Boosting Compiler Testing by Injecting Real-world Code

SHAOHUA LI, ETH Zurich, Switzerland
THEODOROS THEODORIDIS, ETH Zurich, Switzerland
ZHENDONG SU, ETH Zurich, Switzerland

We introduce a novel approach for testing optimizing compilers with code from real-world applications. The main idea is to construct well-formed programs by fusing multiple code snippets from various real-world projects. The key insight is backed by the fact that the large volume of real-world code exercises rich syntactical and semantic language features, which current engineering-intensive approaches like random program generators are hard to fully support. To construct well-formed programs from real-world code, our approach works by (1) extracting real-world code at the granularity of function, (2) injecting function calls into seed programs, and (3) leveraging dynamic execution information to maintain the semantics and build complex data dependencies between injected functions and the seed program.

We implement our idea in a tool, Creal, to test C compilers. In a nine-month testing period, we have reported 132 bugs to GCC and LLVM, two of the most popular and well-tested C compilers. At the time of writing, 121 of them have been confirmed as unknown bugs, and 97 of them have been fixed. Most of these bugs were miscompilations, and many were recognized as long-latent and critical. Our evaluation results evidently demonstrate the significant advantage of using real-world code to stress-test compilers. We believe this idea will benefit the general compiler testing direction and will be directly applicable to other compilers.

1 INTRODUCTION

Compilers are among the most fundamental and critical components in the software ecosystem. They should not crash and are expected to faithfully translate valid input programs. However, compiler bugs can cause crashes or even worse, miscompilations. Decades of research and engineering efforts have been put into improving compilers' reliability [Chen et al. 2020]. Random compiler testing has proven to be effective in discovering compiler bugs. One noticeable contribution is Csmith [Yang et al. 2011], a pioneering C program generator. By generating random programs, Csmith has helped discover many bugs in C compilers. A more recent generator YARPGen [Livinskii et al. 2020] and the follow-up YARPGen v.2 [Livinskii et al. 2023], featuring a set of generation policies, target at scalar and loop optimizations and have successfully identified many bugs missed by previous generators. In addition to generator-based tools, mutation-based compiler testing techniques are also effective in finding compiler bugs. The most representative approach is *equivalence modulo inputs* (EMI) [Le et al. 2014]. EMI techniques [Le et al. 2015; Sun et al. 2016] mutate a program by removing or inserting random code while maintaining the input/output behaviors of the program. This line of research has achieved a surprising result: discovering a considerable number of additional bugs in mature and well-established compilers, such as GCC and LLVM.

Limitations. All aforementioned generation- and mutation-based approaches introduce a fundamental limitation: They are unable to produce programs with features beyond the underlying rules. These approaches rely on *predefined syntactical and semantic rules* in constructing programs. For example, to avoid undefined behavior, Csmith resorts to heavy-handed safe wrappers, which limits

In Submission, 2024, XX

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```c
1   src = data + offsetp;
2   do {
3       b = *src++;
4       ret |= ((b & 0x7f) << shift);
5       (*offsetp)++;
6       shift += 7;
7   } while ((b & 0x80) != 0);
```

(a) A loop from the FreeBSD project.

