Skeletal Program Enumeration for Rigorous Compiler Testing

Qirun Zhang Chengnian Sun Zhendong Su
University of California, Davis, United States
{qrzhang, cnsun, su}@ucdavis.edu

Abstract

A program can be viewed as a syntactic structure $\mathbb{P}$ (syntactic skeleton) parameterized by a collection of identifiers V (variable names). This paper introduces the *skeletal program enumeration* (SPE) problem: Given a syntactic skeleton $\mathbb{P}$ and a set of variables V, enumerate a set of programs $\mathcal{P}$ exhibiting all possible variable usage patterns within $\mathbb{P}$. It proposes an effective realization of SPE for systematic, rigorous compiler testing by leveraging three important observations: (1) Programs with different variable usage patterns exhibit diverse control- and data-dependence, and help exploit different compiler optimizations; (2) most real compiler bugs were revealed by small tests (*i.e.*, small-sized $\mathbb{P}$) — this "small-scope" observation opens up SPE for practical compiler validation; and (3) SPE is exhaustive *w.r.t.* a given syntactic skeleton and variable set, offering a level of guarantee absent from all existing compiler testing techniques.

The key challenge of SPE is how to eliminate the enormous amount of equivalent programs *w.r.t.* α-conversion. Our main technical contribution is a novel algorithm for computing the canonical (and smallest) set of all non-α-equivalent programs. To demonstrate its practical utility, we have applied the SPE technique to test C/C++ compilers using syntactic skeletons derived from their own regression test-suites. Our evaluation results are extremely encouraging. In less than six months, our approach has led to 217 confirmed GCC/Clang bug reports, 119 of which have already been fixed, and the majority are long latent despite extensive prior testing efforts. Our SPE algorithm also provides *six orders* of magnitude reduction. Moreover, in three weeks, our technique has found 29 CompCert crashing bugs and 42 bugs in two Scala optimizing compilers. These results demonstrate our SPE technique's generality and further illustrate its effectiveness.

CCS Concepts • **Software and its engineering** → **Software testing and debugging**; *Source code generation*; • **Mathematics of computing** → *Enumeration*

Keywords Program enumeration, compiler testing

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062379

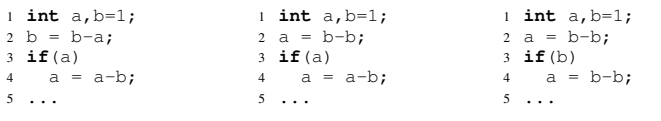
1. Introduction

Compilers are among the most fundamental programming tools for building software. A compiler bug may result in unintended program executions and lead to catastrophic consequences for safety-critical applications. It may also hamper developer productivity as it is difficult to determine whether an execution failure is caused by defects in the application or the compiler. In addition, defects in compilers may silently affect *all* programs that they compile. Therefore, improving compiler correctness is crucial.

The predominant approach to validating production compilers consists of various forms of testing. An important, challenging problem in compiler testing is *input generation*: How to effectively generate "good" test programs? Intuitively, a good test program is productive (*i.e.*, it triggers latent compiler defects) and thorough (*i.e.*, it stress tests the internal passes of a compiler). Besides manually created validation suites (*e.g.*, Plum Hall [3] and Perennial [2]), the main techniques for input program generation can be categorized as *program generation* or *program mutation*. Program generation constructs fresh test programs guided by a language's syntax and semantics. For example, Csmith is the most well-recognized random program generator for testing C compilers [9, 57]. Program mutation, on the other hand, focuses on systematically transforming existing programs. Equivalence Modulo Input (EMI) has been the most representative mutation-based approach by randomly inserting or deleting code [32, 33, 53]. Both approaches are opportunistic because the typical search space is unbounded, and they tend to favor large and complex programs.

Skeletal Program Enumeration. This paper explores a different, much less explored approach of *skeletal program enumeration* (SPE) for compiler testing. Rather than randomly generating or mutating large and complex programs, is it possible to fully exploit small programs to obtain bounded guarantees *w.r.t.* these small programs? Specifically, we view every program P as a syntactic skeleton $\mathbb{P}$ with placeholders (or holes) for a variables set V. Given small sets of $\mathbb{P}$ and V, we obtain new programs $\mathcal{P}$ by exhaustively enumerating *all* variable usage patterns to fill the holes in $\mathbb{P}$. This paper demonstrates its strong practical utility for compiler testing. Three key observations underlie our SPE realization:

- *Most compiler bugs can be exploited through small test programs.* According to a recent large-scale study on GCC and Clang's bug repositories, each reduced test case in



```
1 int a,b=1;
2 b = b-a;
3 if(a)
4   a = a-b;
5 ...
```

(a) Program P_1

```
1 int a,b=1;
2 a = b-b;
3 if(a)
4   a = a-b;
5 ...
```

(b) Program P_2

```
1 int a,b=1;
2 a = b-b;
3 if(b)
4   a = b-b;
5 ...
```

(c) Program P_3

Figure 1. Illustrative example for skeletal program enumeration, where we assume that the code snippets are parts of a function.

the bug reports contains fewer than 30 lines of code on average [54]. Moreover, in our empirical evaluation based on the c-torture test-suite from GCC-4.8.5, each function contains only *3* variables with *7* use-def sites on average.[1]

- *Different variable usage patterns trigger various compiler optimization passes.* Consider the programs based on different variable usage patterns in Figure 1. Note that the programs share the same program skeleton. In P_1, a compiler may issue a warning on the uninitialized variable `a`. In P_2, due to the constant propagation of `b = 1`, variable `a` is folded to 0 on line 3. Therefore, an optimizing compiler performs a dead code elimination of the if statement. Finally, in P_3, variable `b` is folded to 1 on line 3. An optimizing compiler then performs constant propagation for variable `a` on lines 2 and 4. Section 2 illustrates SPE for compiler testing via concrete bugs.
- *Exhaustive enumeration provides relative guarantees.* Given a small syntactic skeleton P with k variables, our approach produces input programs for compiler testing by enumerating all instances of P exhibiting different variable usage patterns. For any programming language, it is also possible to enumerate all syntactically valid token sequences (*i.e.*, the syntactic skeletons $\mathbb{P}$) up to a given bounded length. Our skeletal program enumeration establishes the first step toward realizing bounded verification of compilers.[2]

The essence of SPE is, given a skeleton $\mathbb{P}$ and a set of variables V, producing a set of programs $\mathcal{P}$ by instantiating each placeholder in the skeleton $\mathbb{P}$ with a concrete variable $v \in V$. Given a set of k variables and a $\mathbb{P}$ with n placeholders, a naïve approach produces the SPE set $\mathcal{P}$ with k^n programs. However, most of the programs in $\mathcal{P}$ are α-*equivalent*, *i.e.*, there exists an α-conversation between any two α-equivalent programs. Since α-equivalent programs always exploit the same control- and data-dependence information, it is redundant to enumerate them for most purposes and especially for compiler validation. Generating *only* and *all* non-α-equivalent programs makes SPE a unique and challenging combinatorial enumeration problem. Existing techniques for enumeration are inefficient to deal with α-equivalence in SPE (please refer to Section 4.3 for a detailed discussion).

[1] GCC's c-torture test-suite consists of (small) test programs that broke the compiler in the past.

[2] For languages that allow undefined behaviors, such as C/C++, we assume reliable oracles exist for detecting undefined behaviors (*cf.* Section 5.4).

This paper presents the first practical combinatorial approach for SPE that generates only non-α-equivalent programs in $\mathcal{P}$. To this end, we formulate SPE as a set partition problem and tackle the unique challenge of dealing with variable scoping. As an application of our SPE technique, we implement and apply it to test the development versions of GCC and Clang/LLVM, two popular open-source C/C++ compilers. In less than six months, we have found and reported 217 bugs, most of which are long latent (*e.g.*, more than two thirds of the GCC bugs affect at least three recent stable releases). About half the bugs concern C++, an extremely complex language, making our work the first successful exhaustive technique for testing compilers' C++ support. To further demonstrate its efficiency, we have applied the SPE technique to test CompCert [35] and two Scala compilers [1, 4]. Our three-week testing efforts also yield promising results.

Furthermore, to quantify the effectiveness of our enumeration scheme, we also apply both our approach and the naïve approach to GCC-4.8.5's test-suite. In particular, we use the enumerated programs to test the stable releases of GCC-4.8.5 and Clang-3.6. Besides finding 11 bugs in both compilers, more importantly, our approach achieves six orders of size reduction over the naïve enumeration approach. Approximately, our approach can process all programs in less than one month, while the naïve approach would need more than 40K years to process the same set of test programs.

Contributions. Our main contributions follow:

- We formulate the problem of skeletal program enumeration to aid compiler testing. Unlike existing approaches based on random program generation or mutation, our approach exhaustively considers all variable usage patterns for small programs;
- We propose an efficient combinatorial approach to program enumeration. In our empirical evaluation, our algorithm reduces the search space by six orders of magnitude over naïve enumeration when processing compiler test-suites; and
- We apply our SPE technique to test GCC, Clang/LLVM, CompCert and two Scala compilers. In less than six months, we have found and reported 217 bugs in GCC and Clang. In about three weeks, we have also found 29 CompCert crashing bugs, and 42 bugs in the production Scala compiler [4] and the Dotty [1] research compiler. These bugs have been actively addressed by developers. For instance, as of November 2016, among our reported GCC bugs, 68% have already been fixed, 66% are long latent and 10% are release-blocking. 25 CompCert bugs have been fixed and all 27 Dotty bugs have been confirmed.

Generality. Beyond compiler testing, skeletal program enumeration suggests a general strategy for approaching various enumeration problems. Indeed, rather than enumerating suitable structures from scratch *w.r.t.* syntax or semantics, it can be more profitable to enumerate *w.r.t.* skeletons derived from existing structures, which are arguably more interesting and lead to a more feasible process. Algorithmically, our tech-

```
1 int a = 0;
2 extern int b __attribute__ ((alias (''a'')));
3
4 int main ()
5 {
6   int *p = &a, *q = &b;
7   *p = 1;
8   *q = 2;
9
10 //return b;
11   return a; // Bug: the program exits with 1
12 }
```

Figure 2. This test program is miscompiled by multiple GCC versions from GCC 4.4 to revision 233678. This bug affected revision 104500 in September 2005, and had been latent for over ten years until we discovered it via SPE. The program is expected to return 2, but incorrectly returns 1 instead.

nique of casting SPE as the set partition problem and how to support variable scoping may be adapted to enumeration problems where such information is relevant, such as functional program enumeration, quantified formula enumeration, and other domain-specific settings.

