sequence point, validate all generated but not committed expressions against C. Reject any expression E if it meets one of the following conditions (and commit E otherwise):
  - An abstract variable is read in the E while written in C
  - An abstract variable is written in E while read in C
  - An abstract variable is written in E while written in C
  - A volatile abstract variable is read/written in E while there is a volatile access in C (see Section 4.5.3 for details)
- Add effects of the committed expressions to C

Abstract variables are field-sensitive; therefore, Csmith allows a write of one field of a structure variable to be unordered with a read or write of another field of the same variable, knowing the writes are not in conflict.

The lattices for side-effect analysis are bounded as (L, $\vee$, $\wedge$, $\top$, $\bot$). This is almost identical to that of points-to analysis lattice where:

L is the set of all abstract variables excluding special locations $NULL$ and $GARBAGE$

$\top = \emptyset$

$\bot = \{av_1, av_2, ...av_n\}$ where n is the number of abstract variables

$a \vee b = a \cup b$ for any two sets in the lattice

$a \wedge b = a \cap b$ for any two sets in the lattice

4.5.2 The implementation

Side-effect analysis is implemented in Csmith with two sets of abstract variables representing read-effects and write-effects, and a flag indicating whether there is a volatile access in the context. An abstract variable is included in the appropriate set when it is read or written. Note: a read or write on a field of a union variable is expanded to reads/writes of all fields of the variable.

Side-effect analysis depends on points-to analysis. For expressions with simple pointer dereference $*p$, the analysis conservatively includes all the points-to locations, a.k.a. abstract variables, of p to the appropriate set. For expressions with multilevel pointer dereferences $*^i p$, the possible locations of each dereferenced level are included to the appropriate set.

At generation time, an expression is validated against the accumulated side-effect context before it is committed to the tree. Although the sequence points for committed code stay unchanged, an expression with pointer dereference(s) is likely to have different read or write effects when the pointer value is changed. Thus we need post-commit side-effect analysis/validation.

4.5.3 Side-effect validation for volatiles

C11 [29, §5.1.2.3] defines accessing a volatile object, like a modifying on an object, as a side effect. It [29, §5.1.2.3] states that the least requirements on volatile object accesses are "Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine." In addition, C11 [29, §6.7.3] leaves room for ambiguity by stating, "What constitutes an access to an object that has volatile-qualified type is implementation-defined." The standard implies that for objects defined with a volatile type, accesses must be done before the next sequence point; but otherwise, merging, reordering, and word-size change is allowed [25].

To be conservative, Csmith assumes that a read from or write to a volatile object is a side effect, and there should be no more than one read or write of volatile object(s) between two sequence points. This restriction is inherited from randprog, a program on which Csmith is based. Randprog was designed to find bugs in compilers' optimization of volatile object accesses, as opposed to general compiler bugs targeted by Csmith.

4.6 Union-field analysis and validation within GTAV

Union variables may have padding bytes because of a nonaligned member field. C11 [29, §6.2.6.1] states, "When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values." Specifically for unions, it [29, §6.2.6.1] states, "When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values." Consequently, C11 [29, §6.2.6.1] stipulates that reading "the value of a union member other than the last one stored into" is undefined.

To remove ambiguity caused by this undefined behavior, Csmith has to know which field of a union variable was last written into, and validate that a subsequent read from the union is through that field. Like side-effect analysis, pointers complicate the union-field analysis because a write or read of union variable fields could be expressed with pointer dereference(s). Csmith uses union-field analysis/validation to achieve the above objectives and performs post-commit analysis/validation when necessary.

4.6.1 The lattice

The lattices for union field analysis are bounded as (L, $\vee$, $\wedge$, $\top$, $\bot$) where:

L is the integer set $< 0, 1...n >$ where n is the maximum allowed field id of unions

$\top = \emptyset$

$\bot = \bot$

$a \vee b = \bot$ for any two sets in the lattice

$a \wedge b$ is undefined

When a field f_1 of union variable U is definitely written into, the analysis concludes that the last written field of U is f_1 and denotes the fact as $U : f_1$. The possible states for a union variable are {$\top$, $\bot$, f_1, f_2, ... f_n} where n is the number of named fields in the union type.

A join operation, which occurs typically at the end of an if-else-statement, leads to the bottom, which means the last written field of the union variable is indeterministic, thus, no field in the union variable can be read. This transition allows a fast traversal of the lattice. Once in the bottom state, The union variable can only be written. If the field that is written into is known definitely, then the union variable could escape from the bottom.

Definite write to a union field is defined as either 1) a write to a union field without pointer dereferences, or 2) the pointer being dereferenced has a must points-to relationship

with the union field. Indefinite write to a union field is through a pointer that may point to several locations. Such a write causes the union-field state to be transitioned to the bottom immediately.

4.6.2 Validation of union-field reads

A read from the field f_i of a union variable U is valid only when the last written field is f_i. The validation simply compares the union-field state with the field to be read.

However, this rule is somewhat overstrict and prevents a common practice that is essentially equivalent to what C++ calls a "reinterpret cast." If we do not compare the executions on different platforms, it is safe to write to a union field and read from another when both fields are of integer type, and the former type is wider than the latter type. For example, writing to a 32-bit int field and reading from an 8-bit byte field is generally safe. This intraplatform safety derives from C11's basic mandate on memory layout quoted in the following two rules:

- "A pointer to a union object, suitably converted, points to each of its members, and vice versa." [29, §6.7.2.1] and
- "Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined." [29, §6.2.6.1]

While Section 6.7.2.1 states that all fields of a union object must begin at the same location in memory, Section 6.2.6.1 demands that bytes for an integer must be contiguous. If we write four bytes into a union field, and read the first two bytes, there is no risk of reading padding bytes, which is the source of unspecified values. The only ambiguity left is the endianness. However, endianness is invariant on a given platform, and we do not perform differential testing across platforms. Thus, there is no ambiguity.

Csmith supports a command line option that enables less restrictive union reads (as described above) to be generated. It also supports another relaxation on the C11 restriction by allowing reading a char pointer field when the last written field is also an integer type pointer. This gives Csmith the freedom to use the char* field to read/write contiguous bytes of a wider integer. Listing 4.4 is an example:

Listing 4.4: Example of contiguous byte accesses through char*

```
union
{
    char* pChar;
```

```
    short*  pShort;
} U;

short i = 0xFFFF;
U.pShort = &i;
// clear either lower byte or upper byte of i,
// depending on the endianness
*(U.pChar + 1) = 0;
```

4.6.3 Validation of assignment between overlapping objects

Unions introduce an interesting undefined behavior that is specified in C11 [29, §6.5.16.1]: "If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined."

Most times, different objects occupy nonoverlapping memory locations. Unions are unique as they are designed to share memory space between members. Csmith has a simple validation to ensure no assignment, whether simple or compound, occurs between two fields of the same union object. Although it appears silly to assign a union field to itself, it is permitted by both the standard and Csmith.

4.7 Guaranteed convergence of fixed point of the loop transfer function

Could the transfer function for loops, hereafter denoted as TF_{loop}, ever fail to reach a fixed point? The number of inputs to and outputs from function TF_{loop} is limited to the number of abstract variables in the random program; therefore, TF_{loop} is a discrete function. According to the Knaster Tarski theorem [60], which is applicable to discrete functions, any order-preserving function on a complete lattice has a fixed point.

I have shown that lattices for points-to analysis, side-effect analysis, and union-field analysis are complete. The lattice on which TF_{loop} operates is the conjunction of these three lattices and thus is also complete.

For each iteration, the inputs to TF_{loop} are the join of the outputs and inputs of the previous iteration. This implies that TF_{loop} is order-preserving. Specifically:

$TF_{loop}(X) = X$ when $X = \top$

$TF_{loop}(X) > X$ for all other inputs

Because TF_{loop} is order-preserving and the lattice is complete, the loop transfer function will always converge on a fixed point. This guarantees that GTAV always halt. As a matter of fact, Csmith has an internal assertion which states that the number of iterations for loop

analysis should be less than eight. The assertion has never failed throughout the years of random program generation.

CHAPTER 5

EVALUATION

> Those who trust to chance must abide by the results of chance.
>
> Calvin Coolidge

We developed Csmith for the purpose of testing compilers. Towards this end, we have conducted experiments to evaluate how (potentially) good Csmith is at finding compiler bugs under either uncontrolled or controlled environment. Many of the results presented in this chapter are already published in the PLDI paper in which we firstly introduced Csmith [75].

5.1 Uncontrolled opportunistic compiler bug finding

Our first experiment was uncontrolled and unstructured: Over a period of four years and three months (March 2008–July 2012), we used Csmith to generate random programs and opportunistically found and reported bugs in a variety of C compilers. We found bugs in all the compilers we tested. Overall, we found and reported 461 valid and distinct compiler bugs, many of them classified as high-priority bugs. We reported bugs to 12 different C compiler development teams. Five of these compilers (GCC, LLVM, CIL, TCC, and Open64) were open source. Six were commercial products (Microsoft Visual C++, Intel C Compiler, LANCE C Compiler, Sun C Compiler, CodeWarrior, ARM C Compiler). The 12th, CompCert, is publicly available but not open source. More than 96% of the bugs were found in GCC and LLVM.

5.1.1 What kinds of bugs are there?

The compiler defects we observed can be generally classified into two large categories: 1) the defects that caused the compilation to terminate prematurely, and 2) the defects that caused miscompilations. The former has symptoms that manifest at compile time, while

the latter only manifests when the compiler's output is executed. We have observed the following kinds of compile-time errors:

- Assertion violations or other internal compiler errors (ICE)
- Crashes due to memory-safety problems such as null pointer dereferences
- Crashes due to out of memory
- The compiler is extremely slow or simply hangs on a reasonably sized input, forcing us to kill the compiler process

We say that a compile-time error has occurred whenever the compiler process exits with a status other than zero or fails to produce executable output. The bug behind such an error is called a *compile-time bug*. In most cases, the observed error is a crash, falling into one of the first three kinds of behaviors. The fourth kind occurs rarely. For simplicity, we use the term **compile-time error** and **crash error** interchangeably.

When presented with random programs, the observed behaviors of miscompilations by compilers include:

- Generating an executable that computes wrong result(s)
- Generating an executable that terminates abnormally for a source program that should not
- Generating an executable that terminates normally for a source program that should loop forever
- Generating an executable that loops forever for a source program that should terminate

We refer to these miscompilations as *wrong-code errors*. The bug behind such an error is called a *wrong-code bug*. A *silent wrong-code error* is considered dangerous to compiler users. The error occurs in a program that was produced without any warning from the compiler; i.e., the compiler silently miscompiled the source program.

5.1.2 Experience with commercial compilers

There exist many commercial C compilers. Licensing issues prevent us from obtaining a copy of most of them and running tests easily. The ones we chose to study are fairly popular and were produced by what we believe to be some of the strongest C compiler vendors: Microsoft, Intel, Sun Microsystems (now part of Oracle), among others.

Csmith was able to find wrong-code errors and compile-time errors in each of these commercial compilers quickly, typically within a few hours of testing. However, our bug reports to the vendors received lukewarm responses. That is probably because we are not paying customers, because our findings represent potentially bad publicity, or because the compiler teams regard random testing as unimportant.

We therefore simply tested these commercial compilers until we found a few compile-time errors and/or a few wrong-code errors, reported them, and moved on. Among those bugs found, we reported eight for armcc, two for CodeWarrior, one bug for VC++, and three for icc.

5.1.3 Experience with open source compilers

Our experience with open source compiler teams, on the other hand, is encouraging and smooth. This, combined with several other reasons, led us to direct the bulk of our testing effort towards GCC and LLVM, two well-known open source compilers.

First of all, compiler testing is inherently interactive: we expect feedback from the compiler writers; either to admit the bug and fix it, or dismiss the bug with justifications. We welcome both scenarios. LLVM and GCC teams are the most responsive to our bug reports.

Secondly, random testing tends to find bugs with different frequency of occurrence; some are hard-to-catch one-in-a-million bugs, and some are observed with many test cases. The former could be drowned in a sea of the latter. Thus, bug fixes from developers, counter-intuitively, lead to more bugs to be found by random testing. The testing proceeds most smoothly with help from developers who eliminate the easy bugs, making tricky bugs more observable.

Both the GCC and LLVM teams were very responsive to our bug reports. The LLVM team in particular fixed bugs quickly, often within a few hours and usually within a week. The GCC team has a more loosely organized structure, and the developers have more flexible working schedules. Nevertheless, their speed of fixing bugs we reported gradually picked up, at some points becoming comparable to that of the LLVM team.

Thirdly, we prefer dealing with open source compilers because their development process is transparent: We can join the mailing lists, participate in discussions, and see fixes as they are committed. The mutual interaction benefits both sides: they benefit from our testing, and we benefit by learning their process and insight into compilers. The knowledge is then applied to improve Csmith.

Finally, we want to help harden the open source development tools that we and many others use daily. Knowing our years of testing resulted in better open source compilers is greatly satisfying.

We started reporting bugs to both LLVM and GCC developers in March 2008. By the end of the testing period, we have found numerous bugs in both of them using Csmith and random differential testing. For GCC, we reported 164 bugs, including 18 from Arthur O'Dwyer who is not in our group but volunteered to test compilers with Csmith. Meanwhile, 267 bugs, representing about 2% of all compiler-specific bugs, are reported to the LLVM team.

Most of our reported bugs have been fixed, and 31 of the GCC bugs were marked by developers as P1: the maximum, release-blocking priority for a bug. LLVM developers do not assign priorities to bugs. As of April 2014, we have reported 445 bugs in GCC and LLVM, out of 461 in total across all tested compilers.

An error that occurs at the lowest level of optimization is pernicious because it defeats the conventional wisdom that compiler bugs can be avoided by turning off the optimizer. Table 5.1 counts these kinds of bugs, causing both compile-time and wrong-code errors, that we found using Csmith.

In the next chapter, I will present more details on the bugs we found in GCC and LLVM, while studying the causes, origins, and the fixes of compiler bugs.

5.1.4 Testing CompCert

CompCert [34] is a verified, optimizing compiler for a large subset of C; it targets PowerPC, ARM, and x86. We dedicated many machine-hours to test this compiler from version 1.6 up to version 1.8 in 2010. The effort was initially carried out on a PlayStation powered by PowerPC. It was later expanded to x86 machines when CompCert added a x86 back end in version 1.8. We found and reported six bugs in CompCert's unverified front-end code. The bug reports, coupled with other factors, led the main CompCert developer to expand verifications of CompCert. The team gave formal definitions of CompCert's handling of integer promotions and other implicit casts, and proved it is correct.

Table 5.1: Compile-time and wrong-code bugs found by Csmith that manifest when compiler optimizations are disabled (i.e., when the –O0 command-line option is used)

	GCC	**LLVM/Clang**
Compile-time	2	10
Wrong-code	2	9
Total	4	19

We also found two compile-time errors that caused the assembler to fail. One of them was caused by CompCert's back end which generates an overflowed 16-bit displacement used for stack frame allocation. The error would cause stack overflow if not caught by the assembler. In addition, we found a wrong-code bug related to miscompiling volatiles.

It is impressive to see that middle-end bugs, which are common in both GCC and LLVM, are missing in CompCert. The apparent robustness of CompCert supports a strong argument that developing compiler optimizations within a proof framework increases the compiler's correctness.

However, CompCert is not fully compliant to C language standards. We had to limit the expressiveness of Csmith to generate test cases that could be compiled by CompCert.

5.2 Comparison with the other random program generators

5.2.1 Previously published random C program generators

DDT [44] is a testing system that applies directed differential testing techniques to C/C++ compilers. It is primarily a random program generator based on a stochastic grammar for C, with added capabilities such as reducing error-inducing test cases, eliminating false positives introduced by undefined runtime behaviors, and testing compilers locally or remotely. DDT was mostly developed by William McKeeman while working for Digital Equipment Corporation. DEC (now part of Hewlett-Packard) released the software under a license term that is free to use and free to distribute for noncommercial purposes. We obtained the source code online and made several changes to suit our needs:

- Fixing syntax errors to be able to run with Tcl/Tk 8.5
- Outputting generated test cases in a separate directory from the source code
- Outputting all generated programs instead of only the error-inducing ones

Quest [37] is an open source software developed by Christian Lindig. It is released under a BSD 3-Clause license. I obtained the latest version of source code from Google code [36], and compiled it with Objective Caml 3.12. Quest generates C programs that can uncover bugs in a C compiler. The generated code passes complex arguments from callers to callee, and thus tests the translation of function calls. Quest has uncovered bugs in production-quality compilers such as GCC.

randprog [14] is an open source software developed by Eric Eide and John Regehr. I obtained the latest version (1.0.0) of source code from the authors' web site [13], and

compiled it with GCC 4.6.3. randprog is a significantly enhanced and tailored version of a publicly available program generator written by Bryan Turner [65]. To differentiate these two, we will refer to the original generator as **randprog0**, and the enhanced one as randprog. Randprog focuses on testing volatile variable accesses. It creates C programs that perform computations over signed and unsigned integer variables. However, it does not support characters, arrays, pointers, structures, unions, or floating-points.

5.2.2 Generating performance

I ran the four random C program generators, including Csmith, on a 64-bit Ubuntu 12.10 system with an 8-core Intel i7-2720QM CPU for a period of 12 hours. The hour-by-hour generation results are listed in Table 5.2. Compared to other generators, Csmith generates the smallest number of files in the same amount of time. In terms of KLOC (kilo-lines of code measured by cloc), Csmith beats DDT, but falls far behind Quest and randprog. Csmith is slow in generating lines or files compared to other generators. This is not surprising considering how complex GTAV is in Csmith, and accordingly how many CPU cycles are spent on it. Additionally, a line in Csmith, sometimes composed of chained function calls or integer operations, is typically longer than a line produced by other generators.

However, both file number and KLOC are coarse measurements of the program size. We instrumented Csmith so that it outputs program size at a finer granularity. Table 5.3 shows the averages of generated constructs hourly.