```c
for (p_1 = 0; p_1 <= 6; p_1 -= 8) {
    int32_t l_6 = 0x2E2B9C6;
    l_3 ^= (p_1 > (l_5[0] ^ l_6));
} // Csmith
for(i0=var1;i0 < arr2[0]%var3; i0+=1){
    var4 ^=  arr5[i6] << (var3+1);
} // YarpGen v.2
```

(b) Loops taken from Csmith and YARPGen v.2.

Fig. 1. Loops from real-world project, Csmith, and YARPGen v.2.

the expressiveness of the generated programs [Even-Mendoza et al. 2021]; to guarantee termination, Csmith only generates constant-bounded loops, meaning that all loop conditions are constant; EMI-based mutators [Le et al. 2015; Sun et al. 2016] augment generator-produced programs with random code and thus inherit the limitations. Due to these constraints, experience shows that all random program generators will eventually saturate and find very few bugs [Amalfitano et al. 2015]. Extending the scope of current approaches by integrating more rules is theoretically feasible but is challenging and labor-intensive in practice.

Our idea. This paper proposes using code from real-world projects for compiler testing. The core idea is first to extract code snippets from real-world projects and then inject them into a seed program to enrich its code features. Our key observation is that real-world code exercises a diverse range of syntactical and semantic code features, which are often hard for random program generators or mutators to support. Figure 1a shows a loop from FreeBSD [1], a UNIX-like OS written in C. This code contains an unbounded `while` loop, where the loop index "b" is modifiable at both lines 3 and 5. This code helped us find a latent miscompilation bug in LLVM. Figure 1b shows two common loops from Csmith and YARPGen v.2, respectively. The first loop is from Csmith. It is constant-bounded, which is true for all Csmith-produced loops. The second loop is from YARPGen v.2 and has more complex loop indexes such as `i0` and `arr2`. However, unlike Figure 1a, all these variables are not modifiable in the loop body. Generators need to capture code semantics at generation time to generate well-formed programs, which is hard for complex loops.

One may think that ***why can't we use real-world code directly for compiler testing?*** First, real-world code is unlikely to trigger observable compiler issues directly. Marcozzi *et al.* [Marcozzi et al. 2019] studied the impact of miscompilation bugs with 10 million lines of C/C++ code from 309 Debian packages. They run standard test suites from these packages to identify possible miscompilation bugs. Among dozens of studied miscompilation bugs, only one test failure was observed during a five-month experiment. Such inefficiency is clearly not suitable for stress-testing compilers. Second, real-world code often contains bugs, *e.g.*, undefined behavior. Input programs for compiler testing need to be well-formed. Otherwise, even if a failure, such as incorrect output, is observed, we cannot differentiate whether or not it is caused by undefined behavior or compiler bugs. Numerous efforts [Wang et al. 2013; Zhu et al. 2022] in academia and industry have been made to detect and remove undefined behavior from real-world code, yet they still remain unsolved. Last, reducing a project into a simple enough test case suitable for a bug report is challenging. Non-trivial open-source projects typically have a large code base. Reduction is necessary for compiler developers to diagnose and fix bugs. EMI [Le et al. 2014] reported that although they found inconsistent outputs from real-world code, they were not able to reduce them.

[1] https://github.com/freebsd/freebsd-src/blob/main/contrib/elftoolchain/libdwarf/libdwarf_rw.c

Our approach. To use real-world code, we begin by extracting well-formed functions from real-world projects. Next, we inject function calls into seed programs to establish vigorous connections between real-world functions and the seed program. Our code injection technique ensures that all produced programs are well-formed. There are three main benefits of using real-world code at the function level: (1) most compiler optimizations work at the function level [GCC 2023; LLVM 2023b], ensuring real-world functions can already exercise rich features, (2) it is relatively painless to guarantee the validity of extracted functions in terms of undefined behavior, and (3) it requires moderate changes to the seed program, making it uncomplicated to maintain the validity of the resulting program. An alternative technique is to inject code chunks directly into seed programs. However, this requires extensive modifications to both the extracted real-world code and the seed programs, which can easily result in invalid programs. Nonetheless, we consider this orthogonal approach to be an interesting future work. Our evaluation will show that using real-world functions as building blocks can effectively explore various code features.

We implement our idea in a tool named Creal. Since our approach is data-hungry, we need a diverse range of functions from real-world projects to fuel Creal. We develop a function extractor, with which we construct a function database containing more than 51,000 real-world functions from 146 popular open-source projects, such as Git, Linux Kernel, OpenSSL, to name a few. With this database, we evaluate Creal by stress-testing the latest versions of C compilers, namely GCC and LLVM. In a nine-month testing period, our tool identified 132 compiler bugs. Of those, 121 were confirmed and 97 were fixed by the compiler developers. It is worth noting that most of the bugs were miscompilations, which are the most harmful and difficult-to-detect compiler bugs.

In conclusion, we make the following contributions:

- We propose injecting real-world code into seed programs to create diverse, well-formed programs for testing compilers.
- We develop Creal to implement our idea by injecting real-world functions into seed programs. To fuel Creal, we build a function database containing over 51,000 real-world functions, and
- We conduct a nine-month extensive evaluation of Creal to demonstrate its effectiveness — 132 unique bugs are discovered in widely used production C compilers: GCC and LLVM.

We believe our idea of injecting real-world code into synthetic programs offers an exciting and promising technical direction for testing C compiler implementations and beyond.

2 ILLUSTRATIVE EXAMPLES

We use a concrete GCC miscompilation bug to illustrate (1) what features in the injected real-world function trigger the compiler bug, (2) why current generators and mutators cannot trigger this bug, and (3) how Creal injects a real-world function into the seed program. Since Csmith [Yang et al. 2011] is by far the most powerful and widely used generator for producing generic C programs [Chen et al. 2020; Sun et al. 2016], in this paper, we use Csmith as the seed program generator. To facilitate our presentation, we only show reduced and simplified programs.

Figure 2c shows a program generated by Creal that triggered a long-latent GCC miscompilation bug. This program is produced by injecting the real-world function `print_dec` shown in Figure 2a into the seed program shown in Figure 2b.

➤ Q1: What features in the real-world function are essential to trigger the bug?
The function “`print_dec`” is extracted from *the FreeBSD project* [2]. Its definition can be found in Figure 2a. This function converts a decimal integer “`value`” into a string and returns its number of digits as `i`. Now, let us focus on the program in Figure 2c. The function call “`print_dec(h, c+74, d+48)`”

[2] https://github.com/freebsd/freebsd-src/blob/main/contrib/unbound/compat/snprintf.c