Paper Organization. The rest of the paper is structured as follows. Section 2 motivates our work via concrete examples, and Section 3 defines the SPE problem and program α-equivalence. We present our combinatorial program enumeration algorithm in Section 4 and experimental results in Section 5. Finally, Section 6 surveys related work, and Section 7 concludes.

2. Motivating Examples

This section motivates our work using two real compiler bugs found via SPE: a wrong code bug and a crash bug. A *wrong code bug* is a compiler miscompilation, *i.e.*, the compiler silently produces a wrong executable, whose behavior is unintended and different from that of the original source program. A *crash bug* refers to the compiler crashing when processing an input program. The wrong code bug is an example latent bug, and the crash bug was classified as release-blocking by the GCC developers.

Bug 69951 : GCC Miscompilation. Figure 2 shows a test program that triggers a miscompilation in a series of GCC versions, ranging from GCC-4.4 to the latest development trunk (revision 233678). The bug affects as early as revision 104500 from September 2005, and had been in GCC even before this revision. For over ten years, from then to March 2016, when we found and reported this bug, it had slipped through various compiler testing techniques and thorough in-house testing.

This program is expected to exit with 2. The attribute annotation on line 2 declares that the variable `b` is an alias of `a`. As pointers `p` and `q` point to `a` and `b` respectively, they essentially represent the same memory region (*i.e.*, variable `a`). The last write to `a` is 2 through the pointer `q` on line 8, hence the exit code of this program should be 2. However, the buggy version of GCC optimizes the code as if `p` and `q` were not aliases, and thus the exit code of this program becomes 1 instead. The cause of this bug is that GCC did not

```
1 struct s { char c[1]; };
2 struct s a, b, c;
3 int d; int e;
4
5 void bar (void)
6 {
7 //e ? (d==0 ? b : c).c : (e==0 ? b : c).c;
8   e ? (d==0 ? b : c).c : (d==0 ? b : c).c;
9 }
```

Figure 3. This test program crashes the development trunk of GCC (revision 233377) at all optimization levels. The bug has been marked as release-blocking.

canonicalize two declarations that share the same memory address (*i.e.*, `a` and `b` in this example) into a single one, thus compromising the soundness of its alias analysis.

This test program is enumerated using a skeleton from GCC's own test-suite by replacing the original variable `b` with `a` on line 11. The program in Figure 2 is simplified for presentation purposes. The original program is slightly larger, and a naïve program enumeration approach generates 3,125 programs. In constrast, our approach only enumerates 52 non-α-equivalent programs, and exposes the bug.

Bug 69801 : GCC Internal Compiler Crash. Figure 3 shows another bug example found by SPE. The test program crashes the development trunk of GCC at all optimization levels, including -O0. The reported bug has been marked as release-blocking by the GCC developers.

The program is quite simple. Line 8 tries to access the field `c` via nested conditional expressions. This line is also the key to trigger the bug in the GCC's constant folding pass. GCC crashes when it is checking whether the second operand (`d == 0 ? b : c`) and the third operand (`d == 0 ? b : c`) are equal in the function `operand_equal_p`. This function recursively checks whether each component of the two operands are the same. When it is checking the integer constant 0 of the binary expression `d == 0`, an assertion is violated because `operand_equal_p` is instructed to use the addresses of the integer constants to test the equality, which is undefined. In the bug fix, a flag is set to instruct `operand_equal_p` to check integer equality via value comparison.

This test program is also enumerated from GCC's own test-suite. The difference between them is shown on line 7. The test program is derived by replacing `e` with `d` in the third operand of the whole conditional expression. This replacement makes the second and third operands identical, triggering the bug in the function `operand_equal_p`.

3. SPE Problem Formulation

This section formalizes skeletal program enumeration.

3.1 Problem Statement

As mentioned in Section 1, a program P comprises of two parts: a syntactic skeleton with placeholders for variables, and a set of variables. We define every usage of a variable in program P as a *hole*, and denote it as $\square$. In particular, let us consider a WHILE-style language shown in Figure 4. Figure 4(a) gives the syntax rules for the WHILE language

$a ::= x \mid n \mid a_1\ op_a\ a_2$
$b ::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2$
$S ::= x := a \mid S_1\,;S_2 \mid \texttt{while}(b)\ \texttt{do}\ S \mid \texttt{if}(b)\ \texttt{then}\ S_1\ \texttt{else}\ S_2$

(a) Syntax rules for P.

$[\![a]\!] ::= \square \mid n \mid [\![a_1]\!]\ op_a\ [\![a_2]\!]$
$[\![b]\!] ::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ [\![b]\!] \mid [\![b_1]\!]\ op_b\ [\![b_2]\!] \mid [\![a_1]\!]\ op_r\ [\![a_2]\!]$
$[\![S]\!] ::= \square := [\![a]\!] \mid [\![S_1]\!]\,;[\![S_2]\!] \mid \texttt{while}([\![b]\!])\ \texttt{do}\ [\![S]\!] \mid \texttt{if}([\![b]\!])\ \texttt{then}\ [\![S_1]\!]\ \texttt{else}\ [\![S_2]\!]$

(b) Syntax rules for transformed $\mathbb{P}$.

Figure 4. Hole transformation for the WHILE language.

(a) Program P:
$a := 10;$
$b := 1;$
$\texttt{while}(a)\ \texttt{do}$
$\quad a := a - b;$

(b) Skeleton $\mathbb{P}$:
$\square := 10;$
$\square := 1;$
$\texttt{while}(\square)\ \texttt{do}$
$\quad \square := \square - \square;$

(c) Program P_1:
$b := 10;$
$a := 1;$
$\texttt{while}(b)\ \texttt{do}$
$\quad b := b - a;$

(d) Program P_2:
$a := 10;$
$b := 1;$
$\texttt{while}(b)\ \texttt{do}$
$\quad b := a - b;$

Figure 5. Program enumeration for the WHILE language.

which has been widely used in the program analysis literature [43]. In particular, the nonterminals S, a and b denote statements, arithmetic and Boolean expressions, respectively. The WHILE language plays a pivotal role in explaining the basic ideas of our work. Note that the simple WHILE language does not have scope constraints, and thus every variable is considered global.

To obtain a program with holes, we recursively apply a hole transformation $[\![]\!]$ to the WHILE grammar. Figure 4(b) gives the transformed grammar. For any WHILE program P, we say $\mathbb{P}$ is a *skeleton* of P iff $T_{\mathbb{P}} = [\![T_P]\!]$ where $T_{\mathbb{P}}$ and T_P are the respective abstract syntax trees of $\mathbb{P}$ and P. Every hole $\square_i$ in $\mathbb{P}$ is associated with a hole variable set $\mathbb{v}_i$. The set $\mathbb{v}_i$ describes all variables that belong to the lexical scope of $\square_i$. Therefore, replacing all $\square_i$s in $\mathbb{P}$ with variables $v \in \mathbb{v}_i$ emits a syntactically valid WHILE program P'. We say $v \in \mathbb{v}_i$ *fills* $\square_i$, and P' *realizes* $\mathbb{P}$. A skeleton $\mathbb{P}$ with n holes can be represented as a *characteristic vector* $s_{\mathbb{P}} = \langle \square_1, \square_2, \ldots, \square_n \rangle$. Therefore, a program P' that realizes $\mathbb{P}$ can also be represented as a vector $s_{P'} = \langle v_1, v_2, \ldots, v_n \rangle$ such that $v_i \in \mathbb{v}_i$ fills $\square_i$ in $s_{\mathbb{P}}$ for all $i \in [1, n]$.

DEFINITION 1 (Skeletal Program Enumeration). *Given a skeleton $\mathbb{P}$ and the hole variable sets $\mathbb{v}_i$ for each $\square_i$, skeletal program enumeration (SPE) exhaustively computes a set of programs $\mathcal{P}$, such that each $P \in \mathcal{P}$ realizes $\mathbb{P}$.*

EXAMPLE 1. *Consider the example in Figure 5. Figure 5(a) shows a* WHILE *program P, and Figure 5(b) its skeleton $\mathbb{P}$ with 6 holes. Since both a and b are global variables, we have $\mathbb{v}_1 = \mathbb{v}_2 = \cdots \mathbb{v}_6 = \{a, b\}$. The program P_1 in Figure 5(c) realizes $\mathbb{P}$ with $s_{P_1} = \langle b, a, b, b, b, a \rangle$. Moreover, the program P_2 in Figure 5(d) realizes $\mathbb{P}$ with $s_{P_2} = \langle a, b, b, b, a, b \rangle$. Therefore, in the program enumeration of this example, we have $P, P_1, P_2 \in \mathcal{P}$.*

For a skeleton $\mathbb{P}$ with n holes, program enumeration essentially generates the n-ary Cartesian product over sets $\mathbb{v}_1, \mathbb{v}_2, \ldots, \mathbb{v}_n$. As a result, the search space for generating all possible solutions in $\mathcal{P}$ is $\prod_{i=1}^{n} |\mathbb{v}_i|$, which is clearly exponential in terms of n. For instance, the skeleton $\mathbb{P}$ in Figure 5 realizes $2^6 = 64$ programs, *i.e.*, $|\mathcal{P}| = 64$.

3.2 Program α-Equivalence

The naïve approach to SPE produces an overwhelming amount of programs, where most of the enumerated instances are equivalent *w.r.t.* α-conversion. The α-equivalent programs always exhibit the same control- and data-dependence information. In this paper, we describe a combinatorial approach to exhaustively enumerate only non-α-equivalent programs in $\mathcal{P}$. Section 3.2.1 formally defines α-equivalence using WHILE programs, and Section 3.2.2 discusses α-equivalence of practical C programs with scope information.