Table 5.2: Comparison of hour-by-hour generating performance of random C program generators

	DDT		Quest		randprog		Csmith	
hour	file#	KLOC	file#	KLOC	file#	KLOC	file#	KLOC
1	11,336	255	178,565	283,592	98,082	83,925	4,063	5,097
2	10,744	240	179,522	285,212	83,137	70,784	3,821	4,756
3	8,125	183	179,300	284,784	82,019	69,300	3,908	4,866
4	11,571	259	178,731	283,877	81,732	68,992	3,190	3,992
5	11,459	258	179,349	284,886	77,832	65,931	3,892	4,891
6	11,668	262	179,261	284,706	77,435	65,618	3,597	4,543
7	12,280	274	178,023	282,952	88,325	75,462	4,035	5,024
8	12,842	288	177,979	282,703	93,923	80,482	3,901	5,024
9	12,708	284	177,919	282,696	94,984	81,283	3,920	4,876
10	12,352	276	177,861	282,632	91,643	78,709	3,971	4,917
11	13,258	297	177,958	282,492	117,122	98,492	4,986	6,329
12	12,157	271	178,890	283,106	82,625	69,353	3,598	4,574
Mean	11,708	262	178,613	283,637	89,072	75,694	3,907	4,907

Table 5.3: Hourly generating performance of Csmith at program construct levels

hour	types	variables	functions	blocks	statements	expressions	backtracks	time
1	12	5754.8	8.4	85.6	195.9	1472.5	5.7	0.80
2	12	5710.6	8.4	84.3	193.5	1446.8	5.7	0.85
3	12	5668.8	8.3	84.7	194.1	1460.9	5.6	0.83
4	12	5685.0	8.4	85.1	195.2	1466.5	5.6	1.04
5	12	5617.7	8.3	83.4	191.3	1436.1	5.7	0.81
6	12	5784.0	8.5	85.8	196.6	1479.1	5.8	0.90
7	12	5711.4	8.4	85.2	195.4	1473.0	5.6	0.80
8	12	5800.2	8.4	86.2	197.6	1486.2	5.8	0.84
9	12	5650.4	8.4	84.9	194.5	1471.8	5.7	0.84
10	12	5652.0	8.3	84.2	192.9	1445.1	5.7	0.82
11	12	5787.5	8.4	86.5	192.3	1493.8	5.8	0.94
12	12	5783.4	8.5	86.3	197.7	1491.6	5.8	0.91
Mean	12	5717.2	8.4	85.2	194.8	1468.6	5.7	0.86

The generation time per test case by Csmith falls mostly within a range between 0.25 seconds and 1.25 seconds. But there are extreme cases in which Csmith needs hundreds of seconds to complete the generation. Figure 5.1 shows the time distribution for a 12-hour generation period.

Table 5.4 shows the 10 worst cases in terms of generation time in the 12-hour period.

Compared to the overall averages, the 10 worst cases generate more variables, blocks, statements, and expressions. But the increases in the generated program constructs are penalized by the even greater increase in generation time that is over 150 times more

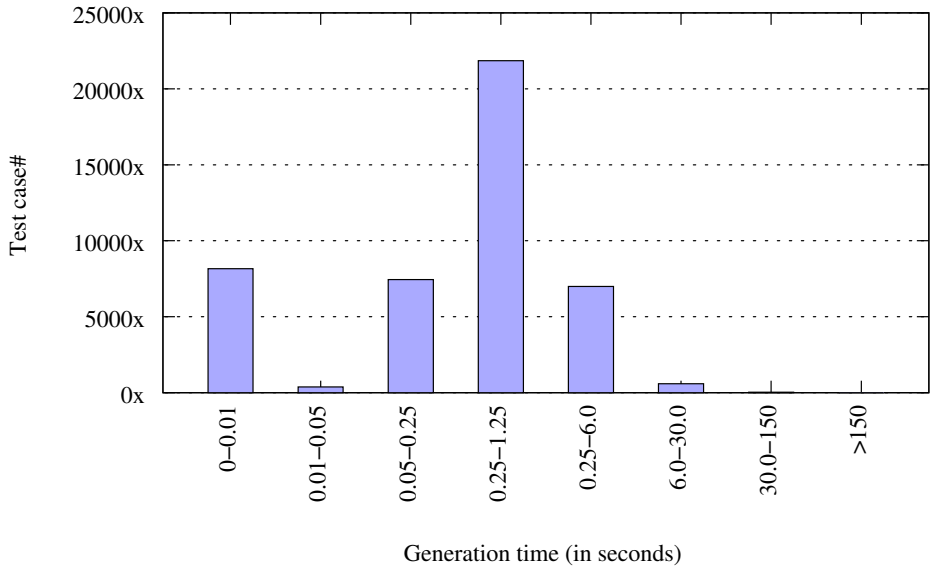

Figure 5.1: Generation times (in seconds) per test case

Table 5.4: 10 test cases taking the longest time to generate by Csmith

test case#	variables	blocks	statements	expressions	backtracks	time (seconds)
1	16681	163	359	2270	10	338
2	13738	227	528	4002	12	222
3	17781	193	468	3047	10	131.1
4	10944	125	295	2263	14	117.9
5	11175	226	532	4126	7	103.1
6	15753	211	510	3721	4	89.13
7	9748	212	524	5069	2	83.45
8	16716	169	383	3046	18	80.04
9	8980	151	349	2717	11	70.28
10	19979	126	303	1835	17	69.7
Mean	14149.5	180.3	425.1	3209.6	10.5	130.5

than the average generation time. Furthermore, the increases of program constructs are accompanied by increases of backtracks which may partly explain the increase of generation time. When Csmith has to backtrack, previously generated and validated tree nodes have to be removed from the tree, causing setbacks to the generation process.

In practice, we found the test cases generated in seconds are almost as valuable as those taking several hundred seconds in terms of bug-finding power. Thus, we empirically set a heuristic time limit to Csmith. If Csmith does not finish generation within the time limit, it is killed and moves to the next test case.

5.2.3 Compiler code coverage

Because of the large number of bugs we find with Csmith, we hypothesized that randomly generated programs exercise large parts of the compilers that were not covered by existing test suites. To test this, we built GCC and LLVM with code-coverage instrumentation enabled. We then used the instrumented compilers to compile their own test suites, and also to compile 10,000 Csmith-generated programs on top of their own test suites. Table 5.5 shows that the incremental coverage due to Csmith is so small as to be a negative result. We believe there are several possible explanations: 1) These code coverage metrics are too shallow to capture effects of random programs, and Csmith is likely generating useful additional coverage in terms of deeper metrics such as path or value coverage; 2) when we report a bug to GCC or LLVM developers, in most cases, the failure-inducing test case is reduced and included to the compilers' test suite; therefore, Csmith is defeated by its own success when fighting for better code coverage with the test suites.

To measure Csmith's code coverage on compilers against other random generators, we

Table 5.5: Augmenting the GCC and LLVM test suites with 10,000 randomly generated programs did not improve code coverage much

		Line Coverage	Function Coverage	Branch Coverage
GCC	make check-c	75.13%	82.23%	46.26%
	make check-c & random	75.58%	82.41%	47.11%
	% change	+0.45%	+0.13%	+0.85%
	absolute change	+1,482	+33	+4,471
Clang	make test	74.54%	72.90%	59.22%
	make test & random	74.69%	72.95%	59.48%
	% change	+0.15%	+0.05%	+0.26%
	absolute change	+655	+74	+926

run a set of random generators for the same length of time, and measure the code coverage on GCC on an hourly basis. The set of random generators includes the previously published C program generators, Csmith, and a naive random generator. The naive generator basically outputs random ASCII characters. In addition, I created two Csmith variants, one with loop generation disabled (Csmith-no-loop), and one with both loop and branch disabled (Csmith-no-loop-branch). The coverage data are shown in Table 5.6. As the baseline comparison, we measured that an empty file could achieve 3.9% line coverage, 3.6% function coverage, and 1.7% branch coverage on GCC's source code.

At its full expressiveness, in 12 hours, and unadjusted for the baseline, Csmith is able to generate random test cases covering at least 42.7% of GCC source code lines, 50.9% of GCC source code functions, and 31.9% GCC source code branches. The above coverage is achieved by compiling random test cases on a single platform (x64) and with optimization levels including -O0, -O1, -O2, -O3, and -Os. The coverage data should be higher if the random test cases are compiled on more platforms and/or with more compiler flags.

Table 5.6: Csmith covers significantly more compiler code than other random C program generators at the end of a 12-hour generation period

	Line Coverage	Function Coverage	Branch Coverage
Naive	6.8%	7.4%	3.6%
DDT	31.3%	39.8%	23.0%
Quest	29.9%	39.9%	19.8%
randprog	37.5%	46.7%	27.8%
Csmith-no-loop	45.2%	46.5%	31.1%
Csmith-no-loop-branch	44.9%	46.4%	30.8%
Csmith	42.7%	50.9%	31.9%

The above percentages are computed against totals of 376,638 source lines, 24,576 functions, and 440,049 branches. The coverage-enabled GCC is based on a revision pulled on April 19, 2013. In spite of seeming a small improvement in terms of code coverage achieved by Csmith over other random generators, a 1% increase in branch coverage actually means 4,400 more branches are covered, implying that Csmith could explore a large portion of the compiler space while other random generators could not.

We limit the random program generation to 12 hours before we start measuring the coverage for practical reasons: the instrumented compilers are very slow to compile programs, and there is a large number of random test cases to be compiled to obtain the coverage data. It could take several days to compile the random programs generated in an hour. The slowdown in compilation speed is more visible in LLVM/Clang than in GCC.

Is it possible that the coverage data would change dramatically if we extended the random program generation past 12 hours? I measured the accumulated coverage data for every three hours of random program generation, and plotted the data on a graph. Figure 5.2 shows that the random generators are able to achieve near maximum coverages within the first three hours of generation. This is another proof that coverage does not directly translate into bug-finding power as we will present in the next section that the random generators are able to find new compiler bugs well beyond the first three hours.

Adjusted for the baseline coverages obtained by a blank input, Csmith was able to achieve a good percentage of the code coverage achieved by GCC's test suite as shown in Table 5.7. It is impressive considering GGC's test suite was built over the decades by a team of compiler writers who know the compiler inside out. By just randomly generating C programs in a 12-hour period, Csmith is able to achieve more than half of the test suite's coverage.

It also worth noting that Csmith achieves a higher percentage on branch coverage than on line coverage compared to coverages reached by GCC's test suite. This implies that the branches Csmith explored have fewer numbers of LOC than the branches explored by the test suite on average. It is possible that these branches, not usually taken when compiling nonrandom code such as the test suite, are more bug-lurking. As the result, Csmith was able to find bugs that the test suite could not.

5.2.4 Bug-finding power

Code coverage is the indirect indication of expressive power. However, no matter how expressive a random program generator is, it has to be able to find bugs to prove its

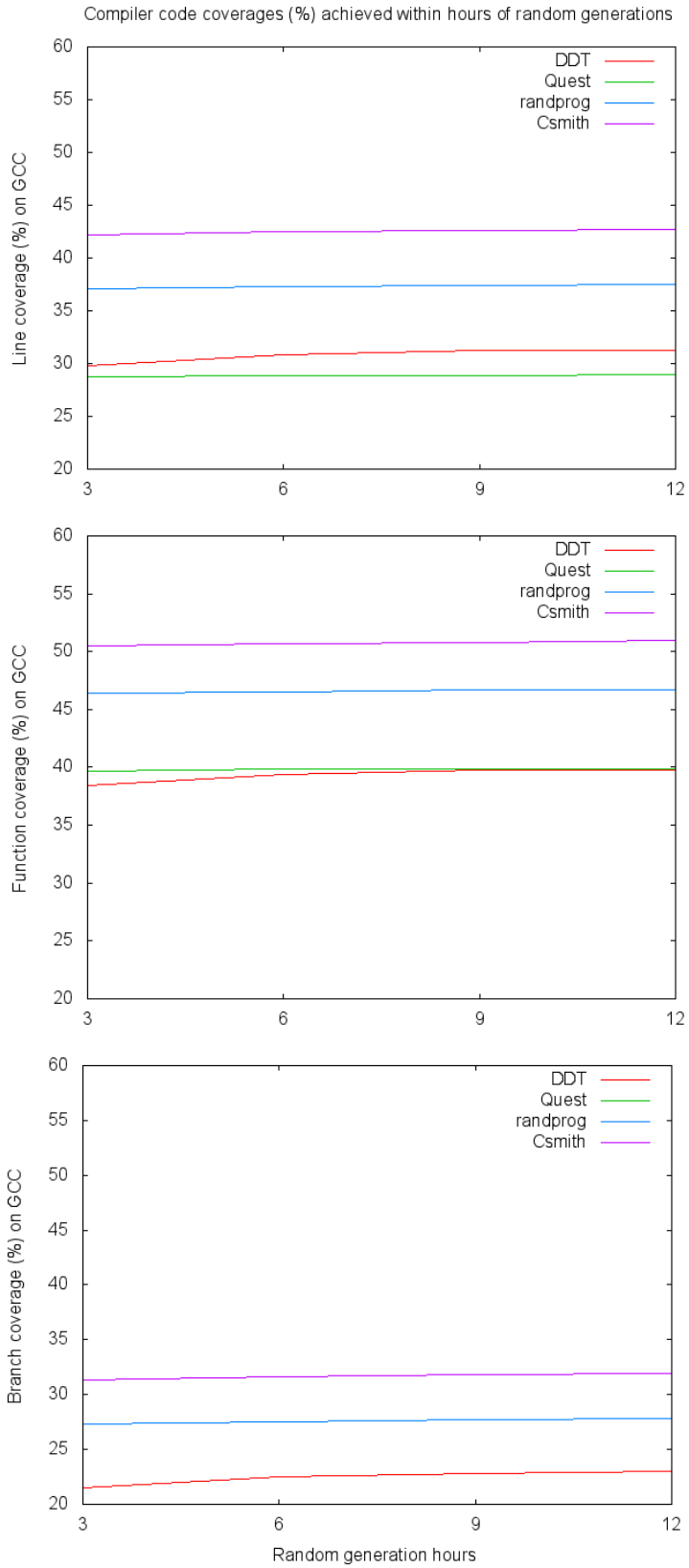

Figure 5.2: Random generators tend to reach near-maximum code coverage on compilers within the first three hours

Table 5.7: Compiler code coverage achieved by Csmith vs. code coverage achieved by GCC's test suite

	Line Coverage	**Function Coverage**	**Branch Coverage**
make check-c	71.23%	78.63%	44.36%
Csmith	38.8%	47.3%	30.2%
Percentage	54%	60%	68%

worthiness; therefore, we need to compare Csmith with other random program generators in terms of bug-finding power.

Here we formally define bug-finding power as the number of distinct compiler bugs found within a certain period of time. We emphasize "distinct" because random programs tend to explore the same set of compiler source code over and over again. A compiler bug located on a "hot" path (a path that is executed frequently) is likely to be triggered by random programs hundreds of times in a few hours. While the number of incidents may help the compiler developers to debug, the duplication adds little value from the perspective of software testing; therefore, we count only distinct bugs while measuring bug-finding power.

How do we know whether two bugs are duplicates? Usually this question can only be positively answered by developers that trace the errors to the same set of source lines. Alternatively, if after fixing bug A, bug B disappears when the compiler is presented the error-inducing test case, then we are almost certain that A and B are duplicates. Still, small chance exists that A and B are not duplicates and the fix of A merely hides the erroneous behavior of B.

However, we are not compiler developers. Nor should we expect compiler writers to investigate whether two bugs found by Csmith are duplicates because the investigation requires great effort. The other option, waiting for a fix for one bug then retrying the compiler with the fix on the second bug, requires indefinite turnaround time.

As an engineering, though less scientific, measure, we judge whether two bugs are duplicates by comparing their symptoms. It is not 100% correct, as in some rare cases, identical symptoms are actually triggered by different bugs. At least this method gives us a speedy determination.

Compile-time bugs are easier to determine than wrong-code bugs. If compilers crash during compilation time due to assertion failures, they typically give a brief explanation about the location of the failed assertion, accompanied by a stack dump sometimes. However, this varies from compiler to compiler. Both GCC and LLVM generate such diagnostic messages on assertion failures. When the compiler crashes due to unsafe memory operations

or resource exhaustion, only a generic error message is dumped, if there is one.

The distinctness of wrong-code bugs triggered by random programs is much more difficult to determine. Because of the nature of random differential testing, when a bug is found, we only know a compiler "is wrong," but we do not know "how it is wrong." At the time we ran our experiments, it was not known how to determine if two wrong-code bugs are duplicates. Subsequent work by Chen et al. [8] offers some hope that two wrong-code bugs, triggered by two random test cases, have a chance to be distinguished after the test cases are reduced and ranked using the furthest point first (FPF) technique.

5.2.4.1 Quantitative comparison of GCC and LLVM versions

Figure 5.3 shows the results of an experiment in which we compiled and ran 1,000,000 randomly generated programs using LLVM 1.9-2.8, GCC 3.[0-4].0, and GCC 4.[0-5].0. Every program was compiled at -O0, -O1, -O2, -Os, and -O3. A test case was considered valid if every compiler terminated (successfully or otherwise) within five minutes and if every compiled random program terminated (correctly or otherwise) within five seconds. All compilers targeted x86. Running these tests took about 1.5 weeks on 20 machines in the Utah Emulab testbed [66]. Each machine had one quad-core Intel Xeon E5530 processor running at 2.4 GHz.

The top row of graphs in Figure 5.3 shows the observed rate of compile-time errors. (Note: The y-axes of these graphs are logarithmic.) It is understandable that earlier versions of the compilers have a higher rate of crashing. In fact, both LLVM 1.9 and GCC 3.0.0 crashed for almost 10% of the random test cases that Csmith generated. These graphs also indicate the number of compile-time bugs that were fixed by compiler developers in response to our bug reports. Both compilers became at least three orders of magnitude less "crashy" due to random programs over the range of versions covered in this experiment. There could be various explanations: the general improvement of the compiler code quality, the fixing of bugs reported by us, and the saturation of code coverage by Csmith.