```
 1 // Input: [*, 75, 48]   => Output: 2
 2 int print_dec(
 3  char* buf, int max, unsigned int value){
 4   int i = 0;
 5   if (value == 0) {
 6     if (max > 0) {
 7       buf[0] = '0';
 8       i = 1;
 9     }
10   } else while(value && i < max) {
11     buf[i++] = '0' + value % 10;
12     value /= 10;
13   }
14   return i;
15 }
```

(a) A real-world function.

```
 1 int a, b;
 2 int main() {
 3   int c = 1, d = 0;
 4   long j[2];
 5   for (a = 0; a < 2; a++) {
 6     {
 7     /* Profile: a={0,1}, b={0,1}, c={1}, d={0}*/
 8       b = d + a;
 9     }
10     j[b] = 1;
11   }
12   printf("%d\n", j[0]);
13 }
```

(b) The seed program.

```
 1 int a, b;
 2
 3 /* A function extracted from the FreeBSD project.*/
 4 int print_dec(
 5  char* buf, int max, unsigned int value){
 6   int i = 0;
 7   if (value == 0) {
 8     if (max > 0) {
 9       buf[0] = '0';
10      i = 1;
11     }
12   } else while(value && i < max) {
13     buf[i++] = '0' + value % 10;
14     value /= 10;
15   }
16   return i;
17 }
18
19 int main() {
20   int c = 1, d = 0;
21   long j[2];
22   for (a = 0; a < 2; a++) {
23     {
24       char h[2] = {0, 0};
25       // replace ``d'' with call to print_dec
26       b =  print_dec(h, c+74, d+48)-2  + a;
27     }
28     j[b] = 1;
29   }
30   printf("%d\n", j[0]);
31 }
```

(c) Our generated new program.

Fig. 2. The program in (c) triggered a latent miscompilation bug in GCC. It is generated by replacing the variable d from line 8 in (b) with a carefully designed function call to (a). The synthesized function call is highlighted in gray in (c).

in line 26 should always return 2 because c+74=75, which is a 2-digit number. Thus, b = a. The correct output from this program should be 1 as j[0] is set to 1 in line 28. The executable compiled by GCC at -O3, however, outputs 13368. The root cause is that when GCC unrolls the "while" loop in lines 12-15, the complex control flow leads to incorrect memory partition for arrays j and h. Consequently, values in j are polluted by h and become incorrect. The complex while loop in "print_dec" is essential to trigger the bug — removing any expression such as "i<max" (line 12), "value%10" (line 13), or "buf[i++]" (line 13) from the loop cannot trigger the bug.

➤ Q2: Why current generator- and mutator-produced programs cannot trigger this bug? As discussed above, the complex "while" loop is the key to triggering the bug. However, as has discussed before, generators like Csmith can only generate constant-bounded "for" loops such as