3.2.1 α-Equivalent Programs

Let us consider two WHILE programs P and P_1 in Figure 5. The characteristic vectors are $s_P = \langle a, b, a, a, a, b \rangle$ and $s_{P_1} = \langle b, a, b, b, b, a \rangle$, respectively. As mentioned in Example 1, both P and P_1 belong to the SPE solution $\mathcal{P}$ in Figure 5. Particularly, we can transform P to P_1 by replacing all occurrences of variables a and b in P with b and a, respectively. The idea behind the transformation is quite similar to the concept of α-conversion in lambda calculus. It is clear in Figure 5 that P exhibits the same control- and data-dependence information as P_1. Consequently, if P is already enumerated, there is no need to consider P_1.

Let V be the set of all variables in a WHILE program P with n holes. Since the WHILE language does not have lexical scopes, the set V is the same as the hole variable set $\mathbb{v}_i$ for each hole $\square_i$, *i.e.*, $V = \mathbb{v}_1 = \cdots = \mathbb{v}_i$, for all $i \in [1, n]$. Let $\alpha : V \to V$ be a permutation of set V. Given P, V and α, we define an α-*renaming* such that it replaces each occurrence of variable v in P with $\alpha(v)$ for all $v \in V$. The α-renaming transforms a program P to P', denoted as $P \xrightarrow{\alpha} P'$. For example, in Figure 5, we have $V = \{a, b\}$, $\alpha = \left(\begin{smallmatrix} a & b \\ b & a \end{smallmatrix}\right)$, and $P \xrightarrow{\alpha} P_1$. Formally, we define α-equivalence below.

DEFINITION 2 (Program α-Equivalence). *Two programs P_1 and P_2 are α-equivalent, denoted as $P_1 \cong P_2$, iff:*

(i). Both P_1 and P_2 realize the same $\mathbb{P}$; and

(ii). There exists an α-renaming such that $P_1 \xrightarrow{\alpha} P_2$.

EXAMPLE 2. *Consider Figure 5 again. P and P_1 are α-equivalent. However, P and P_2 are non-α-equivalent programs since their characteristic vectors are $s_P = \langle a, b, a, a, a, b \rangle$ and $s_{P_2} = \langle a, b, b, b, a, b \rangle$, respectively. It is obvious that there exists no α-renaming between them.*

For α-equivalent programs P_1 and P_2, the α-renaming maps the output value of any variable a in P_1 to the variable $\alpha(a)$ in P_2 for any fixed inputs. Therefore, the α-equivalent WHILE programs are semantically equivalent. As a result, we can safely eliminate those α-equivalent programs in program enumeration, and thus reduce the solution set.

3.2.2 α-Equivalence with Scope Information

The WHILE language in Figure 5 does not take lexical scoping into account. The lexical scope information can reduce the size of the SPE set $\mathcal{P}$, even for the naïve approach. In the remaining sections of this paper, we discuss using C programs. However, the conceptual idea is general and can be adapted to any imperative language.

```
int main(){
  int a=1, b=0;
  if(a){
    int c=3, d=5;
    b = c + d;
  }
  printf("%d", a);
  printf("%d", b);
  return 0;
}
```

(a) Program P.

```
int main(){
  int □=1, □=0;
  if(□){
    int □=3, □=5;
    □ = □ + □;
  }
  printf("%d", □);
  printf("%d", □);
  return 0;
}
```

(b) Skeleton $\mathbb{P}$

```
int main(){
  int c=1, b=0;
  if(c){
    int a=3, d=5;
    b = a + d;
  }
  printf("%d", c);
  printf("%d", b);
  return 0;
}
```

(c) Program P_1

```
int main(){
  int b=1, a=0;
  if(b){
    int d=3, c=5;
    a = d + c;
  }
  printf("%d", b);
  printf("%d", a);
  return 0;
}
```

(d) Program P_2

Figure 6. α-equivalent C programs.

Let us consider the C programs in Figure 6. Given a program P in Figure 6(a), we can construct a skeleton $\mathbb{P}$ shown in Figure 6(b) and a variable set $\mathbb{v}_i = \{a, b, c, d\}$ for all $i \in [1, 10]$. The construction treats all variables as if they were global variables. According to Definition 1, SPE computes $4^{10} = 1,048,576$ programs. However, the variables a and b in P are global variables, while the variables c and d belong to the local scope of the if statement. Therefore, the variable a can be used to fill any hole that belongs to c, but *not vice versa*. With the scope information, a naïve approach only needs to enumerate $2^5 \cdot 4^5 = 32,768$ programs in $\mathcal{P}$.

To cope with lexical scopes in C programs, we extend α-renaming such that it only maps variables of the same scope. We define the extended renaming map as a *compact* α-renaming. Moreover, when transforming a C program P to $\mathbb{P}$, we also associate each hole $\square_i$ and its hole variable set $\mathbb{v}_i$ in $\mathbb{P}$ with the corresponding scope information in P. Therefore, a hole $\square_i$ can only be filled with the variables available at the current scope. The variable types can also be handled by extending the compact α-renaming in a similar way. Finally, it is clear that the compact α-renaming still preserves semantic equivalence.

THEOREM 1. *Given a compact α-renaming, and two C programs P_1 and P_2, $(P_1 \cong P_2) \implies (P_1 \equiv P_2)$.*

EXAMPLE 3. *In Figure 6, P, P_1 and P_2 are α-equivalent programs. In particular, we have $P \xrightarrow{\alpha_1} P_1$ using an α-renaming $\alpha_1 = \left(\begin{smallmatrix} a & b & c & d \\ c & b & a & d \end{smallmatrix}\right)$, and $P \xrightarrow{\alpha_2} P_2$ using a compact α-renaming $\alpha_2 = \left(\begin{smallmatrix} a & b & c & d \\ b & a & d & c \end{smallmatrix}\right)$. They are all semantically equivalent, generating the same output "18". Moreover, for the compact α-renaming, we have $\mathbb{v}_i = \{a, b\}$ and $\mathbb{v}_j = \{a, b, c, d\}$, where $i \in \{1, 2, 3, 9, 10\}$ and $j \in \{4, 5, 6, 7, 8\}$. As metioned above, the SPE* w.r.t. *compact α-renamings computes 32 times fewer programs.*

4. SPE Algorithm

This section presents our combinatorial program enumeration approach. Our approach only enumerates non-α-equivalent programs. Section 4.1 describes the main idea based on programs without scope information. Section 4.2 extends the idea to handle scope information. Section 4.3 provides further relevant discussions.

4.1 Basic Idea

In the SPE problem, the inputs are a syntactic skeleton $\mathbb{P}$ and a set of hole variables $\mathbb{v}_i$. Let us revisit the example in Figure 5. The skeleton $\mathbb{P}$ in Figure 5(b) has 6 holes. Each hole is associated with the same hole variable set $\mathbb{v}_i = \{a, b\}$ for all $i \in [1, 6]$. Therefore, there are 2^6 ways to fill in these holes using a naïve approach.

As discussed in Section 3.2, the α-equivalent programs are redundant for SPE. Having a representative program for all its α-equivalent variants helps reduce the size of the SPE solution $\mathcal{P}$. Therefore, in our approach, we seek to compute an SPE set $\mathcal{P}'$ of all non-α-equivalent programs, *i.e.*, $P_1 \ncong P_2$ for all distinct $P_1, P_2 \in \mathcal{P}'$. To realize this, we formulate SPE as a set partition problem. In particular, we view the n holes in $\mathbb{P}$ as a set $H = \{1, \ldots, n\}$ of n elements. Filling a hole with a variable $v \in \mathbb{v}_i$ can also be considered as partitioning an element $h \in H$ into a subset that corresponds to v. For example, the skeleton $\mathbb{P}$ in Figure 5(b) with 6 holes can be represented as set $H = \{1, \ldots, 6\}$. Let variable a be the first subset and b the second subset to partition. The characteristic vector $s_{P_1} = \langle b, a, b, b, b, a \rangle$ of P_1 in Figure 5(c) can be represented as a set partition $\{\{1, 3, 4, 5\}, \{2, 6\}\}$ of set H_{P_1}, where the first subset represents the holes filled with b and the second subset the holes filled with a. Due to the α-equivalence property mentioned in Section 3.2.1, the variable names are of no importance. Therefore, the partition $\{\{1, 3, 4, 5\}, \{2, 6\}\}$ is equivalent to $\{\{2, 6\}, \{1, 3, 4, 5\}\}$. On the other hand, the partitions are sensitive to the elements in set H such that partition $\{\{1, 3, 4, 5\}, \{2, 6\}\}$ is different from $\{\{2, 3, 4, 5\}, \{1, 6\}\}$.

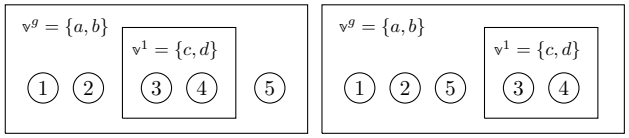
As a result, given a skeleton $\mathbb{P}$ with n elements and a hole variable set $\mathbb{v}_i$, where $|\mathbb{v}_i| = k$ for all $i \in [1, n]$, the SPE problem can be reduced to a combinatorial problem.

> Enumerate the ways to partition a set of n elements into k subsets.

EXAMPLE 4. *Consider the skeleton $\mathbb{P}$ in Figure 6. The characteristic set of P is $\langle a, b, a, c, d, b, c, d, a, b \rangle$. The corresponding set partition is $\{\{1, 3, 9\}, \{2, 6, 10\}, \{4, 7\}, \{5, 8\}\}$. P, P_1 and P_2 are α-equivalent, therefore, they have the same set partition.*

4.1.1 Number of Partitions

In combinatorics, the set partition problems are also known as the *twelvefold way*, since there are twelve ways to classify all related problems [29]. When the set elements are labeled and the subsets unlabeled, the number of ways to partition a set of n elements into k *non-empty* subsets is denoted by the *Stirling number of the second kind* [29], denoted as $\left\{{n \atop k}\right\}$ for $k \leqslant n$. For $k > n$, we let $\left\{{n \atop k}\right\} = \left\{{n \atop n}\right\}$, *i.e.*, we consider at most n partitions. For our SPE problem, let S denote the



(a) Set partition illustration. (b) Corresponding normal form.