The GCC results appear to tell a nice story: The 3.x release series increases in quality, the 4.0.0 release regresses because it represents a major change to GCC's internals, and then quality again starts to improve.

The middle row of graphs in Figure 5.3 shows the number of distinct crashes in LLVM and in GCC induced by our tests. These graphs conservatively estimate the number of distinct crashes by considering crashes accompanied by the same generic messages as one distinct error. We did not include these faults in our graphed results due to the difficulty

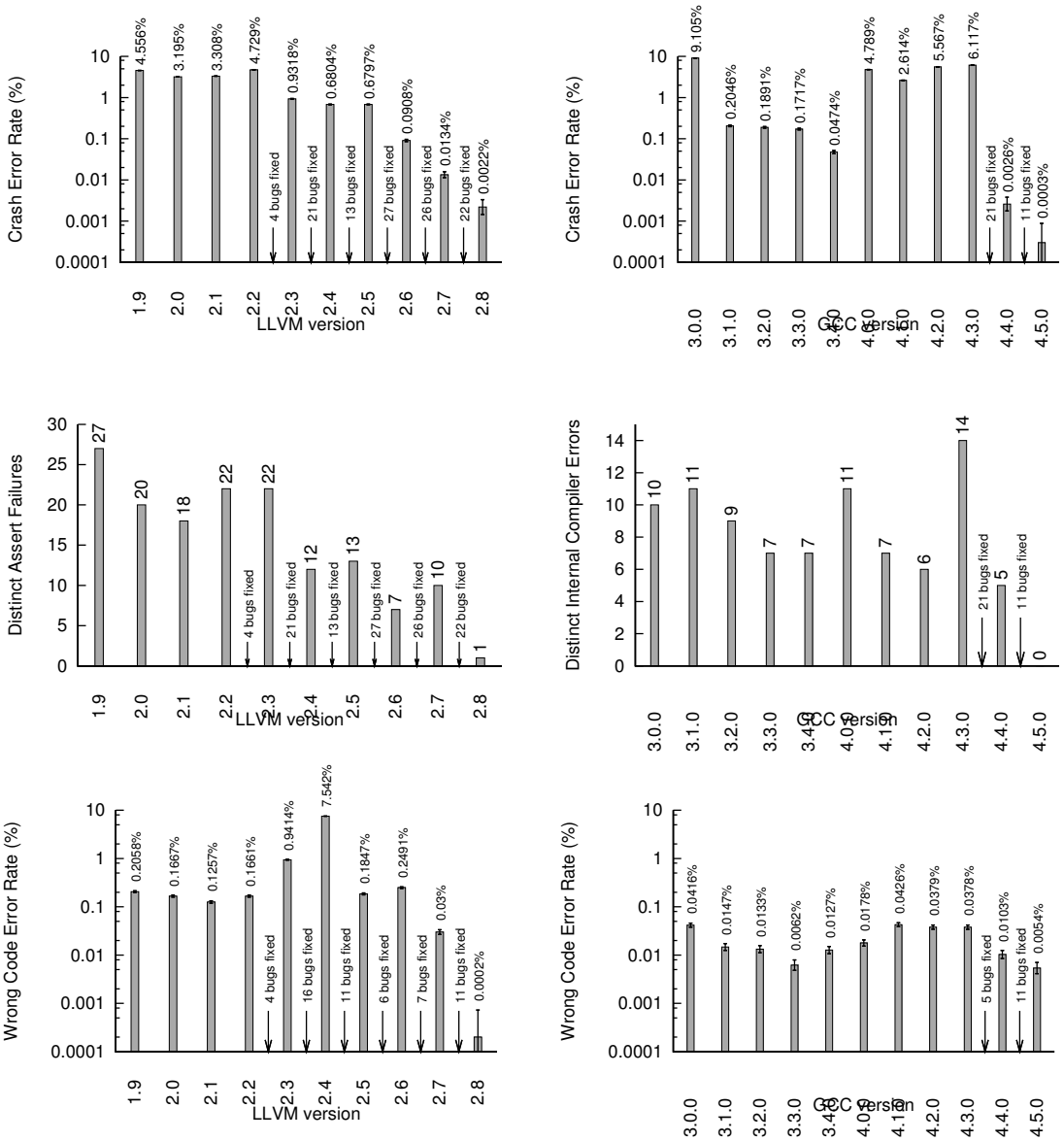

Figure 5.3: Distinct compile-time errors found and rates of compile-time and wrong-code errors, from several LLVM and GCC versions

of mapping crashes back to distinct causes.

The rate of crashes, notwithstanding distinctness, is easy to game: we can make it arbitrarily high by biasing a random generator to produce code triggering known crashes. The number of distinct crashes, on the other hand, suffers from the drawback that it depends on the quantity and style of assertions in the compiler under test. A compiler with only a few assertions would do very well on this metric.

Although GCC has slightly more total assertions than LLVM, LLVM has a much higher density because its LOC is smaller. In their latest releases, there is about one assertion per

130 lines of code in LLVM/Clang, compared to one in 360 for GCC.

The bottom pair of graphs in Figure 5.3 shows the rate of wrong-code errors in our experiment. Unfortunately, we can only report the rate of errors, and not the number of bugs causing them, because we do not know how to automatically map failing test cases back to the bugs that caused the failures. The work of Chen et al. [8], which was published after we concluded this experiment, could be used to rank the test cases; however, a precise mapping is not possible as of April 2014. These graphs also indicate the number of wrong-code bugs that were fixed in response to our bug reports.

5.2.5 Bug-finding performance as a function of test-case size

Csmith can be configured in many ways to influence the outputs: the randomly generated programs. The configurable parameters include enabling or disabling certain language constructs and the maximum allowed numbers of children for certain AST nodes.

We performed an experiment to answer this question: Given the goal of finding as many defects as quickly as possible, should one configure Csmith to generate small programs or large ones? Other factors being equal, small test cases are preferable because they most likely simplify a compiler developer's debugging effort.

Although we can tune Csmith to prefer generating larger or smaller output, it does not have a maximum-size parameter that is enforced during the random program generation. We worked around the limitation by pregenerating a set of random programs, establishing relationships between the random program sizes and the random seeds, and then used only seeds we knew that would generate random programs within a size range.

Using the same compilers and optimization options that we used for the experiments in Section 5.2.4.1, we ran our testing process multiple times. For each run, we selected a size range for test inputs, then used the precomputed seeds and Csmith to generate programs in that range.

We executed the test process for 24 hours, and counted the distinct crash errors found. We repeated this for various ranges of test-input sizes.

Figure 5.4 shows that the rate of crash-error detection varies significantly as a function of the sizes of the test programs produced by Csmith. The greatest number of distinct crash errors is found by programs containing 8 K–16 K tokens: These programs averaged 81 KB before preprocessing. The confidence intervals are at 95% and were computed based on five repetitions.

We hypothesize that larger test cases expose more compiler errors for two reasons. First, throughput is increased because compiler start-up costs are better amortized. Second, with

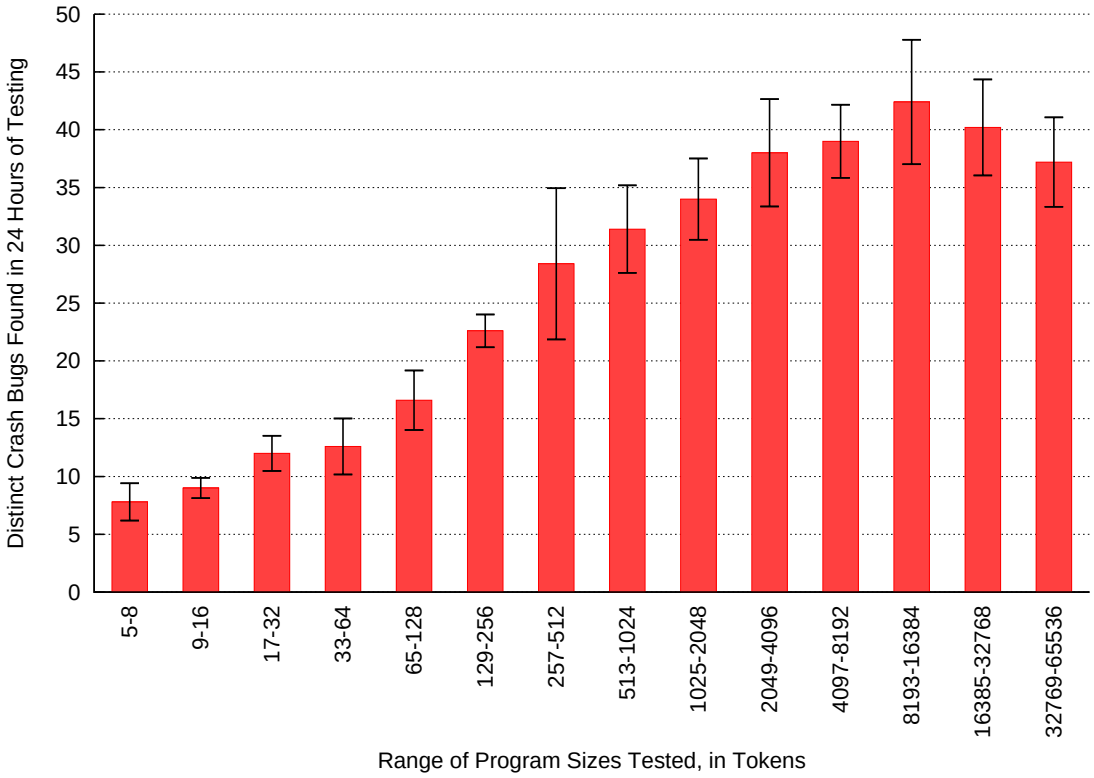

Figure 5.4: Number of distinct crash errors found in 24 hours of testing with Csmith-generated programs in a given size range

the addition of more tokens, possibly more program features as well, the combinatorial complexity caused more compiler source code to be exercised, while at the same time, more compiler optimizations became possible. For example, a single line "a = 3" could excise a limited set of source code in compilers: parsing, register allocation, memory access, etc. With the addition of a following "b = a + 7 + 1", constant folding and constant propagation becomes possible, and register allocation becomes more challenging. The decrease in bug-finding power at the largest sizes appears to come from algorithms—in Csmith and in the compilers—that have super-linear running time. That is, when generating or compiling programs exceeding a certain size, the extra time cost does not compensate for the start-up costs any more, thus the throughput drops.

5.2.6 Bug-finding performance compared to other tools

To evaluate Csmith's ability to find bugs, we compared it to four other random program generators described in Section 5.2.1: randprog (Eide08), randprog0 (Turner05), Quest (Lindig07), and DDT (McKeeman98).

Due to the limitations mentioned in Section 5.2.4, we will measure the bug-finding power by counting unique crash errors with uniqueness determined by the accompanying crash messages. We ran each generator in its default configuration on one of five identical and otherwise-idle machines, using one CPU on each host. Each generator repeatedly produced programs that we compiled and tested using the same compilers and optimization options that were used for the experiments in Section 5.2.4.1.

Figure 5.5 plots the cumulative number of distinct crash errors found by these program generators during the one-week test. Csmith significantly outperforms the other tools. It is worth noting that the bug-finding power of some random generators appears to flat out early on: DDT reached saturation during day 2; Turner05 reached saturation during day 3; and Quest reached the point during day 5. Only randprog and Csmith were able to continue finding distinct crash errors throughout the experiment period. Not surprisingly, the earlier a random program generator reaches saturation point, the less distinct compiler errors it found at the end of the seven-day period.

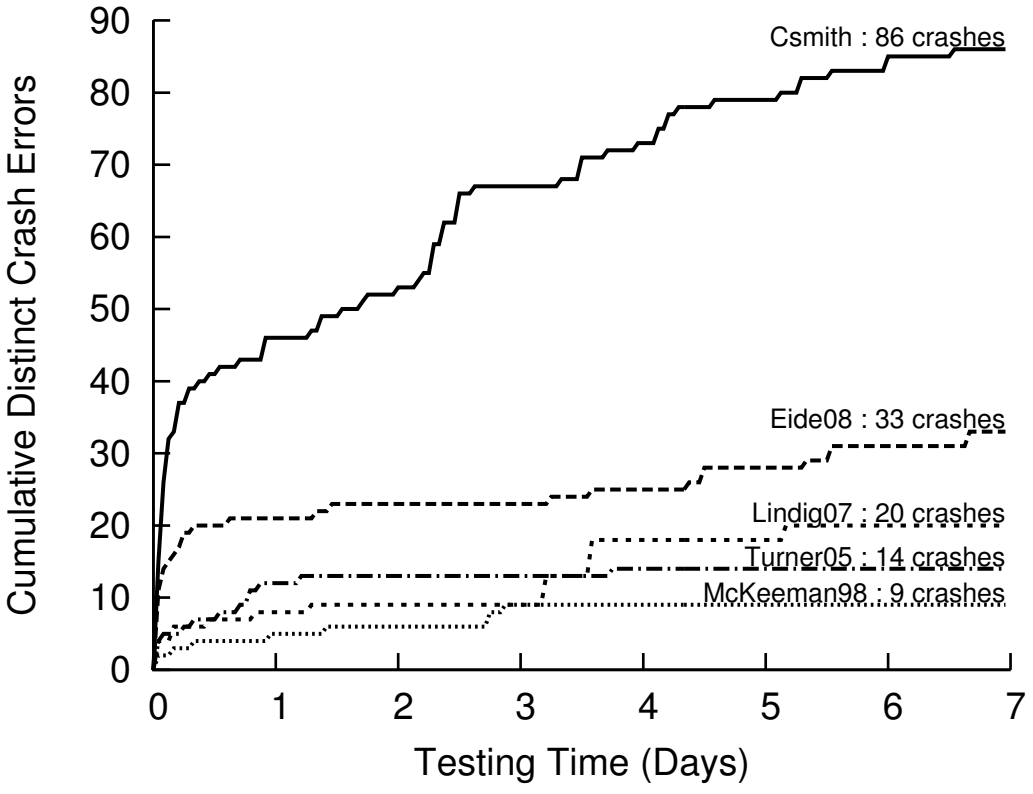

Figure 5.5: Comparison of the ability of five random program generators to find distinct crash errors

We then ran another comparison by taking the top three random generators from the above experiment: Csmith, randprog, and Quest. We also added two more variants of Csmith: cs-noloop and cs-noloopbranch.

We ran each generator in its default configuration on one of five identical and otherwise-idle machines. Each machine is a Dell PowerEdge R820 2U server with 32 cores. We ran the random generators and compilers on all 32 cores in parallel to achieve maximum throughputs. The experiment lasted five days.

To minimize the negative effects of small size programs (described in Section 5.2.5) and long-running random program generations (described in Section 5.2.2), we capped the random generation time to 20 seconds, and excluded random programs smaller than 5 KB. The remaining test cases are used to test various versions of gcc compilers from 3.3.0 to 4.5.0.

From Table 5.8, we can see that Csmith, while generating far fewer test cases than others, finds more distinct crash errors. The ranking of the bug-finding powers is in line with their corresponding code coverages shown in Figure 5.2.

Table 5.8: The bug-finding power of Csmith outperforms other random program generators and its feature-limited variants

Generator	**Total generated test cases**	**Distinct crash errors**
Quest	439375	26
Randprog	510878	76
Csmith-no-loop-no-branch	852644	73
Csmith-no-loop	657946	89
Csmith	283696	114

CHAPTER 6

COMPILER BUG STUDIES

> The Coyote is limited, as Bugs is limited, by his anatomy.
>
> Chuck Jones

Most of our testing efforts using Csmith centered on two open source compilers, GCC and LLVM. We found hundreds of bugs in each of them and reported to respective teams. It is obvious that we have no space to discuss individual bugs in full detail in this chapter, but even if we could, we would lose the big picture while we zoom in on individual bugs.

The objective of this chapter is to answer the following questions:

- What kinds of bugs did Csmith find?
- Are the bugs important?
- Why can Csmith find them while others could not?
- Where are the places in compilers that are potential hotbeds for such bugs?
- Which lessons can be learned from those bugs?

The answers to the above questions can be valuable in several ways: 1) they can be used to justify using random differential testing as an effective methodology of testing compilers; 2) they provide guidance to future designs of random program generators; and 3) they give suggestions to compiler writers on how and where to avoid compiler bugs.

Most of the questions are answered by studying the publicly available bug portals. The bug portal for GCC is located at http://gcc.gnu.org/bugzilla/. The bug portal for LLVM/Clang is located at http://llvm.org/bugs/.

We specifically studied hundreds of reports for bugs found by Csmith during our four years and three months of experiment. Some of the analysis results are published in a PLDI paper [75] and reused in this chapter. It should be noted that even though Csmith

is effective in finding compiler bugs, it still misses the vast majority of them considering the fact that bugs we reported constitute only a small percentage of the overall bugs filed against GCC and LLVM. The conclusions drawn from reading the bug reports related to Csmith do not necessarily apply to all compiler bugs.

In addition, the specific implementations of different compilers vary. In general, GCC and LLVM bear some degrees of similarity because they both compile programs in a common set of languages, namely C, C++, and Objective-C. And they adopt a common set of optimization techniques that have been standardized over the years. The conclusions drawn from analyzing bugs in GCC and LLVM may not be applicable to other compilers, which could have dramatically different implementations.

6.1 Compile-time bugs vs. wrong-code bugs

There are many angles from which we can look at the bugs found by Csmith. The most obvious one is based on manifestation of the bug: whether the error exhibits at compile time or execution time. As defined in Section 5.1.1, the bugs are called compile-time bugs and wrong-code bugs, respectively.