```
for (i = 0; i < 2; i++) {           for (i = 0; i != 2; i-=v) {
  ...                                  ...
}                                    }
```

Csmith is conservative in generating loops to guarantee termination and avoid undefined behavior. All the loops are bounded by constant values, and loop indexes are only modified in the "increment" parts of for loops. The while loop from the real-world function in Figure 2a, on the other hand, has more features: (1) it has two loop indexes, (2) both loop indexes are modified in the loop body, and (3) the loop is not constant-bounded. The Csmith implementation does not support these features. Consequently, mutators relying on generators cannot support these features, either.

➤ Q3: How does our approach generate the new program?

Our approach aims to inject real-world function calls into a seed program. Real-world functions need to be available in the first place. We design a function extractor to extract and preprocess functions from real-world projects. Assume that "print_dec()" is the only function we collected. After collection, we generate a driver function to capture valid input/output pairs for the function. One possible driver function for print_dec() is

```
int main () {
  char arr[2];
  int ret = print_dec(arr, 75, 48);
  print("%d\n", ret);
}
```

Executing the driver function can tell us that when calling with "print_dec(arr, 75, 48)" where arr is pointing to a sufficiently large memory, the return value of "print_dec()" is "2". This I/O information will be added along with the static function information into our function database. For example, the first line in Figure 2a indicates one input/output pair for the function. We apply extensive analysis such as sanitizers [Team 2023] to guarantee that there is no undefined behavior when calling the function with the cached inputs. More details will be included in Section 4.2.

The resulting program should be well-formed. To this end, we construct an *equivalence modulo inputs* (EMI) variant when injecting calls, which means the new program has an input/output behavior identical to the seed program. For example, the injected call "print_dec(h, c+74, d+48)-2" is always evaluated to 0 at run-time, which is identical to the replaced variable "d". This equivalence guarantees that the resulting new program is also well-formed as long as both the seed program and the injected function are well-formed. Specifically, given the seed program in Figure 2b, Creal works as follows:

Step 1. Seed analysis: identify variables or constants after which function calls can be injected. For example, the variables "d" and "a" from line 8 in Figure 2b are valid candidates. Other valid expressions include "0" and "2" in line 5, "b" and "1" in line 10, *etc.*

Step 2. Seed profiling: instrument and profile the program to collect values of in-scope variables. Suppose that only variable "d" in line 8 is matched. Since "a,b,c,d" are in-scope variables in line 8, we collect values of variables a,b,c,d in line 8 with our instrumentation. As indicated by line 7, the profile includes all run-time values of these variables: both "a" and "b" have multiple values, while "c" and "d" have single values.

Step 3. Function call synthesis: synthesize a function call to the real-world function. Since "d" is selected, our target is to synthesize a function call that can replace "d" while maintaining the execution semantics. We then (i) *Choose a function from the database.* We assume "print_dec()" is chosen. (ii) *Synthesize the input parameters to the chosen function.* All input parameters should be identical to the cached input so that we know the return value from the function call during synthesis. Because the first parameter of "print_dec()" is a pointer, we place "char h[2]" ahead of the current statement as shown in line 24 in Figure 2c and use "h" as the first parameter. We do not use any pointer from the seed program to avoid modification to the seed program's memory state. Because the second and last inputs are 75 and 48, we can simply use "print_dec(h,75,48)" as the function call. In order to build data dependency between the seed program and the injected function, we use in-scope variables as input parameters. For example, since "c=1" and "d=0", we use "print_dec(h,c+74,d+48)" as the function call, which returns the same value as "print_dec(h,75,48)". (iii) *Return value compensation:* compensate the return value of the function call to cancel its effect. Since "print_dec()" returns 2, we append "-2" after the function call. The final synthesized expression is “`print_dec(h, c+74, d+48)-2`”, which is evaluated to `0=d` and thus does not modify the run-time semantics of the seed program.

The final generated program is shown in Figure 2c. The injected function call builds dependency between the seed program and the real-world function, bringing the rich features from the real-world function into the seed program. In practice, Creal will insert more than one function call to enhance the expressiveness of the final program further.

3 APPROACH

This section presents our real-world function injection approach. Section 3.1 introduces basic concepts about the constructed function database. Section 3.2 describes the high-level algorithmic sketch of our approach. Sections 3.3 to 3.5 detail the main steps in the sketch. Section 3.6 presents our implementation of Creal.

3.1 Function Database

Our approach is driven by real-world functions. We design and build a function extractor that can automatically extracting, transforming, and profiling functions from source code files. We defer the technical details of function database construction to Section 4. Let us assume a constructed function database $\mathcal{D}_F$ is available. For each function $F_i \in \mathcal{D}_F$, F_i has the following key properties:

(1) F_i takes only numeric input types and returns a numeric value.

(2) F_i does not contain any other function calls.

(3) F_i is pure, meaning that F_i has deterministic outputs and has no side effects.

The first property allows us to avoid passing pointers from the seed program as input parameters, preventing uncontrolled modifications to the seed program’s memory state. For the illustrative function “`print_dec()`” in Figure 2a that takes a pointer as input, we design a transformation that transforms it into a numeric input-only function. Technical details will be clarified in Section 4.1. To simplify our presentation, *we will always assume that all functions take only numeric input types.* The second property indicates that each function can be compiled and executed independently. The last property allows us to fully model a function’s semantics by its input/output pairs.

3.2 Algorithmic Sketch

Our high-level idea is to substitute an expression in a seed program with a function call to a real-world function. The resulting program should have the same semantics as the seed program. Algorithm 1 presents the general process of generating a new program. Given a seed program $\mathcal{P}$ and an input $\mathcal{I}$ to the program, our generator works as follows:

Step 1. *Expression matching* (line 2): find all expressions in the seed program $\mathcal{P}$ that satisfy a set of predefined criteria (see Section 3.3). These expressions will later be replaced with function calls in a certain probability.

Step 2. *Program profiling* (line 3): instrument and run the program $\mathcal{P}$ on input $\mathcal{I}$ to collect the execution profile $\widehat{prof}$, which contains all the needed run-time information at the program location of each matched expression. (see Section 3.4)

Step 3. *Skipping an expression* (line 6): for the matched expressions, we do not aggressively substitute all of them with function calls. First, we query the execution profile to see if *expr* is alive, *i.e.*, exists in the live rather than dead code region. Then, for live expressions, we do substitution in a certain probability. The function `FlipCoin()` returns either `true` or `false`.

Algorithm 1: Program generation