Figure 7. Set partition illustration and its normal form.

number of all partitions, and we have

$$S = \sum_{i=1}^{k} \left\{ {n \atop i} \right\} \quad (1)$$

For fixed value of k, one asymptotic estimation of the Stirling number of the second kind is $\left\{ {n \atop k} \right\} \sim \frac{k^n}{k!}$ [44, §26.8]. Therefore, we estimate the SPE solution set as follows:

$$S \sim \frac{1^n}{1!} + \frac{2^n}{2!} + \cdots, + \frac{k^n}{k!} = O(\frac{kn^k}{k!}) = O(\frac{n^k}{(k-1)!}) \quad (2)$$

The overall complexity of our combinatorial approach is still exponential. However, it reduces the entire solution set by a notable constant factor of $(k-1)!$. In practice, it improves the feasibility of skeletal program enumeration.

4.1.2 Partition Enumeration

We adopt the standard approach to enumerate all set partitions in lexicographic order [29, 30]. The conventional approach to encode a unique set partition is using a *restricted growth string* [29, 30]. For a set of n elements, a restricted growth string $a_1 a_2 \ldots a_n$ of length n satisfies the following:

$$a_1 = 0 \text{ and } a_{i+1} \leqslant 1 + \max(a_1, \ldots, a_i) \text{ if } i \in [1, n)$$

The intuitive meaning of a restricted growth string is that, each element h in H is partitioned to a subset numbered by a_i, where i represents the index of h in H. Moreover, suppose that m elements in H have already been partitioned, if the new element $m+1$ belongs to a new partition, we always assign the smallest available number to a_{m+1}.

EXAMPLE 5. *Consider the skeleton $\mathbb{P}$ in Figure 5. The characteristic set of P is $\langle a, b, a, a, a, b \rangle$. The corresponding restricted growth string is "010001". Since P_1 and P are α-equivalent, their strings are the same. For P_2, we have $s_{P_2} = \langle a, b, b, b, a, b \rangle$ and the corresponding string is "011101".*

4.2 Taming Scopes

The most significant challenge of skeletal program enumeration is to handle variable scopes. Taking the scope information into consideration, each hole in the syntactic skeleton $\mathbb{P}$ can be filled with different sets of variables. As a result, computing the non-α-equivalent programs becomes more difficult. The corresponding set partition problem of skeletal program enumeration with scope information is unique and has not been studied in the literature.

Giving a skeleton $\mathbb{P}$ and hole variable sets $\mathbb{v}_i$ with scope information, we depict the set partition problem using a figure with circle and squares, where each labeled circle denotes the corresponding $\square_i \in \mathbb{P}$ and the squares represent the scope information. In particular, according to the compact α-renaming described in Section 3.2.2, each hole (circle) can only be filled with the variables from a valid scope (square). We use the notations $\mathbb{v}^g$ and $\mathbb{v}^l$ to represent the sets of global variables and the variables declared in scope l, respectively. Consider the example in Figure 7(a). We have $\mathbb{v}^g = \{a, b\}$, and $\mathbb{v}^1 = \{c, d\}$ for the first local scope. It is clear from the figure that hole 2 can be filled with $v \in \mathbb{v}_2 = \mathbb{v}^g$ whereas hole 4 can be filled with $v \in \mathbb{v}_4 = \mathbb{v}^g \cup \mathbb{v}^1$. It is also clear that a naïve approach generates $2^3 \cdot 4^2$ solutions.

4.2.1 Set Partition for Skeletal Program Enumeration

Unlike the standard set partition problem discussed in Section 4.1, the new enumeration problem essentially considers the set partition of a set H with constraints on each element $h \in H$. This section formalizes this new partition problem.

Consider a program skeleton $\mathbb{P}$ with n holes, and t scopes. Each hole $\square_i \in \mathbb{P}$ can be either global or local. The global hole $\square_i^g$ can be filled with only global variables, *i.e.*, $\mathbb{v}_i = \mathbb{v}^g$ whereas the local hole $\square_i^l$ can be filled with additional local variables defined in scope l, *i.e.*, $\mathbb{v}_i = \mathbb{v}^g \cup \mathbb{v}^l$ and $l \in [1, t]$. The set partition problem for SPE can be described as follows:

> Given a set H of n elements, and pre-defined sets $\mathbb{v}_i \subseteq \{1, \ldots, k\}$ for all $i \in [1, n]$. Each element $i \in H$ can be partitioned to a subset labeled by $v \in \mathbb{v}_i$ and $\mathbb{v}_1 \cup \mathbb{v}_2 \cup \ldots \cup \mathbb{v}_i = \{1, \ldots, k\}$. Enumerate the ways to partition H into k subsets.

4.2.2 Partitions with Scopes

A straightforward approach to compute partitions with scopes is computing a local set partition solution S_l for each scope, respectively. Then, obtain the final solution S by computing the Cartesian product over all local solutions S_l together with the solution of global holes S_g, *i.e.*, $S = S_g \times S_1 \times \cdots \times S_t$. However, the set partitions obtained are not the global solution among all elements. For example, consider the holes 3 and 4 in Figure 7(a), where we have $\mathbb{v}_3 = \mathbb{v}_4 = \mathbb{v}^g \cup \mathbb{v}^1$. The local solution for them contains two partitions: $\{\{3, 4\}\}$ and $\{\{3\}, \{4\}\}$. Since the number of holes is smaller than the number of hole variables, we pick two variables and let $\mathbb{v}_3 = \mathbb{v}_4 = \{b, c\}$. Locally, the variable names are unimportant in set partition problems. Therefore, for partition $\{\{3\}, \{4\}\}$, filling variables $\square_3 \leftarrow b$ and $\square_4 \leftarrow c$ is equivalent to $\square_3 \leftarrow c$ and $\square_4 \leftarrow b$. On the other hand, combining the two solutions with the remaining holes filled with $\langle a, a, b \rangle$ obtains two solutions $\langle a, a, b, c, b \rangle$ and $\langle a, a, c, b, b \rangle$. Clearly, they are two unique solutions since they have different restrict growth strings "00121" and "00122", respectively.

To obtain the global solution, the key idea in our partition algorithm is to choose some local holes by considering all combinations of the local holes. Then, the chosen ones are promoted to be global. Finally, we obtain the solution by computing the Cartesian product of the global holes and the remaining local holes.

Procedure PartitionScope(S_L, G, l_i).

```
1  u ← |l_i| and v ← |𝕧_i|
2  foreach k ∈ [1, u] do
3      l ← COMBINATIONS(l_i, k)
4      G ← G ∪ l
5      l̄ ← l_i \ l
6      foreach j ∈ [1, v] do
7          S_{l_i} ← PARTITIONS(l̄, j)
8          S'_L ← S_L
9          S_L ← S_L × S_{l_i}
10         if i is not the last scope then
11             ParititionScope (S_L, G, l_{(i+1)})
12         else
13             S_G ← PARTITIONS'(G, |𝕧^g|)
14             S_f ← S_f ∪ {S_G × S_L}
15         S_L ← S'_L
16     G ← G \ l
```

Algorithm 1: Skeletal program enumeration algorithm.

Input : A program skeleton $\mathbb{P}$ and hole variable set $\mathbb{v}_i$;
Output : a set of programs $\mathcal{P}$.

```
1  foreach function f ∈ ℙ do
2      Normalize function f
3      S'_f ← PARTITIONS(H_f, |𝕧_f|)
4      S_f ← ∅ and S_L ← ∅
5      ParititionScope (S_L, G_f, l_1)
6      S_f ← S_f ∪ S'_f
7      S ← S × S_f
8  foreach characteristic vector s ∈ S do
9      Generating a program P using ℙ and s
```

Handling one scope. Procedure `PartitionScope`(S_L, G, l_i) describes the major steps for handling scope i, where S_L represents the set of all local solutions, G denotes the set of the global holes and l_i the set of the local holes of scope i. The routine COMBINATIONS(Q, k) returns $\binom{|Q|}{k}$ different ways of selecting k elements from the set Q. The routine PARTITIONS(Q, k) partitions the set Q into k subsets in $\sum_{i=1}^{k} \left\{ {|Q| \atop k} \right\}$ ways. Moreover, the routine PARTITIONS$'(Q, k)$ partitions the set Q into k non-empty subsets in $\left\{ {|Q| \atop k} \right\}$ ways. `PartitionScope` handles a scope i as follows:

- *Promoting k holes from scope i.* Line 1 obtains the cardinalities of sets l_i and $\mathbb{v}_i$. On line 3, we choose k holes from l_i and promote them as global holes G on line 4. The set of remaining holes is denoted as $\bar{l}$ on line 5.
- *Computing local solution for scope i.* Lines 7-9 computes the local set partition S_{l_i} of scope i and combines it with the current local solution S_L. If i is not the last scope, it recursively handles the next scope $i + 1$ on line 11.
- *Obtaining the final solution.* If i is the last scope, it computes the solution S_G of global holes G (line 13), combines it with the current local solution S_L and appends it to the final solution S_f (line 14). On line 15 and 16, the information on G and S_L is restored for subsequent recursive calls to Procedure `PartitionScope`.

Program enumeration algorithm. Algorithm 1 describes our combinatorial SPE algorithm. For each function f in skeleton $\mathbb{P}$, we consider its characteristic vector $s_f = \langle 1, \ldots, n \rangle$. Within a function f, the global variable set $\mathbb{v}_f$ contains the global variables in $\mathbb{P}$, function parameters and function-wise variables. Moreover, the set of global holes, denoted as H_f, contains the holes that can be filled with $v \in \mathbb{v}_f$. For t local scopes, we rearrange the vector to be of the *normal form* $\langle \square^g, .., \square^g, \square^1, .., \square^1, \ldots, \square^t, .., \square^t \rangle$, *i.e.*, we pull all global holes to the front and arrange local holes in order. For example, Figure 7(b) gives the normal form of the holes in Figure 7(a). Let G_f and l_i be the sets of the global holes in f and the local holes in scope i, respectively. Algorithm 1 computes the partitions for function f as follows. It normalizes f (line 2) and computes a partial solution for f (line 3) without taking scopes into consideration. It then computes a solution S_f by recursively processing each scope on lines 4-5. Moreover, the global solution of function f is obtained by combining both S_f and S'_f on line 6. The global solution of $\mathbb{P}$ is obtained by computing the Cartesian product of each function on line 7. Finally, we enumerate the programs according to the solutions in S.