As shown in Table 6.1, LLVM/Clang has a higher percentage of wrong-code bugs simply because the LLVM team is generally quicker in fixing bugs that we report. As explained in Section 5.2.4, we could not determine if two wrong-code bugs are duplicate unless they are proven fixed by a single patch. Csmith was able to find many wrong-code errors in a short period of time. The strategy we used to avoid submitting duplicate wrong-bugs is to report one wrong-code error to the compiler team, wait for the fix, then try another wrong-code error discovered earlier. We only report the other error as a bug if the fix fails to make the error disappear. Compile-time bugs, on the other hand, do not have to be subjected to this "stop-and-go" process because we can generally rely on the assertion failure messages to determine if two of them are duplicates.

We have encountered several causes for compile-time errors. As shown in Table 6.2. The most frequent cause is assertion failure. This is good on one hand, meaning a crash is accompanied by a diagnostic message which simplifies the developer's debugging effort.

Table 6.1: Total compile-time and wrong-code bugs found by Csmith

	GCC	**LLVM/Clang**
Compile-time	116	191
Wrong-code	48	90
Total	164	281

Table 6.2: Compile-time bugs by category

	GCC	**LLVM/Clang**
Assertion failure	93	148
Segmentation fault	18	27
Tool chain failure	1	11
Out of resource	4	5
Total	116	191

On the other hand, the failed assertion shows there were some wrong assumptions made by the developer(s). Most likely, either because the developer failed to consider edge case(s) during the original coding, or because a late code modification caused an original assumption to be broken.

A segmentation fault is most likely caused by unsafe memory references such as null pointer dereference. It is worse than an assertion failure due to lack of specific diagnostic messages pointing to a specific function or line in the source code. There could be generic messages such as "received signal SIGSEGV," "Segmentation fault," or "stack dump."

If the outputs of a compiler are assembly programs, they are fed into another tool in the compilation tool chain, e.g., assemblers and linkers. When a compiler produces a malformed output so that a subsequent tool in the chain could not consume it, we refer the error as a "tool chain failure."

A compiler could fail because it uses up all the allocated resources. The resources could be either memory or CPU time. The most likely reason is recursive function calls that eventually lead to out-of-stack space. The second most likely reason is that an infinite loop causing the compiler to hang. In both cases, the lack of termination is caused by either 1) missing some termination conditions or 2) invalid IRs that the functions or loops were operating on.

Compared to compile-time errors, wrong-code bugs are much harder to classify based on their symptoms. We divide them into two broad categories: volatile-related bugs and non-volatile-related bugs. Table 6.3 shows the number of bugs in each category.

Volatile-related bugs are cases when a compiler fails to respect the mandates of volatiles,

Table 6.3: Wrong-code bugs by category

	GCC	**LLVM/Clang**
volatile-related	6	11
non-volatile-related	42	79
Total	48	90

that is, by flipping the read/write orders of volatile objects, or by optimizing away accesses to volatile objects. Many times, the bug was introduced when a compiler tried to apply a transformation on an operation without checking the operands' volatile qualifier. We classify a wrong-code bug as a non-volatile-related bug if it is detected by different checksums from different executions (see Section 3.2.2), or as a volatile-related bug if it is detected by different access summaries to volatile objects from different executions (the checksums are usually the same).

We use a tool based on Valgrind [50] to instrument every memory access in compiled random programs at run time. The Valgrind platform manages the instrumentation and execution of the test program that is being examined. The instrumentation records details of a memory access: address, access size (in bytes), and type (read or write). At the end of the execution, the details are compiled into a volatile access summary. We compare the access summary from executables produced by different compilers, and use differential testing to find volatile-related bugs.

The line between compile-time errors and wrong-code errors is sometimes blurred. When an error is caught by the internal checks of a compiler, it is a compile-time error. If the same error is overlooked and persisted in the compiler output, it is a wrong-code error. Nevertheless, we placed a higher priority on reporting wrong-code bugs than reporting compile-time errors because in our view, a wrong-code bug, more likely being silent, is more dangerous.

6.2 Are bugs found by Csmith important?

A possible argument against random testing has always been that the test inputs are not real and would never happen in a real user scenario. We had concerns that compiler developers could dismiss our bug report based on that argument. Fortunately, we are glad to find that our bug reports are warmly received by GCC and LLVM developers.

Still, the questions remain. Are the bugs important? Do they impair usage of compilers for real users? Are the impairments severe enough to harm the productivity of compiler users?

We argue that many bugs found by Csmith are important, and if left unfixed, could cause serious harm, not just inconvenience, to real compiler users (someone who use compilers in the process of developing softwares). We base our arguments on two facts: admissions of compiler developers and duplicate bug reports from us and other compiler users.

6.2.1 Admissions of compiler developers

Not only did compiler developers acknowledge the bugs we reported, they also spent a great amount of effort fixing them. Not every bug submitted to compiler teams received treatment like this. Bug-reporting portals of LLVM and GCC show a bug report could be rejected for various reasons: invalid, won't fix, works for me. "Invalid" means the test case is invalid, e.g, has undefined behaviors, or the way to use the compiler is not supported. "Won't fix" is for bugs that the compiler developers considered so inconsequential that they do not deserve the effort to fix them. A nonreproducible bug is dismissed as "Works for me."

As of June 2013, approximately 12% of bugs reported to the LLVM team were rejected based on the above reasons, and 14% of bugs reported to the GCC team were rejected for the same reasons. The bugs found by Csmith received better treatment. Out of hundreds of bugs found by Csmith, only 19 of the bugs reported to LLVM team were rejected, accounting for around 6% of the total LLVM bugs found by Csmith; while only eight of the bugs reported to GCC team were rejected, accounting for 5% of the total GCC bugs found by Csmith.

Most of the rejections were justified by "works for me." There are two possible reasons for this: 1) For both GCC and LLVM, we always tried to work on the latest revision from the trunks in their source code repository. However, there was a time window between when we build then test a compiler version and when a compiler developer validated a bug we submitted against that build. Due to the rapid work of compiler writers, the bug may have been fixed or made latent during the window, 2) human error is inevitable when we only have one person looking at logs produced by running thousands of test cases against a dozen compiler configurations on multiple platforms.

Another justification for rejection of bugs we reported is "invalid." During four years and three months of our testing period, only four of our early bug reports were rejected because of it. The failures are attributed to bugs in Csmith. A Csmith bug could produce undefined behaviors and make the bug-inducing test cases invalid. We have had no single bug report get rejected on this ground since 2010, implying Csmith is successful at avoiding generating invalid test cases.

Of the accepted bugs, GCC and LLVM developers have done an excellent job fixing them: Only seven (four from GCC and three from LLVM) out of 445 bugs remain unresolved as of June 2013. There are various and legitimate reasons behind the inaction. In most cases, the bugs are judged as bordering on "won't fix," but the developers have not taken the

action to label them that way.

We can consider the opinion from compiler developers with regards to the importance of bugs reported by us. After all, they have the best knowledge of the compilers, and they gain insight on the bugs while working on them. Even though there is an "importance" field on their bug reporting template, LLVM developers tend to use it only to differentiate between bugs and enhancement requests. GCC developers are more active in using the field to assign priority and severity to bugs. Here are the descriptions from GCC team [21]:

Severity: This field describes the impact of a bug. The choices are:

- Critical: crashes, memory leaks, and similar problems on code that is written in a common enough style to affect a significant fraction of users
- Normal: major loss of functionality
- Minor: minor loss of functionality, misspelled word, or other problem where an easy workaround exists
- Enhancement: request for enhancement

Priority: For regressions this field describes the importance and order in which a bug should be fixed. Priorities are set by the release management team only. If you think a priority is wrong, set it to P3 and add a note. The available priorities are:

- P1: most important. This generally labels a regression which the release manager feels should be addressed for the next release, including wrong-code regressions. A P1 regression blocks the release.
- P2: this generally indicates a regression users will notice on a major platform, though which is not severe enough to block a release. This includes bugs that were already present in a previous release.
- P3: the default priority for new PRs that have not been prioritized yet. Priorities below P3 are not on the radar of release management.
- P4: important regression on a platform that is not in the list of primary or secondary targets or a regression that users will not see for release builds.
- P5: less important regression on a platform that is not in the list of primary or secondary targets.

As shown in Table 6.4, out of 164 bugs reported by us and accepted by the GCC team, 31 of them are assigned the highest priority P1. Without a fix, the next release is blocked by these bugs. In addition, three of the bugs received a severity of "Critical" and above, showcasing their impact on GCC users.

6.2.2 Confirmation from other compiler users

To prove the bugs found by Csmith have impact on real compiler users, we compiled a list of bugs that were independently reported by at least two parties: one being us; and the other the real compiler user. The point is that if the bugs we found through random testing are also reported by real users, the bugs must have affected real code.

Instead of going through all bugs in the database, we only count bugs that are marked as a duplicate of one of the bugs found by Csmith. This is the most conservative estimation of the real-world impact of bugs found by Csmith. We could have missed a duplicate between us and a real-world user because the user did not file a bug report, or because a compiler developer failed to recognize the duplication. Additionally, we tend to test the most up-to-date revisions of compilers while most people tend to use one of the official releases. This means if there was a bug introduced since the last release, it is likely to be seen only by us, thus not reproducible by real users. Finally the success of Csmith makes the duplicate number artificially small. Because Csmith is able to quickly catch a bug soon after its introduction into the code base, and the compiler teams are eager to fix bugs reported by us, other users would not notice the bug unless they were using a revision obtained within the time window between reporting and fixing.

Despite all these adverse factors, we found quite a few duplicates between us and some real users. Sometimes a bug could hurt more than one real world team. We counted every instance of independent bug reports from parties other than us, and the other party was not random testing. The total number of such instances is 13 for GCC, and 15 for LLVM.

The bug that caused pain to most parties is LLVM bug 965. It has been independently

Table 6.4: GCC bugs by priority

Priority	**Bug number**
P1	31
P2	24
P3	108
P4	0
P5	1
Total	164

reported by eight parties (including us) so far. The list is still growing. The last time people complained about this bug was in January 2013. [1]

The bug was first reported in October 2006, and has remained unresolved since then. Csmith found it three years later with the test case in Listing 6.1:

Listing 6.1: Test case for LLVM bug 5644 (duplicate of bug 965)

```
static void func_1(void)
{
    unsigned char l_2 = 1;
    for (l_2 = 0; (l_2 > -1); l_2--)
    {
    }
}

int main(void)
{
    func_1();
    return 0;
}
```

We filed a bug report, [2] and received an explanation from the LLVM team:

> The problem here is that func_1 is an infinite loop and we infer that it can't return, throw, and has no side effects. Because of this, we add an unreachable after the call, and we later delete the dead call (since it has no side effects).

Because of the inserted "unreachable," the compiled program would crash as opposed to the expected behavior, which is hanging. It makes little sense for a programmer to write an infinite loop purposely in his or her application, so the real impact is minimal. Still the number of people complaining about this bug is alarming.

The bug that received the second most complaints on our list is GCC bug 48124, [3] which has been independently reported by four parties (including us). We first reported the bug in March 2011 with the test case in Listing 6.2:

Listing 6.2: Test case for GCC bug 48124

```
struct S0 {
  signed f0 : 26;
```

[1] http://llvm.org/bugs/show_bug.cgi?id=965

[2] http://llvm.org/bugs/show_bug.cgi?id=5644

[3] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=48124

```
  signed f1 : 16;
  signed f2 : 10;
  volatile signed f3 : 14;
};

static int g_1 = 1;
static struct S0 g_2 = {0,0,0,1};

int printf(const char *format, ...);

void func_1(void)
{
   g_2.f3 = 0;
   g_1 = 2;
}

int main (int argc, char* argv[])
{
   func_1();
   printf("g_1 = %d\n", g_1);
   return 0;
}
```

The erroneous behavior is that when compiling this test case with command "gcc -Os", the executable produces "g_1 = 1", which is obviously wrong. Trying to save memory space, GCC lays out global area as 12 bytes to "g_2" which is followed by 4 bytes to "g_1". In "g_1", The four fields f0–f3 get 4, 2, 2, 4 bytes, respectively. Then GCC performs a series of transformations leading to this internal pseudo code:

1. Load 8 bytes starting from address "g_2.f3" into register **R**
2. Assign 0 to the first 4 bytes
3. Assign 2 to "g_1"
4. Write content of **R** into 8 bytes starting from address "g_2.f3". Note this operation writes beyond the boundary of "g_2"!

GCC made the transformations based on a few assumptions: 1) This is a x64 platform so loading/storing 8 bytes is faster than 4 bytes; 2) "g_1" and "g_2.f3" are not aliases, and only one of them is volatile storage; thus writings to "g_1" and "g_2.f3" can be reordered. The assumptions are valid, but the root cause of the error is that the writing back to "g_2.f3" should be limited to the first 4 bytes.

It seems the problem is hidden when all fields are nonvolatiles. This is partly the reason why nobody had reported the problem before. We reported the bug in March 2011, and Arthur O'Dwyer, who was also using Csmith to test GCC, reported the same bug in May 2011. There were attempts to fix the bug, but none of them was committed.

In February 2012, it was reported that the same bug caused miscompilation of the Btrfs file system in the Linux kernel. Writing to a bit-field actually overwrites the content of a neighboring volatile field, which is a lock for synchronization. The bug report from the Linux team led to a heated discussion between the kernel developers and the compiler developers, and eventually involved Linus Torvalds. [4] The bug was fixed the following month. However, a bug report from another party forced a follow-up fix in June 2012.

The fix is inserting a phase in which structure fields are laid out according to the C++11 memory model, and the layout info are stored in the tree. Previously, GCC was going through a complex formula to find the starting location and ending location of a bit-field on demand.

6.3 Where are bugs?

Beside the above criteria which classify bugs based on their exhibited behaviors, we can use a more white-box classification, i.e., labeling them based on where in the compiler source code they occur. This is yet another angle from which to look at the bugs and gain deeper understanding of them.

Generally compilers are divided into front end, middle end, and back end. We rely on two sources of information to put a bug in one of the buckets: 1) the "component" field of the bug report. This is usually filled up by a developer who owns the bug; or 2) the files and descriptions of a committed fix to the bug.

The bug system of GCC divides the source code into the 40 or so components. Csmith finds bugs in the following major components:

- C: A problem with the C compiler front end
- Middle-end: GCC's middle end; folks, expand_*, etc.
- RTL-optimization: A problem occurring in the RTL optimizers
- Target: Target specific issues
- Tree-optimization: A problem occurring in the tree-ssa optimizers

[4] http://gcc.gnu.org/ml/gcc/2012-02/msg00005.html

We consider any component operating at C level to be in the front end; any target-independent operations are performed in the middle-end components, including component *rtl-optimization* and *middle-end*, and the back end is responsible for target-dependent operations, which include component *rtl-optimization* and *target*.

The LLVM team seldom uses the component field in their bug reports. However, LLVM has a cleaner code organization in which code for front end, middle end, and back end are nicely separated into different folders. The bug system shows the list of files that get committed as a bug fix. In most cases, the names of the files are sufficient to determine in which stage the bug was located. A bug is **unclassified** either because it has not yet been fixed or the developer who fixed the bug did not indicate what files were changed.

As shown in Table 6.5, the distribution of bugs across compiler stages clearly demonstrates Csmith's capability of penetrating the front end and finding bugs in the middle end and the back end. There are several possible explanations for the distribution:

- Since Csmith never generates syntactically incorrect programs, the syntax error-handling logics in the front ends are never exercised.
- Since Csmith generates only C programs, the parsing and translating logics for other languages in the front ends are never exercised.
- Since we run Csmith on x86 and x64 platforms primarily, the target code generation and target-specific optimizations in the back end are not fully exercised.
- The front end is partly auto-generated from grammar descriptions, while the back end is partly auto-generated from target descriptions. The automation reduced human errors.
- An error in the front end is more visible than an error in the later stages; therefore, it is more likely to be reported and fixed already.

Table 6.5: Distribution of bugs across compiler stages

	GCC	**LLVM**
Front end	9	12
Middle end	109	122
Back end	46	98
Unclassified	0	49
Total	164	281

- The middle end performs most of the optimizations involving complex analyses and transformations, which are error-prone.

Out of millions of lines of source code, where are the compiler bugs exactly? We studied the files that were checked in for fixing bugs that we reported. Tables 6.6 and 6.7 show the 10 buggiest files in LLVM and GCC as measured by our experiment. They confirm the distribution in Table 6.5.

Both GCC and LLVM show that the bug-rich files are in the middle end and the back end. Most of them are responsible for optimizations. The middle end consists of language-independent and target-independent transformations, which means even though we are only generating C programs with Csmith, and only testing compilers on a few platforms, the bugs we found have impact on programs written in other languages and/or compiled for other platforms, as long as GCC or LLVM are used as the compiler.

The bugs are distributed in a wide range of files. The GCC files that did not make it into the top 10 list but still contained many bugs are: cfgexpand (3), cfgloopmanip (3), expr (3), tree-ssa-sccvn (3), tree-scalar-evolution (2). Likewise, the LLVM files after the top 10 list are: LoopUnswitch (5), X86ISelDAGToDAG (5), SimplifyCFG (5), BasicAliasAnalysis (4), SelectionDAG (4). Numbers in parenthesis are the distinct bug count in the preceding file.

When we created Csmith, we tried hard to make it generate auto-vectorizable loops as shown in Listing 6.3. The effort has paid off well. We found 12 bugs in auto-vectorization of GCC. Auto-vectorization code in GCC are disbursed in several files, so none of them made it into the top 10 list. An example of a GCC vectorization bug found by Csmith is here: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=49926.

Listing 6.3: Example of a vectorizable loop

```
int a[256], b[256], c[256];
foo () {
  int i;

  for (i=0; i<256; i++){
    a[i] = b[i] + c[i];
  }
}
```

An interesting observation is that there are overlaps in terms of functionality between the two file lists. Both GCC and LLVM were often wrong on folding arithmetic operations involving constants, instruction combing, and target (x86) description. We think these areas are full of case-by-case transformations and thus could have tricked compiler writers.