```
1  procedure Generate(Seed Program $\mathcal{P}$, Input $\mathcal{I}$, Function Database $\mathcal{D}_F$):
       // find all matched expressions $expr$
2      $\mathcal{E} \leftarrow$ GetMatchedExpr($\mathcal{P}$)
       // profiling the program at all program locations of $\mathcal{E}$
3      $\widehat{prof} \leftarrow$ Profile($\mathcal{P}$, $\mathcal{I}$, $\mathcal{E}$)
4      $F\_list \leftarrow [\,]$
5      foreach $expr \in \mathcal{E}$ do
           // decide if this expression needs to be substituted
6          if IsAlive($expr, \widehat{prof}$) and FlipCoin() then
               // select a function from the database
7              $F \leftarrow$ SelectFunction($\mathcal{D}_F$)
               // synthesize the function call
8              $expr' \leftarrow$ SynFuncCall($expr, F, \widehat{prof}$)
               // substitute the original expression with the function all
9              $\mathcal{P} \leftarrow$ InsertFunc($\mathcal{P}, expr, expr', F$)
10             if $F \notin F\_list$ then
                   // remember all used $F$
11                 $F\_list$.append($F$)
       // insert function declarations
12     InsertFuncDecl($\mathcal{P}, F\_list$)
13     return $\mathcal{P}$
```

The probability for `FlipCoin()` to return `true` can be configured in our Creal implementation. The reason why we skip dead code regions is that previous work [Le et al. 2015; Sun et al. 2016] has shown that mutating on live regions is more likely to exercise interesting compiler optimizations and trigger bugs.

Step 4. *Function selection and function call synthesis* (lines 7-8): a function $F \in \mathcal{D}_F$ is randomly selected. With the execution profile $\widehat{prof}$, we know the evaluation results of *expr* and a set of in-scope variables. The `SynFuncCall` procedure will then synthesize a function call expression that is evaluated to the same value as *expr*. (see Section 3.5)

Step 5. *Function call insertion* (lines 9-11): the new expression $expr'$ with a function call will replace the original expression *expr* in $\mathcal{P}$. This function will also be appended into F_list.

Step 6. *Function declaration insertion* (line 12): after the iteration is over, insert the declarations of all used functions to the beginning of $\mathcal{P}$, making it to be compilable and executable.

As indicated in the algorithm, the generation happens iteratively over all matched live expressions. The resulting program contains more than one real-world function call. For the rest of this section, we will introduce in detail the three key steps, *i.e.*, *expression matching*, *program profiling*, and *function call synthesis*.

$$\begin{aligned}
S &::= E \oplus \langle e \rangle \mid \langle e \rangle \oplus E \mid S' \odot E \\
E &::= \langle c \rangle \mid \langle v \rangle \mid * \langle v \rangle \mid \langle v \rangle [\langle e \rangle] \\
\langle c \rangle &::= \textit{constants} \\
\langle v \rangle &::= \textit{variables} \\
\langle e \rangle &::= \textit{all expressions in C} \\
\oplus &::= \textit{binary operators}, \text{ e.g., } + \mid - \mid * \mid \%, \text{ etc.} \\
\odot &::= \textit{assignment operators}, \text{e.g., } = \mid += \mid *=, \text{ etc.} \\
S' &::= \textit{left operand of} \odot \textit{ (ignored)}
\end{aligned}$$

Source code	Matched expressions
`a + b`	`a, b`
`(a + b) - c`	`a, b, c`
`*a + b`	`*a, b`
`a = b + 1`	`b, 1`
`a += b + c[1]`	`b, c[1]`

Fig. 3. Syntax rules for matching expressions E appearing in S. Note that, E is of numeric type, such as `int`.

3.3 Expression Matching — GetMatchedExpr(·)

Given a seed program $\mathcal{P}$, we statically scan $\mathcal{P}$ to find all expressions where function calls can be injected. Figure 3 gives the syntax rules used for matching target expressions and a set of matched expression examples. There are two main criteria:

- The expression has to be non-left, *i.e.*, not on the left-hand side of an assignment expression. This criterion ensures the syntactic validness after we inject a function call. For instance, replacing "`a`" in "`a = b + 1`" with a function call violates the C language specification.
- The expression has a numeric type. Since all functions in the database have numeric input/output types, we match numeric expressions only for type compatibility.

A valid expression can contain multiple matched expressions. For example, "`(a+b)-1`" contains three expressions that satisfy our criteria, namely "`a`", "`b`", and "`1`". Specifically, we match three types of expressions, *i.e.*, (1) constants, (2) variables having numeric types, and (3) pointer dereferences or array accesses having numeric types such as "`*a`" or "`c[1]`" of type `int`.

3.4 Program Profiling — Profile(·)

Program profiling collects the run-time variable values at target program locations. The target program locations $\mathcal{L}$ are a set of lines that contain at least one matched expression. Let us denote all matched expressions at the program location l as $\mathcal{E}_l$ and all in-scope variables at l as $\mathcal{V}_l$. It is clear that *all variables from* $\mathcal{E}_l \in \mathcal{V}_l$. With this notation, we define the execution profile as follows:

Definition 1 (Execution Profile). Given a program $\mathcal{P}$, an input $\mathcal{I}$, and program locations $\mathcal{L}$. By running $\mathcal{P}$ on the input $\mathcal{I}$, the execution profile $\widehat{prof}$ records the values of expressions in $\mathcal{V}_l$ at each program location $l \in \mathcal{L}$.

Figure 4 shows the execution profile of an example program. In this program, the target program locations are $\mathcal{L} = \{5, 6, 8\}$ because only these three lines contain matched expressions. In line 5, the variable `b` and constant `1` are matched expressions; variables `a`, `b` and `g` are in the variable set $\mathcal{V}_5$ as they are all in-scope. In line 8, there is an extra in-scope variable `c`. The execution profile, as shown in the bottom right of the figure, records all observed run-time values of variables in each $\mathcal{V}_i$.

For a non-trivial seed program $\mathcal{P}$, the number of in-scope variables $\mathcal{V}_i$ at each program location is typically large. Recording all of their values is thus costly. Since only a part of variables from $\mathcal{V}_i$ will be used for synthesis, to reduce the profiling cost, our implementation of Creal randomly selects a small yet sufficient subset of in-scope variables in each $\mathcal{V}_i$. To facilitate easy access to the execution profile $\widehat{prof}$, we design the following queries:

```c
 1 int g=4;
 2 int main() {
 3   int a = 0, b = 1;
 4   for (;;) {
 5     a += b + 1;
 6     if (a > g) {
 7       int c = 3;
 8       g = a + c;
 9       break;
10     }
11   }
12 }
```

Program locations: $\mathcal{L} = \{5, 6, 8\}$

Matched expressions: $\mathcal{E}_5 = \{\mathtt{b}, 1\}$

$\mathcal{E}_6 = \{\mathtt{a}, \mathtt{g}\}$

$\mathcal{E}_8 = \{\mathtt{a}, \mathtt{c}\}$

Execution profile:

$\mathcal{V}_5 : \mathtt{a} = \{0, 2, 4\}, \mathtt{b} = \{1\}, \mathtt{g} = \{4\}$

$\mathcal{V}_6 : \mathtt{a} = \{2, 4\}, \mathtt{b} = \{1\}, \mathtt{g} = \{4\}$

$\mathcal{V}_8 : \mathtt{a} = \{4\}, \mathtt{b} = \{1\}, \mathtt{c} = \{3\}, \mathtt{g} = \{4\}$

Fig. 4. The program on the left is the seed program. $\mathcal{L}$ is the program locations where we have matched expressions $\mathcal{E}$. The execution profile shows the run-time values of in-scope variables at each program location.

$$
\begin{aligned}
\langle e \rangle &::= \langle c \rangle | \langle v \rangle | (\langle e \rangle) | \ominus \langle e \rangle \\
&\quad\; | \langle e \rangle \oplus \langle e \rangle | \langle F \rangle \\
\langle c \rangle &::= \textit{constants} \\
\langle v \rangle &::= \textit{variables} \\
\langle F \rangle &::= \textit{function call} \\
\ominus &::= \textit{unary operators, e.g.,} \; ! | \sim \\
\oplus &::= \textit{binary operators, e.g.,} \; + | * | \%
\end{aligned}
$$

Fig. 5. BNF grammar for expression synthesis.

Algorithm 2: Function call expression synthesis.