EXAMPLE 6. *Consider the normal form in Figure 7(b). Algorithm 1 computes the set partitions as follows.* Computing S'_f*: There are* $\left\{ {5 \atop 2} \right\} + \left\{ {5 \atop 1} \right\} = 16$ *partitions;* Promoting either 3 or 4*: There are* $\left\{ {4 \atop 2} \right\} \times \left\{ {1 \atop 1} \right\} = 7$ *partitions for each hole;* Promoting both 3 and 4*: There are* $\left\{ {3 \atop 2} \right\} \times (\left\{ {2 \atop 2} \right\} + \left\{ {2 \atop 1} \right\}) = 6$ *partitions;* Final solution*: SPE algorithm computes* $(16 + 2 \cdot 7 + 6) = 36$ *partitions. However, the naïve approach computes* $(2^3 \cdot 4^2) = 128$ *partitions.*

4.3 Discussions

Granularity of enumeration. Algorithm 1 obtains the SPE solution S of a skeleton $\mathbb{P}$ by computing the Cartesian product *w.r.t.* each local solution S_f of function f. We say that Algorithm 1 computes the *intra-procedural* enumeration. Since each function can also be considered as a local scope *w.r.t.* a program, the intra-procedural enumeration approximates the global solution, where we call the global solution as the *inter-procedural* enumeration. Algorithm 1 can be easily extended to obtain the inter-procedural enumeration. The key extension is to replace the **foreach** loop on lines 1-7 with a call to Procedure `PartitionScope`, where l_1 represents the first function scope instead. To handle additional scopes, one can processes all scopes in a bottom-up fashion *w.r.t.* the scope hierarchy. It is a practical design choice of enabling intra- or inter-procedural enumerations. The intra-procedural enumeration — though being an approximation — enumerates fewer variants of a single test program P than the inter-procedural counterpart. Thus, given a fixed budget on the total number of enumerated variants, the intra-procedural enumeration is able to process more test programs. It would be interesting to investigate different enumeration granularities and find the most cost-effective enumeration scheme for practical use.

Enumeration vs. counting. We have discussed the SPE set partition problem, and proposed an enumeration algorithm. An interesting open problem is to investigate the corresponding counting counterpart of the enumeration problem in Section 4.2.1. Specifically, fixing i and k in an SPE problem, the counting problem is to determine the number of non-α-equivalent programs for a syntactic skeleton $\mathbb{P}$ with n holes. In Section 4.1.1, we discussed the counting problem of the

SPE problem without scope information, based on the traditional analysis of set partition problems [29, 30]. However, developing an asymptotic estimation of the SPE problem defined in Section 4.2.1 is nontrivial, as the analytics with the variable set $\mathbb{v}_i$ constraints becomes more complex. A promising direction may be counting the enumeration set using the technique based on e-restricted growth functions [38, 39].

Other enumeration techniques. Algorithm 1 solves the SPE problem based on the combinatorial algorithms for generating set partitions and combinations. In the literature, there has been an extensive body of work that exhaustively generates input structures for software testing. This line of work typically specifies the *invariant* property and enumerates the structures declaratively [7, 19, 28, 50], imperatively [12, 31, 49, 56] or in a hybrid fashion [20, 48].

Unfortunately, these approaches are inefficient to automatically leverage the invariant for the SPE problem. The key challenge of adopting the existing enumeration techniques is to encode the invariant. Specifically, the declarative enumeration techniques specify the invariant and typically use generate-and-test approaches. Our combinatorial SPE algorithm maintains the invariant of the non-α-equivalence. Let P_1 and P_2 be two programs of the SPE set $\mathcal{P}$. The invariant is: $P_1 \ncong P_2$ for all distinct $P_1, P_2 \in \mathcal{P}$. Therefore, to generate $|\mathcal{P}|$ non-α-equivalent programs, it needs to test $\prod_{i=1}^{n} |\mathbb{v}_i|$ programs as a naïve SPE solution discussed in Section 3.1. In addition, the imperative enumeration frameworks are capable of enumerating only valid inputs *w.r.t.* the invariant. However, our SPE algorithm solves a combinatorial problem rather than generating combinatorial structures (*e.g.*, red-black trees, graphs and algebraic representations). Even though it might be feasible to encode the SPE algorithm using the primitive enumerators in the imperative enumeration frameworks, the realization is strictly less efficient than directly applying our combinatorial SPE algorithm in the first place.

Another relevant problem is enumerating lambda terms exhaustively up to a given size [21, 36, 55]. Most of the work enumerates lambda terms using the standard "nameless" de Bruijn representation [14]. These approaches consider a rather different enumeration problem as the lambda terms have distinct syntactic structures and semantics. Specifically, the essential enumeration problem concerns with various unary-binary tree structures [22, 36, 55]. However, in our set partition setting, there is no dependence among set elements.

Algorithm correctness. Algorithm 1 invokes procedure `PartitionScope` to compute the scoped set partitions for each function f. We briefly discuss the correctness of procedure `PartitionScope`. Our algorithm handles functions at different granularities. In Algorithm 1, the input function f is in the normal form. Recall that each hole $\square_i$ in the skeleton $\mathbb{P}$ corresponds to an element $i \in H$. In the normal form, the elements can been filled with both global ($\mathbb{v}^g$) and local ($\mathbb{v}^l$) variables. We define the *configuration* of the normal form to be a map $c : H \rightarrow \{g, l\}$ for all variables $i \in H$. It is then sufficient and necessary to show that: (1) procedure `PartitionScope` computes unique non-α-equivalent partitions for each configuration; and (2) procedure `PartitionScope` finds all configurations in function f.

- *Part (1).* The configure c maps $i \in H$ to either l and g, and it leads to two cases. In the first case, all elements are global. Therefore, the SPE problem becomes the standard set partition problem. Procedure `PartitionScope` calls procedure PARTITIONS to compute the set partitions of size j. In the second case, some elements i representing local holes $\square^l$ are mapped to l. In this case, the partition problems of the global and local elements become independent. Procedure `PartitionScope` computes respectively the set partitions for elements that representing both $\square^g$s and $\square^l$s, and obtains the global solution by computing their Cartesian product.
- *Part (2).* Procedure `PartitionScope` calls procedure COMBINATIONS to find all configurations of function f's normal form by exhaustively selecting the combinators of local holes.

5. Evaluation

To evaluate the effectiveness of skeletal program enumeration, we conduct two sets of experiments. In the first experiment, we enumerate skeletons derived from GCC-4.8.5's test-suite, and test two stable compiler releases, GCC-4.8.5 and Clang-3.6.1. We aim to demonstrate the benefits of combinatorial SPE. In the second experiment, we use a set of small programs to test the trunk versions of GCC and Clang, as well as CompCert and two Scala compilers, to demonstrate the bug-hunting capabilities of SPE.

5.1 Experimental Setup

Our implementation contains two components, *i.e.*, skeleton generation and program enumeration. The skeleton generation component recursively traverses the ASTs to obtain the scope and type information for each variable, and build a skeleton $\mathbb{P}$ for each test program P. The program enumeration component realizes the enumeration algorithm described in Algorithm 1. We compute the intra-procedural enumeration as mentioned in Section 4.3.

Given a set of programs $\mathcal{P}$, we directly feed those programs to the compilers under testing. For GCC and Clang, we use two optimization levels (*i.e.*, -O0 and -O3) and two machine modes (*i.e.*, 32- and 64-bits) for finding crashes. For wrong code bugs, we investigate the program P with CompCert's reference interpreter [35] and additional manual efforts to ensure that it is free of undefined behaviors. All experiments were conducted on a server and a desktop running Ubuntu-14.04. The server has Intel Xeon X7542 CPUs and 128GB RAM, while the desktop has an Intel i7-4770 CPU and 16GB RAM.

5.2 Experiments on Stable Releases

In our first experiment, we evaluate the SPE technique on stable releases of two popular C compilers, specifically GCC-4.8.5 and Clang-3.6.1. We choose GCC-4.8.5 since it is the default C compiler in the long term support version of Ubuntu

Approach	Original Test-Suite			Enumerated Test-Suite		
	Total Size	Avg. Size	#Files	Total Size	Avg. Size	#Files
Naive	5.24×10^{163}	2.49×10^{159}	20,978	1,310,943,547,383	69,538,698.7	18,852
Our	1.48×10^{79}	7.05×10^{74}	20,978	2,050,671	108.8	18,852

Table 1. Evaluation results on size reduction. The "total size" column shows the total numbers of enumerated programs, and the "avg. size" the average numbers of the enumerated programs for each test program. The size of the enumerated test-suite is related to a threshold discussed in Section 5.2.1.

(14.04), and Clang-3.6.1 was released about the same time as the chosen GCC.

We implemented both our combinatorial program enumeration described in Algorithm 1 and a naïve enumeration algorithm mentioned in Section 3.1. We apply the two implementations on the default test-suite which has been shipped with GCC-4.8.5. Most of the test programs belong to the c-torture suite, which contains the code snippets that have historically broken previous releases.[3] According to the GCC's release criteria, any released version must pass the test-suite distributed in the source code. We are particularly interested in understanding the following research questions:

- What is the size reduction achieved by our SPE approach?
- Given the fact that the test-suite contains many programs once broke previous GCCs, what are the characteristics of these programs?
- Can SPE find bugs in the stable GCC and Clang releases using their own regression test-suite?

5.2.1 Enumeration Size Reduction

The GCC-4.8.5 test-suite contains about 21K C files. Table 1 decries the size reduction results of applying our combinatorial SPE algorithm. For the original test-suite, our combinatorial SPE approach reduces the entire size by 94 orders of magnitude. However, it is clear from the table that SPE solution set is still too large to be applied for compiler testing in practice. As a result, we set a 10K threshold such that we ignore the test programs which have more than 10K variants using our combinatorial SPE algorithm. The 10K threshold is chosen *w.r.t.* the characteristics of the test-suite (Table 2), *i.e.*, $|Vars|^{|Holes|} = 3.46^{7.34} \approx 10K$). We then compare the solution spaces based on the remaining programs. From the last three columns in Table 1, we can see that the number of test files is decreased to 19K. Using the 10K threshold, we can still retain 90% of the original test programs. On those files, our SPE algorithm achieves six orders of size reduction over the naïve approach. Specifically, for each test program, the solution of our SPE approach contains merely 109 files on average. In practical settings, suppose that we could process each program in one second, it takes less than one month to handle all enumerated programs. However, for the naïve approach, it takes about 40K years to process the same test programs. Finally, Figure 8 describes size reduction in terms of different program enumeration sets $\mathcal{P}$.