Table 6.6: Top 10 buggy files in GCC

C File Name	Purpose	Wrong-Code Bugs	Compile-Time Bugs	Total Bugs
fold-const	constant folding	2	10	12
combine	instruction combining	3	7	10
tree-vect-loop	loop vectorization	0	7	7
tree-ssa-pre	partial redundancy elim.	0	6	6
tree-vrp	variable range propagation	1	7	8
tree-ssa-ccp	cond. constant propagation	1	3	4
ipa-split	function split for inlining	1	3	4
i386.c	i386 OS-independent desc.	0	4	4
tree-ssa-dce	dead code elimination	0	3	3
tree-ssa-reassoc	arithmetic expr. reassociation	0	3	3
Other (67 files)	n/a	40	63	103
Total (77 files)	n/a	48	116	164

Table 6.7: Top 10 buggy files in LLVM

C++ File Name	Purpose	Wrong-Code Bugs	Compile-Time Bugs	Total Bugs
Instruction-Combining	midlevel instruction combining	15	6	21
DAGCombiner	instruction combining	6	6	12
SimpleRegister-Coalescing	register coalescing	1	10	11
ScalarEvolution	induction variable analysis	2	8	10
ValueTracking	computation chain analysis	2	6	8
LoopStrength-Reduce	loop strength reduction	3	4	7
X86InstrInfo	x86 target description	3	4	7
LiveInterval-Analysis	liveness analysis	0	6	6
JumpThreading	jump threading	0	6	6
TwoAddressInstr-uctionPass	two addr. instr. conversion	0	6	6
Other (83 files)	n/a	58	129	187
Total (93 files)	n/a	90	191	281

6.4 How did compilers get it wrong?

With the hundreds of bugs that we found, it is obvious that compilers are buggy. The next questions naturally are: 1) Why are they buggy? (i.e., what are the root causes of the bugs?) and 2) How do compiler writers avoid the same mistakes in the future?

We have studied a handful of bugs that we reported. We read the bug reports, and

examined the source code modifications in fixes. We tried our best to understand the root causes, and based on that, we derived a few common patterns of these errors. The following pattern list does not intend to be exhaustive or 100% accurate. It merely reflects our best effort to understand of the bugs we found with Csmith.

Most compiler optimizations can be simplified to the structure in Listing 6.4:

Listing 6.4: Compiler optimization flow

```
analysis
if (safety check) {
  transformation
}
```

An optimization can fail to be semantics-preserving if the analysis is wrong, if the safety check is insufficiently conservative, or if the transformation is unsound. In most cases, the transformation has been theoretically proven correct or validated by test cases. However, the pretransformation analysis and safety checks are more error prone. They are more likely to miss on comprehensiveness due to infinity of program space.

An analysis could be wrong for at least three reasons: 1) the analysis made wrong assumptions, or 2) the analysis failed to consider some uncommon cases, or 3) the analysis result became stale due to transformations that happened after the analysis. A few examples of the last reason are as follows:

- LLVM bug 2372:[5] In loop unswitching pass, a version of LLVM forgot to update the immediate dominator after inserting nodes.
- LLVM bug 2455:[6] Transformation (loop unswitching) invalidated previous analysis results (Dominance Frontier).
- LLVM bug 12901:[7] A version of LLVM forgot to propagate the fact that a value is deleted to all the users of the value.

In both GCC and LLVM, the analysis results are stored in the IR trees, and there are periodic checks on the integrity of the trees. The internal checks are likely to throw an exception or generate an assertion failure if 1) a analysis result becomes stale and damages

[5] http://llvm.org/bugs/show_bug.cgi?id=2372

[6] http://llvm.org/bugs/show_bug.cgi?id=2455

[7] http://llvm.org/bugs/show_bug.cgi?id=12901

the integrity of the tree; or 2) an unsafe transformation produces a nonconforming tree. In this case, the bug manifests as a compile-time error.

We have found cases where the analysis is wrong. The analysis could be as simple as overflow checking, or as complex as pointer alias analysis. The most common root cause for bugs that we found is an incorrect safety check, i.e., incorrect gates. In other words, the gate caused certain code to be wrongly transformed while they should 1) not be transformed at all or 2) be transformed differently.

A complex safety check is often expressed as $C_1 \wedge C_2 \wedge ... \wedge C_n$. But sometimes compiler developers fail to consider an additional condition C_{n+1} for some uncommon cases.

Listing 6.5 is an extreme example showing the difficulty of considering every condition when writing a safety check. Reload is a pass in GCC that performs post-register-allocation fix-ups such as spilling and coalescing. As the proclaimed "GCC equivalent of Satan," it has a safety check composed of 10 and-joined clauses. Including the subconditions inside the clauses, there are a total number of 14 subconditions in the safety check. Given the complexity, it is not surprising that some subconditions are overlooked by developers.

Listing 6.5: Example showing a complex safety check in GCC

```
if (REG_NOTE_KIND (note) == REG_DEAD
    && REG_P (XEXP (note, 0))
    && !reg_overlap_mentioned_for_reload_p (XEXP (note, 0), rld[
        output_reload].out)
    && (regno = REGNO (XEXP (note, 0))) < FIRST_PSEUDO_REGISTER
    && HARD_REGNO_MODE_OK (regno, rld[output_reload].outmode)
    && TEST_HARD_REG_BIT (reg_class_contents[(int) rld[output_reload].
        rclass], regno)
    && (hard_regno_nregs[regno][rld[output_reload].outmode]
        <= hard_regno_nregs[regno][GET_MODE (XEXP (note, 0))])
    /* Ensure that a secondary or tertiary reload for this output won't
       want this register. */
    && ((secondary_out = rld[output_reload].secondary_out_reload) == -1
        || (!(TEST_HARD_REG_BIT
              (reg_class_contents[(int) rld[secondary_out].rclass],
                  regno))
            && ((secondary_out = rld[secondary_out].
                secondary_out_reload) == -1
                || !(TEST_HARD_REG_BIT
                     (reg_class_contents[(int) rld[secondary_out].
                         rclass], regno)))))
    && !fixed_regs[regno]
    /* Check that a former pseudo is valid; see find_dummy_reload.*/
    && (ORIGINAL_REGNO (XEXP (note, 0)) < FIRST_PSEUDO_REGISTER
        || (!bitmap_bit_p (DF_LR_OUT (ENTRY_BLOCK_PTR),
                           ORIGINAL_REGNO (XEXP (note, 0)))
            && hard_regno_nregs[regno][GET_MODE (XEXP (note, 0))] == 1)
            ))
  {
    ...
```

6.5 Case studies of bugs

6.5.1 GCC Bug #1: wrong analysis

Does `(0/(unsigned long long)-1) != 1` evaluate to `FALSE`? Apparently at one point, GCC did.[8] The error was introduced while gcc tried to fold an expression `(x/c1)!=c2` where `c1` and `c2` are constants and `x` is variable. Generally, it can be rewritten as `x < c1*c2 || x > (c1*c2)+(c1-1)`. This transformation is profitable because `c1*c2` and `(c1*c2)+(c1-1)` can be computed at compile time. Turning an expensive division into two comparisons with constants saves a lot of CPU cycles. Such an optimization can be applied to the test case in Listing 6.6.

Listing 6.6: Error inducing test case for GCC bug 42471

```
static unsigned long long
foo (unsigned long long x, unsigned long long y)
{
  return x / y;
}

static int a, b;

int
main (void)
{
  unsigned long long c = 1;
  b ^= c && (foo (a, -1ULL) != 1L);
  if (b != 1)
    __builtin_abort ();
  return 0;
}
```

The above transformation is valid only when there is no integer overflow when computing `c1*c2` and `(c1*c2)+(c1-1)`. However, if overflow does occur, a further simplification can be made irrespective of the value of `x`; for example, `(x/1000000000)!=10` always evaluates to 1 when `x` is a 32-bit integer.

In the example of `(0/(unsigned long long)-1) != 1`, `x` is 0, `c1` is -1 (unsigned), and `c2` is 1. `(c1*c2)+(c1-1)` does overflow. However, a bug in GCC determined it does not; the wrong determination caused the whole expression to be folded into `FALSE`.

GCC determines whether there is an integer overflow while adding two 64-bit integers in function `add_double_with_sign` (double-int.c). It performs a 32-bit addition on the lower half of the 64-bit integers first. If the result is smaller than one 32-bit operand, then a carry flag is set. The function then performs the second 32-bit addition on the upper half of the 64-bit integers, plus 1 if the carry flag is set. GCC developers thought that no

[8] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42721`

overflow during an addition if the sum is greater than or equal to the second operand. The assumption is apparently wrong when the first operand is unsigned -1 and the carry flag is set.

In this bug, the gate to transformation is correct, but a wrong analysis result (overflow analysis) led to a transformation that should have not happened.

In order to trigger this bug, we have to create an expression `(x/c1)!=c2` where `c1` is an unsigned 64-bit integer with the upper half being `0xFFFFFFFF`, and `c2-1` should be large enough so that there is overflow while adding the lower half of it with the lower half of `c1`. By mixing integer types with various widths, either signed or unsigned, and by generating random constant values near 0 or maximum representable values, and by producing a chain of random arithmetic operations, it is not surprising that Csmith could find this bug.

6.5.2 GCC Bug #2: wrong transformation

In C, if an argument of type `unsigned char` is passed to a function with a parameter of type `int`, the values seen inside the function should be zero-extended; thus, the value is positive. For the test case in Listing 6.7, a version of GCC inlined this kind of function call and then sign-extended the argument rather than zero-extending it while trying to perform constant propagation. `p_13` was set to -1 as opposed to the correct value 255.[9]

Listing 6.7: Error inducing test case for GCC bug 43438

```
extern int printf (__const char *__restrict __format, ...);

static unsigned char g_2 = 1;
static int g_9;
static int *l_8 = &g_9;

static void func_12(int p_13)
{
  int * l_17 = &g_9;
  *l_17 &= 0 < p_13;
}

int main(void)
{
  unsigned char l_11 = 254;
  *l_8 |= g_2;
  l_11 |= *l_8;
  func_12(l_11);
  printf("%d\n", g_9);
  return 0;
}
```

[9] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43438`

In order to trigger this bug, we have to trigger both inling (with small function bodies), constant folding, and constant propagation. A key requirement is that the inlined function must have a formal parameter of a signed integer type while its actual parameter must be of an unsigned narrower integer type. Csmith was able to generate functions with random length, mix constants and variables in expressions, and allow flexible type matching between a formal parameter and an actual parameter of a function. All above contributed to the discovery of this bug.

6.5.3 GCC Bug #3: wrong analysis

The test case in Listing 6.8 returned 1 instead of 0 if compiled with a version of GCC.[10] The problem occurred when the compiler failed to recognize that p and q are aliases; The wrong not-alias fact caused the store "*p = 0" in line 7 to be masked by the next-line store "*p = *q", triggering a DSE (dead-store elimination) pass following the alias analysis to remove the first store statement.

Listing 6.8: Error inducing test case for GCC bug 42952

```
static int g[1];
static int *p = &g[0];
static int *q = &g[0];

int foo (void) {
  g[0] = 1;
  *p = 0;
  *p = *q;
  return g[0];
}
```

The wrong not-alias fact was based on the wrong assumption that q points to a read-only memory location because $*q$ is not in LHS of any assignments, and pointers to read-only locations should never be the aliases of pointers to mutable locations. The GCC team fixed the bug by disregarding the read-only attributes of pointed locations while performing dead store elimination.

In order to trigger this bug, we would have to create aliased pointers, and write to the same location through different pointers in the same alias set. Adding support to pointers and pointer dereferences (in both LHS and RHS) in Csmith enabled us to find this deeply hidden bug.

[10] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42952

6.5.4 GCC Bug #4: inconsistent state

A version of GCC[11] miscompiled the function in Listing 6.9:

Listing 6.9: Example showing a complex safety check in GCC

```
int x = 4;
int y;

void foo (void) {
  for (y = 1; y < 8; y += 7) {
    int *p = &y;
    *p = x;
  }
}
```

When function `foo` returns, `y` should be 11, while instead, a bug caused the ending value of `y` to be 8. A loop-optimization pass, loop invariant code motion (LICM), determined that a temporary variable representing `*p` was invariant with value `x+7` and hoisted it in front of the loop, while retaining a transient dataflow fact indicating that `x+7 == y+7`, a relationship that no longer held after code motion. `y+7` is 8 before the loop, and becomes 11 after the loop. This inconsistent state leads GCC to directly set the value of the induction variable `y` to 8.

In order to trigger this bug, we have to create loops with the induction variable being modified in the loop body through a pointer, a scenario that is unlikely to happen in real programs. Csmith generates flexible loop headers and loop bodies where the induction variable is not precluded from choices of read/write variables in the loop. This partly explains why Csmith was able to find bugs that remained hidden to other testing efforts.

6.5.5 LLVM Bug #1: wrong analysis

`(x==c1)||(x<c2)` can be simplified to `x < c2` when `c1` and `c2` are constants and `c1<c2`. An LLVM version[12] incorrectly transformed `(x==0)||(x<-3)` to `x < -3`. LLVM does a comparison between 0 and -3 in the safety check for this optimization, but a version of it only looked at the left operand of the comparison to determine if it should perform a signed comparison or an unsigned one. The transformation was determined safe because unsigned 0 is less than unsigned -3.

To trigger this bug, we have to generate chains of arithmetic operations with a mixture of signed or unsigned integers, either in the form of variables or constants. These are also required for triggering GCC bug 42721.

---

[11] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43360`

[12] `http://llvm.org/bugs/show_bug.cgi?id=2844`

6.5.6 LLVM Bug #2: wrong safety check

`(x|c1)==c2` evaluates to 0 if `c1` and `c2` are constants and `(c1&~c2)!=0`. In other words, if any bit location is set in `c1` and is not set in `c2`, the original expression cannot be true. LLVM uses `!IsNullValue(c1 & ~c2)` to determine if the transformation can be made. A version of LLVM[13] contained a logic error in the safety check which failed to verify whether `c1` and `c2` are actual constant integers. In the error inducing test case, `c1` is the sum of two constant integers, but not a constant integer itself. Function `IsNullValue` returns a boolean type, and is implemented as returning `FALSE` when the expression could not definitively evaluate to zero. The bug led to the expression being folded into 0 even when `c1` was not a constant integer.

The requirement to trigger this bug is similar to what is required to trigger the LLVM bug 2844, i.e., chained integer arithmetics involving constants.

6.5.7 LLVM Bug #3: wrong safety check

"Narrowing" is a strength-reduction optimization that can be applied to loads when only part of an object is needed, or to stores where only part of an object is modified. For example, at the level of the abstract machine, the code in Listing 6.10 loads and stores an unsigned int:

Listing 6.10: Example showing a complex safety check in GCC

```
unsigned y;

void bar (void) {
  y |= 255;
}
```

Optimizing compilers for x86 may translate function **bar** into the code in Listing 6.11, which loads nothing and stores a single byte:

Listing 6.11: Example showing a complex safety check in GCC

```
bar:
  movb  $-1, y
  ret
```

We found a case in which LLVM[14] attempted to perform an analogous narrowing operation, but the safety check failed to consider a store to the object-to-be-narrowed prior to the narrowed store. If there is such an in-between store, the narrowing fails to overwrite the bits that are not included in the narrowed store, violating semantic preserving.

---

[13] `http://llvm.org/bugs/show_bug.cgi?id=7750`

[14] `http://llvm.org/bugs/show_bug.cgi?id=7833`

In order to trigger this bug, we need to generate integer load-modify-store sequences with modification that only happens to partial bits of the integer. In addition, another store should be generated between the load and the store in the above sequences.

6.5.8 LLVM Bug #4: wrong analysis

The code in Listing 6.12 should print "5":

Listing 6.12: Error inducing test case for LLVM bug 7845

```
1 void foo (void) {
2   int x;
3   for (x = 0; x < 5; x++) {
4       if (x) continue;
5       if (x) break;
6   }
7   printf("%d", x);
8 }
```

LLVM's scalar evolution analysis computes properties of loop induction variables, including the maximum number of iterations. Line 5 of the program above caused this analysis in a version of LLVM to mistakenly conclude that `x` was 1 after the loop executed.[15] It failed to consider that line 6 is skipped after the first iteration of the loop.

6.6 What lessons can be learned from bugs reported by us?

We think everybody who has a stake in compilers, including compiler developers, compiler testers, and compiler users, could potentially learn something from our years of compiler testing process and the bug reports produced out of the process. If lessons are learned and applied, we are more likely to see better compilers with improved correctness.

6.6.1 Compiler developers

Adding more internal checks can turn wrong-code bugs into compile-time bugs. If the internal checks have proper assertions, the compile-bug errors should be in the form of assertion failures. A compile-time error not only alerts the users of a compiler error, but also gives valuable diagnostic messages to developers who try to fix the bug. The internal checks could be one of the following: 1) validating the integrity of the IR, 2) validating certain invariant properties, or 3) validating the semantic preservation of the source programs. Granted, adding more checks will adversely impact the compiler's performance. But we

[15] `http://llvm.org/bugs/show_bug.cgi?id=7845`

think the balance between correctness and performance should be carefully weighted by compiler teams, and correctness should generally trump performance.

Both GCC and LLVM have comprehensive test suites that achieve over 70% of the lines of their respective source code. For compiler bugs, executing a line does not necessarily means its correctness is proved. The execution should be followed by some validations. Compiler teams should at least consider using a "checked" version of a compiler (in which more internal checks are turned on) to validate against their test suites.

The safety checks for a transformation should be simple and humanly conceivable. If a safety check is convoluted, in most cases, it can be rewritten to make it more understandable. Sometimes it requires a redesign of the transformation. The clear benefits are that the code becomes much more maintainable, and a potential incorrect safety check is much more identifiable.