```
procedure SynFuncCall(Program Location $l$, Profile $\widehat{prof}$, Database $\mathcal{D}_F$, Target Value $\overline{\mathtt{val}}$):
    // randomly select a function from the database and get its input set
    $F_i \leftarrow$ SelectFunction($\mathcal{D}_F$)
    [$\overline{\mathtt{inp}_1}, \overline{\mathtt{inp}_2}, \cdots, \overline{\mathtt{inp}_m}$ ] $\leftarrow F_i$.input
    // initialize the function call template
    FC = " $F_i$.name (<para>$_1$, <para>$_2$, $\cdots$, <para>$_m$)"
    // iteratively synthesize the function parameters
    foreach $k \in [1 \ldots m]$ do
        $V \leftarrow$ GetStableVariable($\widehat{prof}$, $l$)
        // sythesize expression $para$ using $V$ such that $[[para]] = \overline{\mathtt{inp}_k}$
        $para \leftarrow$ SynthesizeExpression ($V$, $\overline{\mathtt{inp}_k}$)
        FC.Substitute(<para>$_k$, para)
    $V' \leftarrow$ GetStableVariable($\widehat{prof}$, $l$)
    // synthesize expression $expr'$ using $V'$ and FC such that $[[expr']] = \overline{\mathtt{val}}$
    $expr' \leftarrow$ SynthesizeExpression ($V' \cup$ FC, $\overline{\mathtt{val}}$)
    return $expr'$
```

- **`GetStableVariable(`$\widehat{prof}, l$`)`**: returns all in-scope variables where *the variable has only one run-time value.* For example, `GetStableVariable(`$\widehat{prof}$, 5`)` returns only {`b`, `g`} but not `a`.
- **`GetVariableValue(`$\widehat{prof}, l, v$`)`**: returns the run-time value(s) of variable v at location l, *e.g.*, `GetVariableValue(`$\widehat{prof}$, 5, `b)` returns {1}.

3.5 Function Call Synthesis — SynFuncCall(·) & InsertFunc(·)

Given a selected expression *expr*, we synthesize a new expression *expr*′, such that

$$[[expr']] = [[expr]]$$

where $[[x]]$ denotes the run-time value of expression x. The above equation indicates the run-time values of *expr* and *expr*′ are identical. Let $\mathcal{P}[expr/expr']$ be a program formed by replacing *expr* with *expr′* at the same source location. The run-time value equivalence of *expr* and *expr′* guarantees that (1) $\mathcal{P}[expr/expr']$ has the same output as $\mathcal{P}$, and (2) if both $\mathcal{P}$ and *expr′* are well-formed, *e.g.*, free of undefined behavior, $\mathcal{P}[expr/expr']$ is also a well-formed program.

There are two cases for *expr*: (1) *expr* is stable, meaning that $[[expr]]$ is a single value $\overline{\texttt{val}}$, *e.g.*, variable "b" at line 5 in Figure 4. In this case, we require $[[expr']] = \overline{\texttt{val}}$. (2) *expr* is unstable, meaning that $[[expr]]$ has different values at run-time, *e.g.*, variable "a" at line 5 in Figure 4. In this case, we synthesize the new expression as "$expr' = expr + \widehat{expr}$", where $[[\widehat{expr}]] = 0$. This ensures that $[[expr']] = [[expr]]$. The second case can be viewed as the first case by treating $\widehat{expr}$ as the target expression *expr′*. Without loss of generality, our presentation always assumes that *expr* is stable, and our target is to synthesize a new expression *expr′* that evaluates to $[[expr]] = \overline{\texttt{val}}$.