[3] `https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.

Test-Suite	#Holes	#Scopes	#Funcs	#Types	#Vars
Original	7.34	2.77	1.85	1.38	3.46
Enumerated	3.84	1.85	1.50	1.29	1.60

Table 2. Characteristics of the GCC-4.8.5 test-suite. The first four columns display the average counts of holes, scopes, functions and variable types in each file, respectively. The last column displays the variable counts for each hole.

internal compiler error: in assign_by_spills, at lra-assigns.c:1281
error in backend: Do not know how to split the result of this operator!
error in backend: Invalid register name global variable.
error in backend: Access past stack top!
Assertion 'MRI->getVRegDef(reg) && "Register use before def!"' failed.
Assertion 'Num < NumOperands && "Invalid child # of SDNode!"' failed.

Table 3. Crash signatures of bugs found in GCC-4.8.5 and Clang-3.6.1 using the GCC-4.8.5 test-suite.

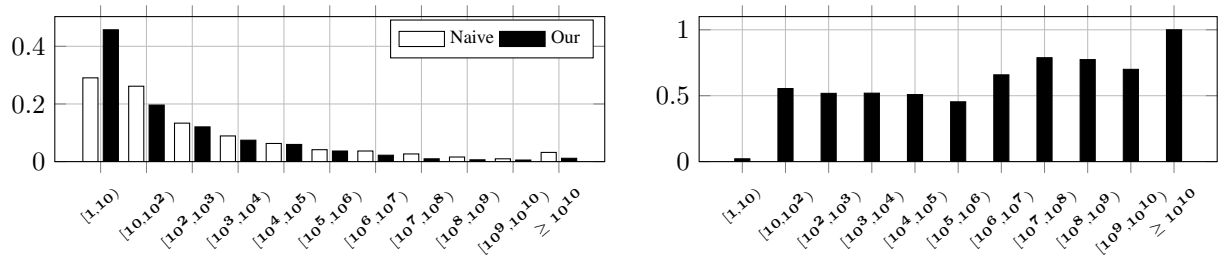
5.2.2 Test-Suite Characteristics

Table 2 gives an overview of the test programs in GCC-4.8.5 test-suite. It also describes the programs used in our evaluation based on the aforementioned 10K threshold. It is interesting to observe that most of the programs are quite small even though most of them have triggered bugs in previous versions of GCC. Indeed, this observation has motivated our current program enumeration work. The programs used in our evaluation are smaller due to the 10K threshold setting *w.r.t.* our combinatorial enumeration algorithm. Recall that these programs represent 90% of the programs in the original test-suite. It clearly demonstrates that it is feasible to apply combinatorial SPE on practical test-suites.

5.2.3 Benefits of Skeletal Program Enumeration

Hunting bugs. We apply our enumeration algorithm to test GCC-4.8.5 and Clang-3.6 by enumerating the skeletons from the GCC-4.8.5 test-suite. Our SPE technique have found 1 and 10 crash bugs in GCC and Clang, respectively. It is perhaps interesting to note that we are able to find GCC bugs by enumerating its own test-suite even if the release criteria force it to pass the original test-suite. In this evaluation, we only focus on crash bugs since wrong code bugs usually require compiler developers' confirmation (mostly due to possible undefined behaviors in test programs). For crash bugs, compiler messages clearly indicates their occurrence. Table 3 gives the signatures of some crash bugs found in this evaluation. We can see that most of the bugs are in the backend and optimization passes.

Improving coverage. As described in Section 1, one of our insights is that SPE can help trigger more internal compiler passes. In order to validate the claim, we compare our SPE technique against the seminal work Orion of program muta-



(a) Distribution of the numbers of variants. For example, the first pair of vertical bars shows that 29% of the test programs have fewer than 10 variants enumerated using the naïve approach. The percentage increases to 46% using our approach.

(b) Distribution of the ratios of reduced variants. For example, the second bar shows that our approach has eliminated 55% of the programs with $n \in [10, 10^2)$ variants compared with the naïve approach on average.

Figure 8. Overview of the size reduction. In both figures, the x-axis lists the size ranges of enumeration sets $\mathcal{P}$. In particular, $\mathcal{P}$ is described using the number of variants enumerated for each test program P. The y-axis represents the percentage.

(a) GCC-4.8.5 coverage improvements.

(b) Clang-3.6 coverage improvements.

Figure 9. Coverage improvements over the baseline tests. PM-X represent improvements achieved by program mutation (the Orion tool) which deletes X statements, and SPE represents improvements achieved using our SPE algorithm.

tion [32]. We choose Orion since it only considers statement deletion. Therefore, the overall search spaces for both approaches are bounded. We randomly select 100 test programs from the test-suite to run both approaches. Figure 9 gives the empirical results. The selected test programs achieve 41% function coverage and 32% line coverage for GCC, respectively. For Clang, they achieve 20% function coverage and 17% line coverage, respectively. Our SPE approach brings approximately 5% coverage improvement for GCC and 2.4% coverage improvement for Clang, respectively. On the other hand, Orion provides less than 1% coverage improvement. This comparison also demonstrates the advantage of applying our SPE technique on small programs.

It is also worth noting that Orion has found 1 and 3 bugs in Clang-3.6 and GCC-4.8.5, respectively, using the same test-suite. The three GCC bugs are unique as they are different from what we have found. This evaluation has also provided practical evidence that program enumeration and program mutation offer complementary benefits.

5.3 Experiments on Development Versions

We apply our combinatorial program enumeration for finding bugs in the trunk versions of GCC and Clang. We select a set of small C programs from the unit test-suite of many open-source projects, such as CompCert [35], Frama-C, the Rose compiler and KCC [17]. In particular, most of our test C programs are from the test-suite in the trunk version of GCC. The test programs share similar characteristics with those described in Section 5.2. We began our testing process in early January. In less than six months, our technique has discovered 217 GCC and Clang bugs. To date, more than half of them have been fixed.

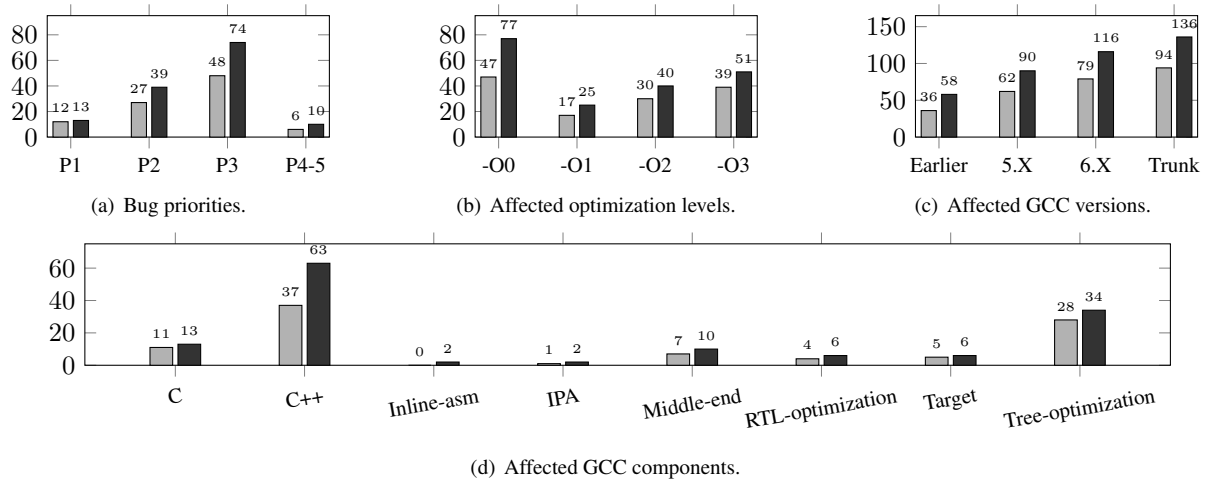
To demonstrate SPE's generality, we have also applied it to test the CompCert verified C compiler, and two optimizing Scala compilers, *i.e.*, the production Scala compiler and the Dotty research compiler. In about three weeks, we have reported 29 CompCert crashing bugs and 42 bugs in the two Scala compilers. The developers have appreciated and promptly addressed our reports — 25 CompCert bugs have already been fixed (all have been confirmed), and 27 Dotty bugs have been confirmed. We started testing the two Scala compilers recently in late October. Among the Dotty bugs, 9 have been fixed so far. Until now, there are only five high-priority bugs in total in the Dotty code repository, and our SPE technique has discovered four of them. The rest of this section focuses discussing the GCC and Clang/LLVM bugs.

5.3.1 Overall Results

Table 4 gives an overview of the bugs that we have found during the testing course. We have reported 217 bugs in total. Developers have confirmed almost all of our reported bugs. Moreover, more than half of them have already been fixed within the six-month period. Some of our reported bugs are quite complex. For example, two bugs have been reopened by developers for further inspection. Although we ensure that our reported bugs have different symptoms, it is sometimes inevitable that we have occasionally reported duplicates as it is quite difficult for us to track the root cause for each bug. However, less than 5% of the bugs are duplicates. Two of our reported GCC bugs have been marked as invalid. In

Compiler	Summary					Classification		
	Reported	Fixed	Duplicate	Invalid	Reopened	Crash	Wrong code	Performance
GCC	136	93	10	2	1	127	6	3
Clang	81	26	3	1	1	79	2	0

Table 4. Overview of bugs reported for trunk versions of GCC and Clang in six months.



(a) Bug priorities.

(b) Affected optimization levels.

(c) Affected GCC versions.

(d) Affected GCC components.

Figure 10. Characteristics of GCC trunk bugs. The darker bars denote the numbers of reported bugs and the lighter bars the numbers of fixed bugs.

particular, one of them is about multiple inheritance and casting in C++, and the other is a C program that contains undefined behavior concerned with strict aliasing. We further discuss the undefined behavior issue in Section 5.4.

Table 4 also gives the classification of the bugs. Most of the bugs cause compiler crashes. As mentioned in Section 5.1, we leverage the SPE technique to find both frontend and optimization bugs. Among all GCC crash bugs, 56% of them trigger frontend crashes, where most of them are related to the C++ frontend. On the other hand, 44% lead to crashes in the optimization passes. Moreover, we have discovered 8 bugs related to miscompilation. As mentioned in Section 2, one of them has been around for more than ten years. Finally, three of the bugs are related to compilation performance. We describe one such bug in Appendix A.