Interestingly, both GCC and LLVM were often wrong around the same areas, e.g, constant folding, register allocation, loop optimization, instruction combining, and target description. We hope both teams and other C compiler teams pay special attention to these areas. These areas deserve more validation or verification efforts in the forms of code inspection, test case coverage, model-based testing, or theory proving.

6.6.2 Compiler testers

We think random testing should be an integral component of compiler testing. Developing "good" random program generators may take no less effort than writing a compiler, but the effort is well-justified. Once it is written, the random program generator can be used to test multiple compilers as long as they implement the same language standard(s). We have demonstrated the effectiveness of random differential testing in this thesis. The ability to find bugs in a short period of time makes random testing a good addition to a build verification test (BVT) suite.

6.6.3 Compiler users

From our experience, bug reports are most beneficial to and welcomed by compiler writers when some of the following steps are taken and the results are included:

- Reduce the error inducing test case to as small as possible. GCC team provides a link [18] on how to do that. C-Reduce [54] is a great tool to achieve the goal. Bugpoint [39] is another excellent tool to reduce error-inducing test cases written in LLVM IR.

- Find a set of compiler options that would trigger or hide the bug. If the error happens when an optimization flag, such as "-fcrossjumping," is specified, and disappears when the flag is absent, mention the fact in the bug report.

- Find the first revision that shows the regression for a bug. The bug is most likely hidden in the change list of the revision. Both git [16] and svn [17] provide commands to automatically find the first revision showing a regression.

- Limit the symptom of a wrong-code bug to a small set of values, and specify the expected correct values.

- Review your code to make sure there are no undefined behaviors. Be prepared for arguments from compiler writers such as "your code is undefined." Even better, be preemptive by explaining code that might appear undefined while actually it is not, along with a citation from the language standard. Signed integer overflow and evaluation orders are two major sources of surprises that the "correct" value may not be the expected one. We have found that KCC [15] and Frama-C [11] are valuable tools to ensure bug reporters are in a defensible position.

[16] `http://git-scm.com/`

[17] `http://subversion.tigris.org/`

CHAPTER 7

RELATED WORK AND FUTURE DIRECTIONS

I have seen the future and it works.

Lincoln Steffens

The research paper [75] about Csmith was published in 2011. Before that, we had received periodic requests on how to adapt Csmith for various purposes. Csmith was open sourced in April 2011 [74]. Since then, it has been used in at least eight research projects that resulted in published papers.

Besides academic researchers, Csmith has found users in industrial environments. There are efforts to port GCC to new target platforms, and Csmith was used as a quick way to validate the compilers, especially the back ends.

The usage of Csmith ranges from casual to serious. A casual user of Csmith is defined as someone who uses it as it is, and often times, the randomly generated C programs are used to validate an implementation that is not related to compilers. A serious user of Csmith, on the other hand, changes or configures the generator in various ways, hoping to gain better results. A serious user is more likely to use Csmith in the course of compiler testing.

The serious users of Csmith have illustrated well how the tool could be extended. There are additional directions we could go from here. I will present my thoughts on the future of Csmith in Section 7.4.

7.1 Research that uses Csmith

7.1.1 Validation of FPGA-based emulation for postsilicon validation

Traditional ASIC (Application-Specific Integrated Circuit) designers use software simulation when verifying their designs. Unfortunately, software simulation runs approximately eight orders of magnitude slower than actual silicon. The slowness limits the functionalities

of the circuits that can be validated by software simulation. FPGA-based emulation technology executes the design at orders of magnitude faster than simulation, and thus achieves much higher validation coverage. The emulation speeds are fast enough to perform common validation tasks, albeit slower than the ASIC chip [3].

In their research, Balston et al. [3] argue that emulation can be used as an important tool to assist validation of more than just functional behavior. In particular, they focus on timing validation and the task of quantifying the critical-path coverage of a validation plan.

The prove their hypothesis, the researchers constructed various benchmarks, applied them to the emulator, and measured path coverages for different function units including a DDR2 controller, a 7-stage Pipelined Integer Unit, a Memory Management Unit, and a 32-bit integer Multiplier. One of the benchmarks, `Random`, is composed of 10,000 random programs generated by Csmith. `Random` is run side-by-side with five other benchmarks, including `Linux`, a task that boots up an embedded Linux operating system.

Balston et al. observed that:

> The dominant overall trend is that more cycles translates into better coverage, hence emphasizing the value of long-running postsilicon tests. For example, the Linux boot tends to achieve the highest coverage, but the long-running random benchmark gradually catches up. We also see that coverage often comes in bursts, when an application starts a new phase of computation that performs different operations. There are also some peculiarities, e.g., the random benchmark does particularly well on the DDR controller. We believe this is because the benchmark is loading in a series of programs one-after-another from memory, so it exercises the DRAM extensively very early on. [3]

`Random` ranked second only to `Linux boot` in overall coverage of the most critical paths in all four function units. Random won over several strong competitors, including a synthetic computing benchmark (Dhrystone), a commercial system level benchmark (systest), and a benchmark from Stanford.

An obvious question a validation engineer would ask is whether any test is dominated by the others. The researchers presented data showing that `Random` covered 25 paths that were missed by all other benchmarks. This is second only to `Linux boot`, which covered 52 paths exclusively. The distant third is `systest` with only three.

7.1.2 Validation of executable semantics

CompCert [34], the "verified" compiler, tries to demonstrate its correctness by proving its preservation of semantics, formally defined as any observable behavior of an assembly output, if ever generated, must match one of the acceptable behaviors of the C source

program. Most of the CompCert is formalized in the Coq proof assistant and accompanied by correctness results showing the preservation of semantics along the compilation steps.

Formal semantic definitions must be given to both the source language (C) and the target language (assembly) before correct properties can be reasoned. In CompCert, these are given by inductively defined small-step relations. Needless to say, errors and omissions in these semantics could cause bugs in the compiler. Thus, the semantics must be validated.

In his research, Campbell [6] explored another approach to validate CompCert's semantics: building an equivalent executable semantics and applying test suites to both executable semantics. If the test results do not agree, then it must be an error in either Campbell's semantics or CompCert's semantics. This is fundamentally another form of differential testing.

Campbell's executable semantics is translated into an interpreter, and random programs are generated by Csmith then applied to both the interpreter and CompCert. Campbell noted that:

> Csmith is particularly interesting because it has already been used to test the CompCert compiler as a whole where it detected several bugs in the informal part of the compiler. ... Given the previous testing of the compiler, it was unsurprising that we found no problems when executing the randomly generated code with the interpreter and comparing the results against the compiler. However, we did encounter a failure with the nonrandom support code.... [6]

The "nonrandom support code" was actually one of the safe math wrappers. Based on the failure, Campbell was able to construct a failure-inducing test case that caused CompCert to fail at compile time.

7.1.3 Validation of machine code analysis

Machine code sometimes are analyzed for security reasons. This is essential for execution of machine code in a restrictive sandbox environment, such as Google's Chrome browser. Morrisett et al. constructed a x86 code analyzer [47] that could be validated with formal methods. They first constructed a pair of domain-specific languages (DSLs) for specifying the semantics of machine architectures, including x86, and embedded those languages within Coq. Then they used the DSLs to generate OCaml code that can be run as a simulator. The two DSLs, one specifying the translation from bits to abstract syntax, and the other specifying small-step operational semantics at RTL level, are architecture-independent and could thus be reused to specify the semantics of other machine architectures.

Morrisett et al. used two different techniques to generate test cases to exercise the simulator. First, they generated small, random C programs using Csmith and compiled

them using GCC. In this way, they simulated and verified over 10 million instruction instances. However, they mentioned this technique has limitations. It did not exercise instructions that are avoided by compilers/assemblers. They suggested "a more thorough technique is to fuzz test our simulator by generating random sequences of bytes, which has previously proved effective in debugging CPU emulators" [47].

7.1.4 Validation of static analysis

Frama-C [9] is a framework for analysis and transformation of C programs. The framework provides a set of built-in analyses and transformations, such as 1) a value analysis that computes an over-approximation of the values each variable can take at each point of the program, 2) a constant propagation plug-in; and 3) a slicing plug-in that removes redundant code based on certain criteria. In addition, the framework allows third-party analyses and transformations to be plugged in.

Cuoq et al. [10] argue that static analysis should not overlook the impact of compilers on the accuracy of any analysis result. They further argue that subjecting a compiler and an analyzer to automated random testing could minimize the adverse impact. They put their argument into practice by random testing Frama-C using Csmith, the tool that has been used to random test well-known compilers.

Based on its value analysis, Cuoq et al. turned Frama-C into an interpreter of C programs. Using random programs generated by Csmith, it is then validated against a reference compiler by comparing the interpreter outputs with the execution outputs of the compiler. In all, they found 50 bugs in Frama-C with random differential testing. The bugs are located in value analysis, the constant propagation plug-in, and the slicing plug-in.

An interesting observation is that they filtered out random programs based on various file sizes. At the smallest setting, 20 KB, the random programs were able to find all Frama-C bugs. They then increased the filter so larger files could be generated, but no additional bugs were revealed.

More interestingly, several bugs were also found by Frama-C in Csmith. The bugs involved passing around dangling pointers, unsequenced assignments to the same memory location, or access uninitialized members of unions. Lastly, Csmith could generate a pre-increment causing undefined signed overflow.

The bugs in Csmith were found during a period when we added support for unions and increment/decrement operators to Csmith. There was a positive feedback loop between Frama-C and Csmith: as soon as we introduced the constructs into random programs, they revealed quite a few bugs in Frama-C's handling of such constructs, while Frama-C team

found a few bugs in the generation logic in Csmith that avoid undefined behaviors related to the constructs. The positive feedback loop leaped forward both tools' quality.

7.2 Research that extends Csmith

7.2.1 Test case reduction

Random programs generated by Csmith are typically from a few kilobytes to hundreds of kilobytes. As explained in Section 5.4, Csmith is designed to generate medium size programs to increase bug-finding performance.

Regehr et al. [54] tried to solve the problem of automatic test case reduction: i.e., given an error-inducing test case, how to reduce it to a size that is acceptable to compiler teams while preserving the error-inducing capability. The problem is difficult in that the final results depend not only on which types of transformations we apply in each step, but also on the order of the transformations. It is practically infeasible to explore all possible types of transformations in all possible orders with a large number of transformations.

The problem has haunted us since the first day we started random testing compilers. We tried manual reduction at first, which on average took one half to one hour for each test case even for an expert team member who is extremely good at reducing them. This is definitely not scalable. As more bugs piled on, test case reduction would have overwhelmed our four-member team.

At first, we tried Berkeley delta [45], an implementation of delta debugging [77]. Berkeley delta is generic and line-based. We found Berkeley delta sometimes emits reduced program with undefined behaviors, and the simple line-based reduction could not describe effective simplifications, such as removing a function parameter from both the definition and call sites.

Secondly, we tried to alter the random program generation sequence inside Csmith to reduce error-inducing test cases. Csmith makes a set of random choices while constructing the abstract syntax tree. The random choices are encoded by a sequence of random numbers. For example, if it chooses to generate an if-statement out of 10 production rules for statement, the choice is encoded as 5/10 since if-statement is the number five choice. The reduction, called Seq-Reduce, mutates the random sequence and force Csmith to generate based on the prescribed sequence. If the mutation produces a test case that triggers the compiler failures and is smaller, we repeat the process staring with the mutated sequence. Seq-Reduce is both simple and easy to parallelize. The main problem is that it does not do a good job of reducing programs in cases where the problematic code is generated near

the end of the random sequence. Mutating choices early in the sequence tends to suppress error-inducing code generated late in the sequence.

Our next reducer, Fast-Reduce, is also based on altering the random program generation inside Csmith. It explores the idea that test-case reduction can benefit from runtime information. It simplifies the test cases with several transformations including dead-code elimination, branch divergence detection and simplification, and effective inlining. The transformations are assisted by runtime information, which is obtained by instrumenting the random program and then executing it. Fast-Reduce generally is fast and effective. Its primary disadvantage is that it does not always provide good results. It follows a fixed order of transformation steps. Each type of transformation is applied only once. Adding a new type of transformation requires intimate knowledge of Csmith; thus, only a limited set is supported.

Both Seq-Reduce and Fast-Reduce rely on Csmith to generate variants of an error-inducing test case. The GTAV inside Csmith ensures that the variation is still defined. Thus, no validation on the variants is necessary.

The third reducer, C-Reduce, developed by Regehr et al., uses a mixture of transformations to reduce the test cases. Some of the transformations are C-specific. The transformations are pluggable. They are applied iteratively until a global fix-point is reached.

C-Reduce supports 50+ transformations, which can be categorized as below:

- Peephole optimizations: replacing contiguous segments of the tokens, if applicable and profitable.
- Line removing: replacing contiguous lines, if applicable and profitable.
- Pretty-printing: formatting the code.
- Compiler-like optimizations: transformations employed by compilers, such as scalar replacement of aggregates, lifting a local variable to global scope. The transformations are implemented using LLVMs Clang front end.

C-Reduce does not rely on Csmith. It can reduce programs whether they are generated by Csmith or not. To ensure any transformation in C-Reduce does not introduce undefined behaviors, Regehr et al. [54] use semantics-checking C interpreters such as KCC [15] and Frama-C [9] to detect invalid transformations.

Regehr et al. [54] use 98 bug-inducing programs created by Csmith as the inputs to the three above reducers. The most effective reducer (in terms of size of output) is C-Reduce:

reducing test cases with average 58 KB file sizes into average 258-byte test cases. Regehr et al. found the outputs of C-Reduce are usually good enough to be directly copied into bug reports.

7.2.2 Swarm testing using Csmith

Csmith can be configured in many different ways by providing command line options. For example, it can be configured to not generate 64-bit integers and arithmetics in random programs. Facing so many options, the foremost question is this: Should we use only one default configuration or experiment with different configurations during random testing?

Groce et al. tried to answer the question [26] with Csmith-based swarm testing. The default configuration of Csmith errs on the side of expressiveness: enabling most of the parts of the C language. But most of the expressiveness can be turned off by command line arguments. Groce et al. experimented with different configurations. Each configuration includes a collection of options including enabling or disabling C features such as comma operators, compound assignments, embedded assignments, increment and decrement operators, goto-statements, and 64-bit math operations. In their swarm testing, a configuration is used to generate 1,000 test cases before moving to the next configuration.

They reported that a week-long swarm testing found 104 distinct ways to crash a collection of compilers including GCC, LLVM, Sun CC, and Intel CC.The default configuration found only 73 ways, an improvement of 42%. Swarm testing also slightly improved code coverage of two compilers: LLVM/Clang 2.9 and GCC 4.6.0.

Of these 104 crash symptoms, 52 of them had at least one feature whose presence was significant and 42 had at least one feature whose absence was significant. While they found that pointers, arrays, and structures are great triggers of compiler bugs, it was surprising to find that pointers, embedded assignments, jumps, arrays, and the auto increment/decrement operators are high in the list of bug suppressors. The lesson learned is that some features (most notably, pointers) strongly trigger some bugs while strongly suppressing others. This observation directly validates swarm testing.

7.2.3 Validation of C11/C++11 memory model

The C and C++ languages were originally designed without concurrency support. With the recently approved C11/C++11 memory model [61] a precise semantics is defined for threads accessing shared memories: well-synchronized programs must exhibit only sequentially consistent behaviors, racy programs are undefined, while low level atomics enables high-performance yet defined code.

The question is how to validate compilers' implementation of such memory model, and how does random differential testing help? Concurrency compiler bugs can be identified by comparing the memory trace of compiled code against a reference memory trace for the source code. However, a naive comparison would fail because concurrent programs are inherently nondeterministic and optimizers can compile away nondeterminism. Two executables might have different memory traces, and yet both be correct with respect to a source C11 or C++11 program.

Morisset et al. [46] proposed a solution by reducing the problem to validating the traces generated by running optimized sequential code against a reference (unoptimized) trace for the same code. Their comparison algorithm allows sound transformations, such as eliminations, reorderings, and introductions of actions, as long as the transformations conform to the C11/C++11 concurrency model.

With some help from me, The team extended Csmith to generate random concurrent programs. Some extensions are 1) adding mutex variables (as defined in pthread.h), and 2) adding system calls to pthread mutex lock and pthread mutex unlock. Since Csmith generates jump statements with arbitrary destinations, enforcing balancing of calls to lock and unlock along all possible execution paths is difficult, so mutex variables are declared with the attribute PTHREAD MUTEX RECURSIVE.

Another validation is based on the observation that each compiler has its own set of internal invariants. For instance, GCC, as of 2013, forbids all reorderings of a memory access with an atomic one. The comparison algorithm was tuned to check for violations of this kind of invariant.

The team reported four concurrency GCC bugs which have all been promptly fixed by the compiler developers. All these are silent wrong-code bugs. They do not only break the C11/C++11 memory model, but also the Posix DRF guarantee which is assumed by most concurrent software written in C and C++.

7.2.4 Triage of failure-inducing test cases

Csmith is impressively effective at finding compiler bugs. Ironically, the effectiveness introduces difficulty to its users: an overnight run may result in hundreds or thousands of failure-inducing test cases. Moreover, some bugs tend to be triggered much more often than others, sometimes thousands of times more, creating needle-in-a-haystack problems. Sometimes Csmith keeps generating test cases that trigger noncritical bugs that may also already be known. As a random tester, I have found test case triage is time-consuming and unrewarding.

Chen et al. [8] defined the fuzzer taming problem this way: "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." They solved the problem by finding distance (similarity) between failure inducing test cases, and ordering them with FPF (furthest point first). The FPF computation takes into account a list of bugs known to be uninteresting, and lowers the rank of test cases corresponding to them.

Instead of using a universal distance function working for all bug trigger patterns, Chen et al. defined a collection of distance functions, each of which captures the usefulness of some kinds of bug triggers. The functions include Levenshtein distance and Euclidean distance, both computed with a normalized input.