5.3.2 Bug Characteristics

We discuss the characteristics of our reported GCC bugs. It is worth mentioning that we have made more effort testing GCC since GCC developers are relatively more responsive. In particular, GCC developers not only have fixed 68% of our bugs but also provide more feedback. Figure 10 characterizes the 136 reported GCC trunk bugs. Specifically, Figure 10(a) shows the importance of the reported bugs. P3 is the default priority in GCC's bugzilla system. About two thirds of the bugs fall into this category. About 10% of them are release-blocking (P1). Developers have to fix all P1 bugs in order to release a future version. Figure 10(b) shows that our reported bugs cover all optimization levels. Specifically, our approach has found more -O3 bugs than the -O2 and -O1 bugs. This demonstrates that the SPE technique is able to cover deep compiler optimization passes. Figure 10(c) shows the affected GCC versions. We can see that 85% of the bugs affect the latest 6 release. Moreover, 66% of the bugs affect at least three stable GCC 5 releases. Perhaps the most interesting to note is that 43% of the bugs affect earlier GCC versions from at least one year ago. It demonstrates that our techniques can find long latent bugs. Figure 10(d) shows the diversity of our reported bugs. Over half of our bugs are C++ frontend bugs. The second category of most frequent bugs concern the tree-optimization component. The results suggest that our SPE technique is useful for testing various compiler components.

Our technique has discovered a large number of diverse bugs in a relatively short period of time. One unique, noteworthy aspect of our work is the large number of reported bugs in the compilers' C++ support, making it the first successful exhaustive technique to provide this capability. C++ is an active, enormously complex language and has a growing set of features — it is very challenging to develop practical C++ program generators. Note that we have many more bugs to triage, reduce and report, but have been reporting bugs in a steady fashion so as not to overwhelm the developers. The results highlight the novelty and benefits of our approach.

5.3.3 Case Studies on Sample Bugs

We select and discuss four reported GCC and Clang bugs. Figure 11 describes the corresponding test programs with bug classifications and status. Eight additional bug samples may be found in Appendix A.

Figure 11(a). This test program exposes a long latent bug of GCC that affects all versions since GCC-4.4, which was released four years ago. The bug is in the C++ frontend of GCC, and manifests when GCC computes the path of the base

```
class A {
  virtual void foo()
  { }
};

class B : public A, A
{ };

B b1, &b2 = b1;
A a = b2;
```

(a) G++ crash 70202 (fixed)

```
char a; short b;
void fn1() {
  if (b)
    ;
  else {
    int c[1] = {0};
    l1: ;
  }
  if (a) goto l1;
}
```

(b) GCC crash 69740 (reopened)

```
int a;
double b, *c;

void fn1(int p1) {
  for (;; p1--) {
    a = p1;
    for (; p1 >= a; a--)
      b = c[p1];
  }
}
```

(c) Clang crash 26973 (fixed)

```
int main() {
  int *p = 0;
trick:
  if (p)
    return *p;
  int x = 0;
  p = &x;
  goto trick;
  return 0;
}
```

(d) Clang wrong code 26994 (confirmed)

Figure 11. Sample test programs that trigger bugs of GCC and Clang.

classes for the class `B`. The GCC developers have confirmed this bug and are investigating its root cause.

Figure 11(b). This is a crash bug of the GCC trunk (6.0 revision 233242). It manifests when GCC compiles the test program at -O2 and above. The `goto` statement in the program introduces an irreducible loop, and GCC incorrectly handles the backend and consequently triggers the assertion `verify_loop_structure` to fail. This reported bug had been fixed once, and later reopened by the GCC developers. Note that this program is enumerated from the test program in GCC bug report PR68841 .

Figure 11(c). This test program crashes the trunk (3.9 revision 263641) of Clang at -O1 and above. This bug is a regression, and had been latent for eleven months until we discovered it. The culprit revision incorrectly passes a wrong parameter to infer the loop invariant, and consequently corrupts the emitted LLVM bitcode and causes an assertion violation in the compiler backend.

Figure 11(d). The program is miscompiled by the Clang trunk (3.9.0 revision 263789). The expected exit code is 0. However the miscompiled executable returns 1 instead. The root cause is that the Clang frontend deems that the lifetime of the variable `x` ends after the control flow jumps to the label `trick`, which is incorrect. Consequently the write to variable `x` (*i.e.*, `int x = 0`) was eliminated, and the miscompiled executable just returns a memory cell with uninitialized data. This bug is also a regression affecting the stable release of Clang 3.7 and all later versions.

5.4 Toward Bounded Compiler Verification

As mentioned in Section 1, our approach is general and establishes the first step toward practical techniques for proving the absence of compiler bugs for any programming language. For C/C++ compilers, the SPE technique itself does not guarantee that the generated programs are free of undefined behaviors. Specifically, for the incorrect return value of the program described in Figure 2, our technique cannot determine directly whether it is a compiler miscompilation or a false alarm due to possible undefined behavior. We rely on the heuristics discussed in Section 5.1 and manual inspection to confirm the bug. The test program was generated by SPE, which we believe can help prove the absence of miscompilations in C/C++ compilers. Our SPE technique has indeed found several wrong code bugs in both GCC and Clang, but much fewer than crash bugs. This section briefly discusses practical considerations in finding wrong code bugs with skeletal program enumeration.

The most significant challenge is to avoid enumerating programs with undefined behaviors. In both program generation and mutation, one can design different heuristics to avoid producing "bad" programs. For instance, when performing statement insertions in Athena [33], one can carefully choose the candidate statements to avoid introducing undefined behavior such as uninitialized variables or out-of-bound array accesses. Moreover, a key contribution of Csmith [57] is to ensure that its generated programs are, most likely, free of undefined behavior. However, in our SPE work, what heuristics to use is less obvious since we consider all variable usage patterns. On the other hand, SPE is deterministic and exhaustive rather than opportunistic. As a result, applying static analysis on each enumerated program would not be too expensive. We leave as interesting future work to explore static analysis techniques or efficient enumeration schemes to avoid undefined behaviors in the enumerated programs.

Besides avoiding undefined behaviors, it is also challenging to detect undefined behaviors given a set of enumerated programs. This is perhaps more general since the issue itself has been an interesting, actively researched problem. The reference interpreter in CompCert [35], for example, offer tremendous help in detecting "bad" programs. However, since CompCert only works on a subset of C, it may not handle many practically useful features such as inline assembly, attributes and compiler-specific extensions. It also defines certain undefined behaviors, such as signed integer overflows. Tools such as Clang's undefined behavior sanitizers are also useful, but incomplete. As a result, we resort to manual inspection to rule out the remaining "bad" programs, which hinders productivity. Reliable tools for detecting undefined behaviors would be extremely helpful.

6. Related Work

Csmith is the most popular random program generator for testing C compilers [9, 57]. Compared with the testsuite used in our study, Csmith generates large and complex programs. Csmith is a highly influential project. Over the years, it has helped find a few hundred bugs in both GCC and Clang/LLVM. Based on Csmith, the CLsmith work of Lidbury *et al.* focuses on testing OpenCL compilers [37]. Orange3 is a random program generator that tests arithmetic optimizations in C compilers [41]. CCG is another random C program generator which finds crash bugs in early versions

of GCC and Clang [5]. Epiphron is a randomized technique to detect defects in compiler warning diagnostics in GCC and Clang [52]. For functional languages, there has also been an extensive body of work on exhaustive or random test-case generators for compiler testing [10, 11, 16, 18, 45, 49]. Boujarwah and Saleh conduct a thorough survey on generation techniques for compiler testing [6].

A recent work of Le *et al.* proposes the idea of testing compilers using the equivalence modulo inputs (EMI) [32] concept. Practical testing tools based on EMI mutate programs by inserting and deleting statements in unexecuted branches. In particular, Orion randomly deletes program statements in dead regions [32]. Athena adopts the Markov Chain Monte Carlo (MCMC) method to guide both statement insertions and deletions to obtain more interesting test programs [33]. Hermes inserts code fragments to live regions [53]. Moreover, Proteus applies the EMI technique to test link-time optimizers [34]. The frameworks based on EMI are quite efficient for compiler testing. They have revealed many bugs in both GCC and Clang/LLVM. Most of them are deep wrong code bugs. Besides testing C compilers, LangFuzz mutates syntactically correct JavaScript programs using failing code fragments [24]. It has discovered many vulnerabilities in the Mozilla JavaScript interpreter. Finally, the well-known mutation testing technique mutates a program to evaluate the quality of its testsuite [15, 23].

To guarantee the correctness of compilers, the two most notable developments are, perhaps, translation validation [42, 46] and verified compilers [35]. Besides verification, compiler testing is another important practical approach. For testing C compilers, all of the program generation, program mutation and our SPE techniques realize the same idea of differential testing [40]. The three approaches complement each other. Specifically, for program enumeration, we consider small test programs. Our technique exhaustively exploits all variable combinations. On the other hand, the other two approaches tend to produce large and complex programs in a randomized fashion. The buggy programs discovered using these techniques could be processed using CompCert's reference interpreter to identify undefined behaviors [35]. To file high-quality bug reports, test programs should also be reduced first, using tools like C-Reduce [47] and Berkeley Delta [13].

Our work is also related to bounded-exhaustive testing, which concerns the enumeration of all possible input structures up to a given size [51]. Two popular techniques are declarative enumeration and imperative enumeration. In particular, declarative approaches leverage any given invariant to search for valid inputs [7, 19, 28, 50], and the imperative approaches directly construct the inputs based on more prescriptive specifications [12, 31, 49, 56]. In program synthesis, there have been studies on inductive functional programming systems for exhaustively synthesizing small programs [8, 25–27]. The essential enumeration techniques, categorized as analytical or generate-and-test approaches, share similar conceptual ideas. As mentioned in Section 4.3, existing enumeration techniques are expensive and impractical for the combinatoral enumeration problem that this work considers.