Chen et al. applied different distance functions to 1,979 reduced test cases triggering 46 bugs (11 compile-time bugs and 35 wrong-code bugs) in GCC 4.3.0. For compile-time bugs, the best distance function is the normalized Levenshtein distance between test cases plus normalized Levenshtein distance between failure outputs. The result tracks the "theoretical best" line, i.e, examining each additional test case reveals a new distinct bug, and the discovery keeps going until all bugs are revealed.

Test cases inducing wrong-code bugs are more difficult to be auto-triaged: they produce no compiler output indicating failures, and in compilers, the execution distance between the buggy code and code emission phase could be long. The best distance function is Euclidean distance between function coverage on the compiler.

7.3 Random testing other compilers/interpreters

While Csmith focuses on generating C programs, there are fuzzing tools targeting other programming languages. They are most successful in finding bugs in JavaScript engines in various browsers. Following are a few examples.

jsfunfuzz [55] is a JavaScript fuzzer used for testing the JavaScript engine in Firefox. It tests the JavaScript language engine itself, not the DOM. It has found over 1,500 bugs in Firefox's JavaScript engine, and most of them have already been fixed. Many were memory safety bugs and are believed to be exploitable to run arbitrary code. The author attributed its success to its knowledge of how JavaScript is parsed and interpreted, and applying the knowledge toward random program generations. The fuzzer can be used for differential testing because a compiled Java script can be decompiled to produce the second test case to be compared.

LangFuzz [28] is a JavaScript fuzzer based on a grammar, and thus is more generic than jsfunfuzz which targets only the JavaScript engine in Firefox. Given a PHP grammar, it can be used to generate PHP programs. Moreover, LangFuzz can use its grammar to learn code fragments from a given code base. Given a suite of previously failing programs, LangFuzz will use and recombine fragments of the provided test suite to generate new programs, making it more effective. Applied on the Mozilla JavaScript interpreter, it discovered a total of 105 new severe vulnerabilities within three months of operation.

Google has revealed it is using **ClusterFuzz** [2] to generate millions of test cases on a daily basis and apply them to its Chrome browser. The test cases are mostly targeted to Chrome's JavaScript engine V8. The name "ClusterFuzz" comes from a cluster of several hundred virtual machines running approximately six-thousand simultaneous Chrome instances. Like us, Google team also faces problems such as test infrastructure management, test case reduction, and bug triage. ClusterFuzz had already detected 95 unique vulnerabilities in a four-month period when it was announced to the outside world.

7.4 Future directions of Csmith

7.4.1 Supporting object oriented programming (OOP) languages

Csmith generates only random programs in C, which is a procedural language. However, many popular programming languages are object oriented, such as **C++**, **Java**, **C#**, **Objective-C**, and JavaScript. There are no official rankings of program languages. TIOBE Programming Community Index [62] is an unofficial ranking of programming languages based on popularity obtained from search engines. As of July 2013, C/C++, Java, and Object-C combined account for over half of the total popularities. C is the leading language followed closely by Java.

Can we adapt Csmith so that random programs in these top-ranking languages can be generated? The answer is probably no. In Csmith, the internal representation (the abstract syntax tree), is closely aligned with C syntax, and the generation time analysis and validation (GTAV) is strictly based on C semantics. To support a new language, three prerequisites must be met:

- Constructs for the new language must be added to the internal representations. The most notable ones are class, object, and exception. They are missing from current Csmith (version 2.1) tree representations. Class would be a new storage scope besides

the two currently supported scope in Csmith (version 2.1): global and block. Different languages have different libraries. They should be represented as well.

- A GTAV must be implemented according to the language semantics. Both analyses and validation at generation time vary from language to language. For example, the evaluation order of function parameters is unspecified in C, but specified in Java. In addition, C tends to have the largest set of undefined behaviors among all above languages, thus, its validation is the most strict. We expect many of the GTAV logics to be shared across languages, at least across the enlarged C-family including C/C++, C#, and Objective-C.

- GTAV must be made dynamically loadable. Each language should have its own GTAV, which is loaded when the generator is targeted to the particular language. Currently in version 2.1, code generation and GTAV are intermingled. It requires an architectural change to Csmith to separate both modules.

Considering the efforts for these tasks, I feel it is more economic to redesign and reimplement Csmith. I envision the next incarnation of our random generator to be generic and simple, with most of the complexity moving to a syntax description file. A semantic description file for assisting GTAV could also be used.

7.4.2 Supporting low level representations

C is a low level programming language. I think moving Csmith toward supporting other low level representations is easier than moving Csmith toward supporting higher programming languages. Java byte-code, assembly languages, and machine languages are a few examples of low level representations.

Since Csmith is generating random C programs which can be compiled into assembly code or machine code, why should we generate assembly/machine code randomly? The problem is that compilers tend to normalize the randomness during compilation. A good example is given by Yee et al. [76]: generating random C programs and then compiling them into x86 machine code does not exercise instructions that are avoided by compilers/assemblers, such as a typical MOV instruction.

It is infeasible to create random generators for a vast amount of target platforms. We can simplify the problem by targeting the intermediate representations in compilers. GCC RTL and LLVM IR are good candidates. Once we are able to generate random programs in a compiler IR, the compiler's back end can be used to convert the random program into

desirable assembly code or machine code. The randomness could still be normalized by the back end, but the problem is less severe than compiling the random programs all the way from C to assembly/machine code. LLVM IR is especially valuable in that it can be compiled into the byte-code of virtual machines and even OpenGL code. Implementing a random generator targeting LLVM IR enables us to random test a large set of execution environments, physical or virtual.

7.4.3 Supporting new target platforms

We have received requests from a couple of projects planning to port GCC to a new target platform. Essentially, a target-specific back end needs to be implemented and tested. In one instance, we helped a team to modify Csmith. And in another instance, the team modified Csmith according to their own needs. The required changes are minimal since Csmith generates programs in C, which is target-independent. However, for a platform based on microcontrollers with limited capabilities, Csmith's expressiveness sometimes needs to be limited. For example, certain language constructs that are not available on the platform have to be disabled.

A compiler back end can be implemented in parallel with the hardware. Before the hardware is ready, the teams use simulation for implementing and testing their compilers. It was reported to us that Csmith found over 40 bugs in one of the compiler implementations quickly. Being able to develop software and hardware at the same time is a great time saver.

7.4.4 Supporting static analyzers

Cuoq et al. has shown a way [9] to validate static analysis using Csmith by turning the analyzer into an interpreter and compare its outputs from random programs with the outputs from a reference compiler. For static analyzers that are not easy to be converted into an interpreter, there are alternative ways of random testing.

Csmith performs path-sensitive static analysis at generation time. It (in version 2.1) currently performs points-to analysis and side-effect analysis. Extending to other analyses such as reaching definition, liveness analysis, use-define chain, is possible. The analysis results are path-joined (thus, becoming path-insensitive) by the time random generation finishes. A straight forward validation is comparing the path-insensitive results from Csmith with the results from a static analyzer. Csmith could expose its analysis results in the random programs as comments. A static analyzer could read the results and validate against its own.

If a static analyzer performs path-sensitive analysis, its results can be validated against Csmith's results. As of 2014, Csmith does not store path-sensitive results after computing them, but it can be easily made so.

7.5 Toward bug-free compilers

Can the compilers be improved to a point where they are essentially bug free? We have frequently observed that new bugs are introduced while the developers are 1) adding a new front end; 2) adding a new back end; 3) adding a new transformation; and 4) fixing a bug (ironically). An evolving compiler has to undergo the above code changes periodically, and unless all above activities are protected by correctness guarantees, bugs are inevitable.

CompCert offers some hope to the correctness problem. It sticks to a semantic preservation guarantee which states:

> For all source programs S and compiler-generated code C, if the compiler, applied to the source S, produces the code C, without reporting a compile-time error, then the observable behavior of C improves on one of the allowed observable behaviors of S.

Observable behaviors include 1) whether the program is halting or nonhalting; 2) all calls to standard library functions that perform input/output; and 3) all read and write accesses to global variables of volatile types. A semantic preserving compiler is allowed to fail at compile-time or optimize away runtime error when safe.

With the help of the Coq proof assistant, CompCert developers have verified that the compiler is guaranteed to preserve semantics during the translation from a C-AST to an assembly AST [34]. Their argument is strengthened by experimental results from our random testing using Csmith. During our year-long testing of CompCert, Csmith was able to find six bugs, but none of them are in the verified part.

Unfortunately, CompCert has its own limitations. First, the parsing phase (from C to C AST) or the assembling phase (from assembly AST to machine code) are still left to be proven, and thus are possibly buggy. The CompCert team explains that those phases lack mathematical specifications, making it difficult to state, let alone prove, the correctness theorem.

Second, CompCert, as of 2014, has limited optimizations including inlining, tail call optimization, constant propagation, common subexpression elimination, live range splitting, graph-coloring-based register allocation and coalescing, instruction selection and combining. Notably, it does not support loop optimizations.

Third, CompCert does not support full C99 specification. And its back ends are limited to x86, 32-bit PowerPC, or ARM v2 and above.

Even if CompCert is able to eliminate all the limitations, we are still unsure whether the engineering activities identified at the beginning of the section, e.g., adding new front ends or back ends, are protected or not.

In summary, we expect the compiler qualities to improve over time, but a bug-free compiler is still a far stretched goal.

7.6 Toward error-free compilations

If a bug-free compiler is impossible near term, the next goal is naturally to ensure the compilations of a subset of programs are error free. In most cases, the programs we care about are safety critical or other high-importance programs. The logic is that if we validate a program's safety at the source level, and we also validate that the compilation is faithful, then the program is validated at the binary level, and therefore poses no threat when executed on a trusted computing base.

Research in translation validation has made progress over the last few years. Tristan et al. [63] presented a scalable solution that validates whether the normalized value-graphs of a pretranslation program and a posttranslation program are equivalent. The transformations being validated include a broad set of intra-procedural optimizations in LLVM, and they were able to scale the validation to large programs such as GCC, SQLite3, and several programs drawn from SPEC CPU 2006. The key insight is that the pretranslations value-graphs have to be simplified (normalized) as compiler optimizations would do. However, there were still approximately 20% of the functions that could not been proven that they were translated faithfully.

Sewell et al. [57] focused on validating the translation of seL4 , a much studied microkernel with around 9,500 lines of C source code. The first contribution is that they created a representation that can express both C programs and binary programs, and is highly amenable to SMT solvers. They then deduced C-semantics from the source code with Isabelle/HOL [64], and deduced the binary-semantics from the binary code with HOL4 [48], and converted both semantics into the above-mentioned representations for validation, which is carried out by SMT solvers. They were able to validate translations of all functions in seL4 except for the assembly routines and functions involving volatile accesses with zero false positives. However, several limitations remain: 1) they can fully validate the translations at optimization level "gcc -O1", but not at -O2; 2) they have to

modify the source code of seL4 to work around the restrictions caused by the C-parser, the decompiler, and other components of the validation.

CHAPTER 8

CONCLUSION

We have developped a random program generator, Csmith, that performs program analysis and code generation in parallel. These two activities work hand-by-hand: code generation invalidate previous program analysis results and requires new rounds of analysis, while analysis results are used to guide the code generator. The approach resulted in a random program generator that produces unambiguous code which is essential for random differential testing. Although Csmith is slower in generating code than previously published random program generators, most likely due to generation time analysis, it is empirically proven to be more expressive and with higher bug-finding performance.

For years, Csmith was able to consistently find bugs in widely used compilers such as GCC and LLVM, despite the fact that most bugs we reported were fixed in a short period of time. Although we only tested the compilers with C programs, the bugs we found are mostly located in the middle end and back end of the compilers, and thus are likely to impact compiler users even if they program in other languages. Everybody, especially developers of safety critical applications, should be alarmed of the fact that bugs are present in the strongest compilers, and errors occur with the lowest level of compiler optimization.

APPENDIX

GENERATION GRAMMAR OF CSMITH

⟨*empty*⟩ ::= ‘’

⟨*space*⟩ ::= (‘ ’)+

⟨*new-line*⟩ ::= (‘`\n`’)+

⟨*block open*⟩ ::= ⟨*new-line*⟩ ‘`{`’ ⟨*new-line*⟩

⟨*block close*⟩ ::= ⟨*new-line*⟩ ‘`}`’ ⟨*new-line*⟩

⟨*decimal digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’

⟨*decimal digits*⟩ ::= ⟨*digit*⟩+

⟨*hex digit* ⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’ | ‘`A`’ | ‘`B`’ | ‘`C`’ | ‘`D`’ | ‘`E`’ | ‘`F`’

⟨*hex prefix*⟩ ::= ‘`0x`’

⟨*hex suffix*⟩ ::= ‘`L`’ | ⟨*empty*⟩

⟨*long long suffix*⟩ ::= ‘`LL`’ | ⟨*empty*⟩

⟨*8-bit hex*⟩ ::= ⟨*hex digit*⟩ ⟨*hex digit*⟩ ⟨*hex digit*⟩ ⟨*hex digit*⟩

⟨*16-bit hex*⟩ ::= ⟨*8-bit hex*⟩ ⟨*8-bit hex*⟩

⟨*32-bit hex*⟩ ::= ⟨*16-bit hex*⟩ ⟨*16-bit hex*⟩

⟨*64-bit hex*⟩ ::= ⟨*32-bit hex*⟩ ⟨*32-bit hex*⟩

⟨*hexadecimal constant*⟩ :: ⟨*hex prefix* ⟩ ⟨*8-bit hex*⟩ ⟨*hex suffix*⟩
| ⟨*hex prefix*⟩ ⟨*16-bit hex*⟩ ⟨*hex suffix*⟩
| ⟨*hex prefix*⟩ ⟨*32-bit hex*⟩ ⟨*hex suffix*⟩
| ⟨*hex prefix*⟩ ⟨*64-bit hex*⟩ ⟨*long long suffix*⟩

⟨*decimal prefix*⟩ ::= '`-`' | ⟨*empty*⟩

⟨*decimal suffix*⟩ ::= '`L`' | '`LL`' | '`UL`' | '`U`' | ⟨*empty*⟩

⟨*decimal constant*⟩ ::= ⟨*decimal prefix*⟩ ⟨*decimal digits*⟩ ⟨*decimal suffix*⟩

⟨*global variable identifier*⟩ ::= '`g`' '`_`' ⟨*decimal digits*⟩

⟨*local variable identifier*⟩ ::= '`l`' '`_`' ⟨*decimal digits*⟩

⟨*parameter identifier*⟩ ::= '`p`' '`_`' ⟨*decimal digits*⟩

⟨*variable identifier*⟩ ::= ⟨*global variable identifier*⟩ | ⟨*local variable identifier*⟩
| ⟨*parameter identifier*⟩

⟨*function identifier*⟩ ::= '`func`' '`_`' ⟨*decimal digits*⟩

⟨*jump label identifier*⟩ ::= '`lbl`' '`_`' ⟨*decimal digits*⟩

⟨*struct type identifier*⟩ ::= '`S`' ⟨*decimal digits*⟩

⟨*union type identifier*⟩ ::= '`U`' ⟨*decimal digits*⟩

⟨*field identifier*⟩ ::= '`f`' ⟨*decimal digits*⟩

⟨*integer constant*⟩ ::= ⟨*decimal constant*⟩ | ⟨*hexadecimal constant*⟩

⟨*pointer constant*⟩ ::= '`(void*)0`'

⟨*struct constant*⟩ ::= '{' ((⟨*integer constant*⟩ | ⟨*struct constant*⟩) ',')+ '}'

⟨*union constant*⟩ ::= '{' ((⟨*integer constant*⟩ | ⟨*union constant*⟩) ',')+ '}'

⟨*constant*⟩ ::= ⟨*integer constant*⟩ | ⟨*pointer constant*⟩ | ⟨*struct constant*⟩
| ⟨*union constant*⟩

⟨*storage specifier*⟩ ::= '`static`' | ⟨*empty*⟩

⟨*type qualifier*⟩ ::= '`const`' | '`volatile`'

⟨*type qualifiers*⟩ ::= ⟨*type qualifier*⟩ ⟨*space*⟩ ⟨*type qualifiers*⟩
| ⟨*empty*⟩

⟨*type specifier*⟩ ::= '`void`'
| '`int8_t`' | '`int16_t`' | '`int32_t`' | '`int64_t`'
| '`uint8_t`' | '`uint16_t`' | '`uint32_t`' | '`uint64_t`'
| '`struct`' ⟨*space*⟩ '`S`' ⟨*decimal digits*⟩
| '`union`' ⟨*space*⟩ '`U`' ⟨*decimal digits*⟩

⟨*qualified non-pointer type*⟩ ::= ⟨*type qualifiers*⟩ ⟨*type specifier*⟩

⟨*qualified pointer type*⟩ ::= (⟨*qualified non-pointer type*⟩ | ⟨*qualified pointer type*⟩)
⟨*space*⟩ '`*`' ⟨*type qualifiers*⟩

⟨*qualified type*⟩ ::= ⟨*qualified non-pointer type*⟩ | ⟨*qualified pointer type*⟩

⟨*array dimension*⟩ ::= '`[`' ⟨*decimal digits*⟩ '`]`'

⟨*array dimensions*⟩ ::= ⟨*array dimension*⟩+

⟨*non-array declarator*⟩ ::= ⟨*storage specifier*⟩ ⟨*qualified type*⟩ ⟨*variable identifier*⟩

⟨*non-array initializer*⟩ ::= ⟨*constant*⟩

⟨*array declarator*⟩ ::= ⟨*declarator*⟩ ⟨*array dimensions*⟩

⟨*array initializer*⟩ ::= '{' ⟨*constant*⟩ (', ' ⟨*constant*⟩)* '}'
| '{' ⟨*array initializer*⟩ (', ' ⟨*array initializer*⟩)* '}'

⟨*declarator with initializer*⟩ ::= ⟨*non-array declarator*⟩ ' = ' ⟨*non-array initializer*⟩
| ⟨*array declarator*⟩ ' = ' ⟨*array initializer*⟩

⟨*global variable declarator*⟩ ::= 'static' ⟨*space*⟩ ⟨*declarator with initializer*⟩

⟨*global variable declarators*⟩ ::= (⟨*global variable declarator*⟩ ';' ⟨*new-line*⟩)*

⟨*local variable declarator*⟩ ::= ⟨*declarator with initializer*⟩

⟨*local variable declarators*⟩ ::= (⟨*local variable declarator*⟩ ';' ⟨*new-line*⟩)*

⟨*constant expression*⟩ ::= ⟨*integer constant*⟩ | ⟨*pointer constant*⟩

⟨*simple variable expression*⟩ ::= ⟨*variable identifier*⟩ | ⟨*simple variable expression*⟩
'.' ⟨*field identifier*⟩

⟨*dereferenced variable expression*⟩ ::= '*' (⟨*simple variable expression*⟩
| ⟨*dereferenced variable expression*⟩)

⟨*escaped variable expression*⟩ ::= '&' ⟨*simple variable expression*⟩

⟨*variable expression*⟩ ::= ⟨*simple variable expression*⟩
| ⟨*dereferenced variable expression*⟩
| ⟨*escaped variable expression*⟩

⟨*l-value expression*⟩ ::= ⟨*simple variable expression*⟩
| ⟨*dereferenced variable expression*⟩

⟨*unary operator*⟩ ::= '+' | '-' | '!' | '~'

⟨*unary expression*⟩ ::= ⟨*unary operator*⟩ '(' ⟨*expression*⟩ ')'

⟨*binary operator*⟩ ::= '+' | '-' | '*' | '/' | '%'
| '>' | '<' | '>=' | '<=' | '==' | '!='

| ‘&&’ | ‘||’ | ‘^’ | ‘&’ | ‘|’
| ‘>>’ | ‘<<’

⟨*binary expression*⟩ ::= ⟨*expression*⟩ ⟨*binary operator*⟩ ⟨*expression*⟩

⟨*comma expression*⟩ ::= ⟨*expression*⟩ ‘,’ ⟨*expression*⟩

⟨*assignment operator*⟩ ::= ‘=’ | ‘*=’ | ‘/=’ | ‘%=’ | ‘+=’ | ‘-=’
| ‘<<=’ | ‘> >=’ | ‘&=’ | ‘^=’ | ‘|=’

⟨*increment decrement operator*⟩ ::= ‘++’ | ‘--’

⟨*increment decrement expression*⟩ ::= ⟨*increment decrement operator*⟩ ⟨*l-value expression*⟩
| ⟨*l-value expression*⟩ ⟨*increment decrement operator*⟩

⟨*assignment expression*⟩ ::= ⟨*l-value expression*⟩ ⟨*assignment operator*⟩ ⟨*expression*⟩
| ⟨*increment decrement expression*⟩

⟨*parameter type*⟩ ::= ⟨*integer type*⟩ | ⟨*pointer type*⟩

⟨*argument*⟩ ::= ⟨*expression*⟩

⟨*arguments*⟩ ::= ⟨*argument*⟩ (‘,’ ⟨*argument*⟩)*

⟨*call expression*⟩ ::= ⟨*function identifier*⟩ ‘(’ ⟨*arguments*⟩ ‘)’

⟨*expression*⟩ ::= ⟨*constant expression*⟩
| ⟨*variable expression*⟩
| ⟨*unary expression*⟩
| ⟨*binary expression*⟩
| ⟨*comma expression*⟩
| ⟨*assignment expression*⟩
| ⟨*call expression*⟩

⟨*labelled statement*⟩ ::= ⟨*jump label identifier*⟩ ‘:’ ⟨*statement*⟩

⟨*return statement*⟩ ::= ‘return’ ⟨*space*⟩ ⟨*variable expression*⟩

⟨*assignment statement*⟩ ::= ⟨*assignment expression*⟩

⟨*comparison operator*⟩ ::= '`>`' | '`<`' | '`>=`' | '`<=`' | '`==`' | '`!=`'

⟨*loop initialization*⟩ ::= ⟨*l-value expression*⟩ '`=`' ⟨*integer constant*⟩

⟨*loop termination test*⟩ ::= ⟨*l-value expression*⟩ ⟨*comparison operator*⟩ ⟨*integer constant*⟩

⟨*for-loop statement*⟩ ::= '`for`' '`(`' ⟨*loop initialization*⟩ '`;`'
⟨*loop termination test*⟩ '`;`'
⟨*increment decrement expression*⟩'`)`'
⟨*new-line*⟩ ⟨*block*⟩

⟨*if-else statement*⟩ ::= '`if`' '`(`' ⟨*expression*⟩ '`)`'
⟨*new-line*⟩ ⟨*block*⟩
'`else`'
⟨*new-line*⟩ ⟨*block*⟩

⟨*call statement*⟩ ::= ⟨*call expression*⟩

⟨*break statement*⟩ ::= '`break`'

⟨*continue statement*⟩ ::= '`continue`'

⟨*goto statement*⟩ ::= '`if`' '`(`' ⟨*variable expression*⟩ '`)`'
⟨*new-line*⟩ '`goto`' ⟨*space*⟩ ⟨*jump label identifier*⟩

⟨*array iteration statement*⟩ ::= ⟨*for-loop statement*⟩

⟨*statement*⟩ ::= ⟨*labelled statement*⟩
| ⟨*return statement*⟩
| ⟨*assignment statement*⟩
| ⟨*for-loop statement*⟩
| ⟨*if-else statement*⟩
| ⟨*call statement*⟩
| ⟨*break statement*⟩
| ⟨*continue statement*⟩

| ⟨*goto statement*⟩
| ⟨*array iteration statement*⟩

⟨*block*⟩ ::= ⟨*block open*⟩
⟨*local variable declarators*⟩
(⟨*statement*⟩ '`;`' ⟨*new-line*⟩)+
⟨*block close*⟩

⟨*pre-struct pragma*⟩ ::= '`#pragma pack(push)`' ⟨*new-line*⟩ '`#pragma pack(1)`'

⟨*post-struct pragma*⟩ ::= '`#pragma pack(pop)`'

⟨*non-bitfield field*⟩ ::= ⟨*type qualifiers*⟩ ⟨*space*⟩ ⟨*type specifiers*⟩
⟨*space*⟩ ⟨*field identifier*⟩

⟨*signedness*⟩ ::= '`signed`' | '`unsigned`'

⟨*bitfield field*⟩ ::= ⟨*signedness*⟩ ⟨*space*⟩ ⟨*field identifier*⟩ '`:`' ⟨*decimal digits*⟩

⟨*anonymous field*⟩ ::= ⟨*signedness*⟩ ⟨*space*⟩ '`:`' '`0`'

⟨*struct union field*⟩ ::= ⟨*non-bitfield field*⟩ | ⟨*bitfield field*⟩ | ⟨*anonymous field*⟩

⟨*struct union fields*⟩ ::= (⟨*struct union field*⟩ '`;`')+

⟨*struct definition*⟩ ::= '`struct`' ⟨*space*⟩ ⟨*struct type identifier*⟩
'`{`' ⟨*struct union fields*⟩ '`}`'

⟨*struct type declarator*⟩ ::= ⟨*pre-struct pragma*⟩ ⟨*struct definition*⟩ ⟨*post-struct pragma*⟩

⟨*struct type declarators*⟩ ::= (⟨*struct type declarator*⟩ '`;`')*

⟨*union definition*⟩ ::= '`union`' ⟨*space*⟩ ⟨*union type identifier*⟩
'`{`' ⟨*struct union fields*⟩ '`}`'

⟨*union type declarator*⟩ ::= ⟨*union definition*⟩

⟨*union type declarators*⟩ ::= (⟨*union type declarator*⟩ ‘;’)*

⟨*include header*⟩ ::= ‘`#include ‘‘csmith.h”` ⟨*new-line*⟩

⟨*define header*⟩ ::= ‘`static long __undefined;`’ ⟨*new-line*⟩

⟨*headers*⟩ ::= ⟨*include header*⟩ ⟨*define header*⟩

⟨*parameter* ⟩ ::= ⟨*parameter type*⟩ ⟨*space*⟩ ⟨*parameter identifier*⟩

⟨*parameters*⟩ ::= ⟨*parameter*⟩ (‘`,`’ ⟨*parameter*⟩)*
| ⟨*empty*⟩

⟨*function declarator*⟩ ::= ⟨*qualified type*⟩ ⟨*space*⟩ ⟨*function identifier*⟩
‘`(`’ ⟨*parameters*⟩ ‘`)`’

⟨*function declarators*⟩ ::= (‘`static`’ ⟨*space*⟩ ⟨*function declarator*⟩ ‘`;`’ ⟨*new-line*⟩)+

⟨*function definition*⟩ ::= ⟨*function declarator*⟩ ⟨*block*⟩

⟨*function definitions*⟩ ::= (⟨*function definition*⟩ ⟨*new-line*⟩)+

⟨*program*⟩ ::= ⟨*headers*⟩
⟨*struct type declarators*⟩
⟨*union type declarators*⟩
⟨*global variable declarators*⟩
⟨*function declarators*⟩
⟨*function definitions*⟩

REFERENCES

[1] Apple Inc. Developer tools overview. `https://developer.apple.com/technologies/tools/`.

[2] Arya, A., and Neckar, C. Chromium blog: Fuzzing for security. `http://blog.chromium.org/2012/04/fuzzing-for-security.html`.

[3] Balston, K., Hu, A. J., Wilton, S. J. E., and Nahir, A. Emulation in post-silicon validation: It's not just for functionality anymore. *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)* (2012), 110–117.

[4] Boujarwah, A. S., and Saleh, K. Compiler test case generation methods: a survey and assessment. *Information and Software Technology 39*, 9 (1997), 617–625.

[5] Burgess, C. J., and Saidi, M. The automatic generation of test cases for optimizing Fortran compilers. *Information & Software Technology 38*, 2 (1996), 111–119.

[6] Campbell, B. An executable semantics for CompCert C. In *Proceedings of the Second International Conference on Certified Programs and Proofs* (Berlin, Heidelberg, 2012), CPP'12, Springer-Verlag, pp. 60–75.

[7] Cem Kaner, Jack Falk, H. Q. N. *Testing Computer Software*. Wiley, 1999.

[8] Chen, Y., Groce, A., Zhang, C., Wong, W.-K., Fern, X., Eide, E., and Regehr, J. Taming compiler fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), PLDI '13, ACM, pp. 197–208.

[9] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods* (Berlin, Heidelberg, 2012), SEFM'12, Springer-Verlag, pp. 233–247.

[10] Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., and Yang, X. Testing static analyzers with randomly generated programs. In *Proceedings of the 4th International Conference on NASA Formal Methods* (Berlin, Heidelberg, 2012), NFM'12, Springer-Verlag, pp. 120–125.

[11] Cuoq, P., Signoles, J., Baudin, P., Bonichon, R., Canet, G., Correnson, L., Monate, B., Prevosto, V., and Puccetti, A. Experience report: OCaml for an industrial-strength static analysis framework. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (2009), ICFP '09, ACM, pp. 281–286.

[12] Diehl, S. Natural semantics-directed generation of compilers and abstract machines. *Formal Aspects of Computing 12*, 2 (2000), 71–99.

[13] Eide, E. Compiler-testing tools. http://www.cs.utah.edu/~eeide/emsoft08/.

[14] Eide, E., and Regehr, J. Volatiles are miscompiled, and what to do about it. In *EMSOFT* (2008), ACM, pp. 255–264.

[15] Ellison, C., and Rosu, G. An executable formal semantics of C with applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 533–544.

[16] GNU. GCC, the GNU compiler collection. http://gcc.gnu.org/.

[17] GNU. Gcc_research. http://gcc.gnu.org/wiki/GCC_Research.

[18] GNU. How to minimize test cases for bugs. http://gcc.gnu.org/bugs/minimize.html.

[19] GNU. Installing GCC: Testing. http://gcc.gnu.org/install/test.html.

[20] GNU. Language standards supported by gcc. http://gcc.gnu.org/onlinedocs/gcc/Standards.html#Standards.

[21] GNU. Managing bugs. http://gcc.gnu.org/bugs/management.html.

[22] GNU. A new project to merge the existing GCC forks. http://gcc.gnu.org/news/announcement.html.

[23] GNU. People - gcc wiki. http://gcc.gnu.org/wiki/People.

[24] GNU. Status of supported architectures from maintainers' point of view. http://gcc.gnu.org/backends.html.

[25] GNU. Volatile objects - autoconf. http://www.gnu.org/software/autoconf/manual/autoconf-2.67/html_node/Volatile-Objects.html.

[26] Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J. Swarm testing. In *ISSTA* (2012), pp. 78–88.

[27] Hanford, K. V. Automatic generation of test cases. *IBM Syst. J. 9*, 4 (Dec. 1970), 242–257.

[28] Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 445–458.

[29] ISO/IEC. ISO/IEC 9899:201x. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[30] ISO/IEC. ISO/IEC 9899:TC2. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[31] ISTQB. *Certified Tester Foundation Level Syllabus*. 2011. http://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html.

[32] Johnstone, A., and Scott, E. a grammar for the 1989 ANSI standard C language. http://www.cs.rhul.ac.uk/research/languages/projects/grammars/c/c89/ansi_c_89/ansi_c_89.raw.

[33] Leitner, A., Ciupa, I., Oriol, M., Meyer, B., and Fiva, A. Contract driven development = test driven development - writing test cases. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2007), ESEC-FSE '07, ACM, pp. 425–434.

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

[35] Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schatz, B. R., and Wulf, W. A. An overview of the production-quality compiler-compiler project. *Computer 13*, 8 (Aug. 1980), 38–49.

[36] Lindig, C. quest-tester - automatically test calling conventions of C compilers. `http://code.google.com/p/quest-tester/`.

[37] Lindig, C. Random testing of C calling conventions. In *AADEBUG* (2005), pp. 3–12.

[38] LLVM. Clang - C language family frontend for LLVM. `http://clang.llvm.org/`.

[39] LLVM. LLVM bugpoint tool. `http://llvm.org/docs/Bugpoint.html`.

[40] LLVM. The LLVM compiler infrastructure. `http://www.llvm.org/`.

[41] LLVM. LLVM language reference manual. `http://llvm.org/docs/LangRef.html`.

[42] LLVM. LLVM related publications. `http://llvm.org/pubs/`.

[43] LLVM. LLVM testing infrastructure guide. `http://llvm.org/docs/TestingGuide.html`.

[44] McKeeman, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.

[45] McPeak, S., and Wilkerson, D. S. Delta. `http://delta.tigris.org/`.

[46] Morisset, R., Pawan, P., and Zappa Nardelli, F. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), PLDI '13, ACM, pp. 187–196.

[47] Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B., and Gan, E. Rocksalt: better, faster, stronger SFI for the x86. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[48] Myreen, M. O., Gordon, M. J. C., and Slind, K. Decompilation into logic - improved. In *FMCAD* (2012), G. Cabodi and S. Singh, Eds., IEEE, pp. 78–81.

[49] Naroff, S. New LLVM C front-end. `http://llvm.org/devmtg/2007-05/09-Naroff-CFE.pdf`.

[50] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[51] Perennial Inc. ACVS ANSI/ISO/FIPS-160 C validation suite. http://www.peren.com/pages/acvs_set.htm.

[52] Plum Hall Inc. The Plum Hall validation suite for C. http://www.plumhall.com/stec.html.

[53] Purdom, P. A sentence generator for testing parsers. *BIT Numerical Mathematics 12*, 3 (1972), 366–375.

[54] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[55] Ruderman, J. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[56] Sauder, R. L. A general test data generator for cobol. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference* (New York, NY, USA, 1962), AIEE-IRE '62 (Spring), ACM, pp. 317–323.

[57] Sewell, T. A. L., Myreen, M. O., and Klein, G. Translation validation for a verified os kernel. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 471–482.

[58] Sheridan, F. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience 37*, 14 (2007), 1475–1488.

[59] Software Engineering Institue, C. M. U. 04. integers (int) - secure coding - CERT secure coding standards. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=270.

[60] Tarski, A. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics 5*, 2 (1955), 285–309.

[61] The C++ Standards Committee. ISO/IEC JTC1/SC22/WG21. http://www.open-std.org/jtc1/sc22/wg21/.

[62] TIOBE Software. TIOBE software: The coding standards company. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[63] Tristan, J.-B., Govereau, P., and Morrisett, G. Evaluating value-graph translation validation for llvm. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 295–305.

[64] Tuch, H., Klein, G., and Norrish, M. Types, bytes, and separation logic. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 97–108.

[65] Turner, B. random C program generator. https://sites.google.com/site/brturn2/randomcprogramgenerator/.

[66] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[67] Wikipedia. Compiler - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Compiler`.

[68] Wikipedia. Compiler correctness - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Compiler_correctness`.

[69] Wikipedia. Fuzz testing. `http://en.wikipedia.org/wiki/Fuzz_testing`.

[70] Wikipedia. GNU compiler collection - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/GNU_Compiler_Collection`.

[71] Wikipedia. History of compiler construction. `http://en.wikipedia.org/wiki/History_of_compiler_writing`.

[72] Wikipedia. Infinite monkey theorem. `http://en.wikipedia.org/wiki/Infinite_monkey_theorem`.

[73] Wikipedia. Software testing - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Software_testing`.

[74] Yang, X., Chen, Y., Eide, E., and Regehr, J. Csmith, github repository. `https://github.com/csmith-project/csmith`.

[75] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2011), PLDI '11, ACM, pp. 283–294.

[76] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N., and Inc, G. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2009).

[77] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng. 28*, 2 (Feb. 2002), 183–200.

[78] Zhao, C., Xue, Y., Tao, Q., Guo, L., and Wang, Z. Automated test program generation for an industrial optimizing compiler. In *AST* (2009), IEEE, pp. 36–43.