7. Conclusion

This paper has introduced skeletal program enumeration (SPE) for compiler testing and developed a practical combinatorial solution. Our approach significantly reduces the number of enumerated programs. For an empirical demonstration of its utility, we have applied it to test two production C/C++ compilers, CompCert C comipler and two Scala compilers. Our results are extremely promising. For instance, in less than six months, our approach has helped discover more than 200 bugs in GCC and Clang. More than half of our reported bugs have already been fixed, the majority are long latent, and a significant fraction are classified as critical, release blocking.

Our SPE strategy and techniques are general and may be applied in other enumeration settings. This work also demonstrates the practical potential of program enumeration, and opens up opportunities toward bounded compiler verification.

Acknowledgments

We would like to thank our shepherd, Mayur Naik, and the anonymous PLDI reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. This research was supported in part by the United States National Science Foundation (NSF) Grants 1319187, 1528133, and 1618158, and by a Google Faculty Research Award. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

A. Additional Sample Bugs

We briefly discuss eight additional sample bugs found by SPE to show its generality and the diverse bugs that it can detect.

Figure 12(a). This bug is long latent and intriguing as it causes different symptoms for multiple GCC versions. It affects optimization levels -O1 and above. When compiling it, GCC 4.6 and 4.7 hang, whereas 4.8 to trunk crash. GCC incorrectly computes the address for exception handling, which later causes an assertion violation in the middle end.

Figure 12(b). This test program is miscompiled by the GCC trunk (6.0 revision 234026) at -O3. It is derived by enumerating a test case in GCC's testsuite. The expression `c=c+u[a+1335*a]` on line 8 is obtained by replacing `b` in the original expression `c=c+u[a+1335*b]` with `a`. Then this replacement triggers a regression in the loop vectorizer pass.

Figure 12(c). This test program crashes the trunk (7.0 revision 237059) of GCC at -Os and above. The code overrides the placement new operator of C++ on line 6. A replacement new operator creates an object in a given memory region. In the main function, this overridden new operator is called to create an object of type `C` at the address of the local variable `a` (*i.e.*, '`new (&a) C`'). However, because `C` and `a` have different types, GCC translates the code into an ill-formed intermediate representation (*i.e.*, GIMPLE code), which does not pass the GIMPLE verification pass.

Figure 12(d). This program triggers a bug in the name mangling module of Clang for the Itanium C++ ABI. On line 6,

```
1 void foo()
2 {
3   unsigned long l;
4   void *p = 0;
5
6   __builtin_unwind_init ();
7
8   l = 0;
9
10  __builtin_eh_return (l, p);
11 }
```

(a) GCC crash/performance bug 67619 (fixed)

```
1 double u[1782225];
2 int a, b, d, e;
3 static void foo(int *p1) {
4   double c = 0.0;
5   for (; a < 1335; a++) {
6     b = 0;
7     for (; b < 1335; b++)
8       c = c + u[a + 1335 * a];
9     u[1336 * a] *= 2;
10  }
11  *p1 = c;
12 }
13 int main() {...}
```

(b) GCC wrong code bug 70138 (fixed)

```
1 struct C {
2   C() {}
3   int i;
4 };
5
6 void *operator new(size_t, void *p2)
7 { return p2; }
8
9 int main() {
10  int a;
11  new (&a) C;
12  return 0;
13 }
```

(c) G++ crash bug 71405 (fixed)

```
1 enum Color {
2   R, G, B
3 };
4
5 template < typename T >
6 void test(T, __underlying_type (T))
7 {}
8
9 int main() {
10  Color c = R;
11  test (c, c);
12  return 0;
13 }
```

(d) Clang++ crash bug 28045 (fixed)

```
1 union U u = { 0 };
```

(e) CompCert crash bug 125 (fixed)

```
1 object Main extends App {
2   case class Foo(field: Option[String])
3   val x: PartialFunction[Foo, Int] = {
4     c => c.field match {
5       case Some(s) => 42
6     }
7   }
8 }
```

(f) Dotty crash bug 1637 (fixed)

```
1 void foo (struct A a)
2 {
3   a++;
4 }
```

(g) CompCert crash bug 121 (fixed)

```
1 class Bar {
2   def f (x : { def g }) {}
3   f (new Foo { def g })
4 }
```

(h) Scala crash bug 10015 (open)

Figure 12. Additional sample bugs.

the template function `test` takes as input two parameters of the types: a generic type `T` and the underlying type of `T`. When Clang was trying to mangle the function name of the call on line 11, the bug (*i.e.*, the type trait `__underlying_type` was improperly handled) led the compilation to unreachable code, thus failing an assertion.

Figure 12(e). This test program triggers a crashing bug in CompCert's frontend. Before the initialization, the parser does not check whether the type is incomplete, which triggers an assertion failure in CompCert.

Figure 12(f). This test program crashes the Dotty compiler — a next generation compiler for Scala. It triggers an assertion in the Dotty typer. The bug has been fixed and marked as high priority in Dotty's GitHub repository. As of March 2017, there are five high-priority bugs in the Dotty code repository, and SPE discovered four.

Figure 12(g). This test program crashes CompCert. Function `foo`'s parameter has a structure type `A`, whose definition is unavailable in this translation unit. CompCert did not reject the program early, thus leading to an "Unbound struct A" assertion failure in the subsequent compilation of the program.

Figure 12(h). This test program crashes the 2.12 stable release of the Scala compiler. Specifically, it triggers an assertion failure in Scala's type checker. The test program is enumerated from the regression test-suite in the Scala release.

References

[1] Dotty Compiler. `http://dotty.epfl.ch/`.

[2] Perennial, Inc. Perennial C Compiler Validation Suite. `http://www.peren.com/pages/cvsa_set.htm`.

[3] Plum Hall, Inc. The Plum Hall Validation Suite for C. `http://www.plumhall.com/stec.html`.

[4] Scala Compiler. `http://www.scala-lang.org/`.

[5] A. Balestrat. CCG. `https://github.com/Mrktn/ccg`.

[6] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39(9):617–625, 1997.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[8] F. Briggs and M. O'Neill. Functional genetic programming and exhaustive program search with combinator expressions. *KES Journal*, 12(1):47–68, 2008.

[9] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *PLDI*, pages 197–208, 2013.

[10] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[11] K. Claessen, J. Duregård, and M. H. Palka. Generating constrained random data with uniform distribution. *J. Funct. Program.*, 25, 2015.

[12] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *FSE*, pages 185–194, 2007.

[13] S. M. Daniel S. Wilkerson and S. Goldsmith. Berkeley Delta. `http://delta.stage.tigris.org/`.

[14] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae*, volume 75, pages 381–392, 1972.

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[16] J. Duregård, P. Jansson, and M. Wang. Feat: functional enumeration of algebraic types. In *Haskell*, pages 61–72, 2012.

[17] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[18] B. Fetscher, K. Claessen, M. H. Palka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *ESOP*, pages 383–405, 2015.

[19] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias. TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.

[20] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *ICSE*, pages 225–234, 2010.

[21] K. Grygiel and P. Lescanne. Counting and generating lambda terms. *J. Funct. Program.*, 23(5):594–628, 2013.

[22] K. Grygiel and P. Lescanne. Counting and generating terms in the binary lambda calculus. *J. Funct. Program.*, 25, 2015.

[23] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, 3(4):279–290, 1977.

[24] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *USENIX Security*, pages 445–458, 2012.

[25] S. Katayama. Systematic search for lambda expressions. In *TFP*, pages 111–126, 2005.

[26] S. Katayama. Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening. In *Proceedings of the 10th Pacific Rim International Conference on Artificial Intelligence*, pages 199–210, 2008.

[27] S. Katayama. An analytical inductive functional programming system that avoids unintended programs. In *PEPM*, pages 43–52, 2012.

[28] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4): 403–434, 2004.

[29] D. E. Knuth. *The art of computer programming. Vol. 4A., Combinatorial algorithms. Part 1*. Addison-Wesley, 2011.

[30] D. L. Kreher and D. R. Stinson. *Combinatorial algorithms: generation, enumeration, and search*. CRC Press, London, New York, 1999.

[31] I. Kuraj, V. Kuncak, and D. Jackson. Programming with enumerable sets of structures. In *OOPSLA*, pages 37–56, 2015.

[32] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, page 25, 2014.

[33] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*, pages 386–399, 2015.

[34] V. Le, C. Sun, and Z. Su. Randomized stress-testing of link-time optimizers. In *ISSTA*, pages 327–337, 2015.

[35] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[36] P. Lescanne. On counting untyped lambda terms. *Theor. Comput. Sci.*, 474:80–97, 2013.

[37] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *PLDI*, pages 65–76, 2015.

[38] T. Mansour and G. Nassar. Gray codes, loopless algorithm and partitions. *Journal of Mathematical Modelling and Algorithms*, 7(3):291–310, 2008.

[39] T. Mansour, G. Nassar, and V. Vajnovszki. Loop-free Gray code algorithm for the e-restricted growth functions. *Information Processing Letters*, 111(11):541–544, 2011.

[40] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[41] E. Nagai, A. Hashimoto, and N. Ishiura. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Trans. System LSI Design Methodology*, 7:91–100, 2014.

[42] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.

[43] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[44] F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, editors. *NIST Handbook of Mathematical Functions*. Cambridge University Press, New York, NY, 2010.

[45] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *AST*, pages 91–97, 2011.

[46] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166, 1998.

[47] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.

[48] N. Rosner, V. S. Bengolea, P. Ponzio, S. A. Khalek, N. Aguirre, M. F. Frias, and S. Khurshid. Bounded exhaustive test input generation from hybrid invariants. In *OOPSLA*, pages 655–674, 2014.

[49] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell*, pages 37–48, 2008.

[50] V. Senni and F. Fioravanti. Generation of test data structures using constraint logic programming. In *TAP*, pages 115–131, 2012.

[51] K. J. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA*, pages 133–142, 2004.

[52] C. Sun, V. Le, and Z. Su. Finding and analyzing compiler warning defects. In *ICSE*, pages 203–213, 2016.

[53] C. Sun, V. Le, and Z. Su. Finding compiler bugs via live code mutation. In *OOPSLA*, pages 849–863, 2016.

[54] C. Sun, V. Le, Q. Zhang, and Z. Su. Toward understanding compiler bugs in GCC and LLVM. In *ISSTA*, pages 294–305, 2016.

[55] P. Tarau. On type-directed generation of lambda terms. In *ICLP (Technical Communications)*, 2015.

[56] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107, 2004.

[57] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.