Algorithm 2 shows the general procedure of synthesizing a new expression. It first randomly selects a function item F_i from the constructed function database $\mathcal{D}_F$ and obtains the function input set (lines 2-3). Note that all $\overline{\texttt{x}}$ such as $\overline{\texttt{inp}_1}$ and $\overline{\texttt{val}}$ are concrete numeric values, *e.g.*, "0", "2", *etc.* Then, it initializes a function call template as FC (line 4). This template is of string format and has a set of placeholders for input arguments, *i.e.*, <para>$_1$, <para>$_2$, $\cdots$, and <para>$_m$. Next, it tries to replace all these placeholders with concrete expressions. For each of the parameter placeholder $\texttt{para}_k$, it obtains stable variables V from the execution profile (line 6). With the grammar shown in Figure 5, it uses V and synthesizes an expression *para* that evaluates to $\overline{\texttt{inp}_k}$ (line 7). The parameter placeholder <para>$_k$ is then substituted by the synthesized parameter expression *para* (line 8). After the for loop, the function call to "F_i.name(...)" is evaluated to F_i.output. Finally, with the grammar shown in Figure 5, it synthesizes an expression with a set of stable variables V' and the synthesized function call FC. This new expression *expr′* is now evaluated to the target value $\overline{\texttt{val}}$.

Example. We use an example to illustrate the algorithm. Let us assume the target value is $\overline{\texttt{val}} = 5$:

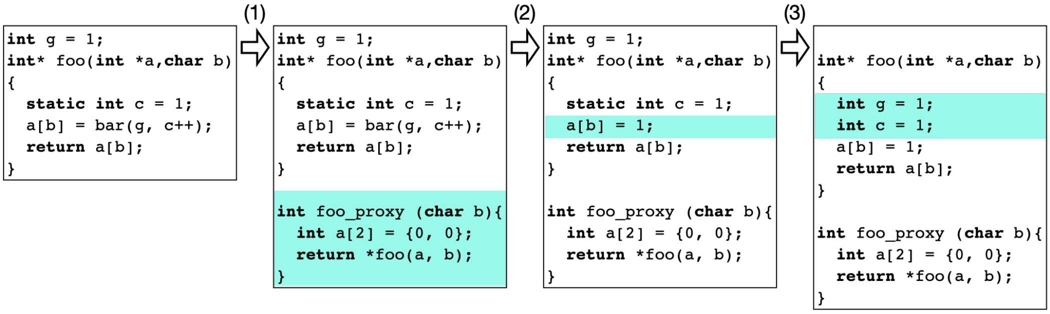
- In line 2, suppose the selected function is "`int real(int)`", which takes an integer as input and returns an integer value.
- In line 3, suppose the single input is $\overline{\texttt{inp}_1} = 2$ and the function return is 1.
- In line 4, the template is then "real(<para>$_1$)".
- In lines 6-7, suppose the stable variables V is $\{ b \}$ and "b" has single value 1. One instance of the synthesized expression can be "$b + 1$", which evaluates to $2 = \overline{\texttt{inp}_1}$.
- In line 8, the synthesized input expression is then "$b+1$" and the function call FC is "real(b+1)".
- In line 10, suppose the stable variables V' is $\{ g \}$ and "g" has single value 4. One instance of the synthesized expression can be "$(-\texttt{real}(b + 1) * 2) + g * 2 - 1$" because

$$\begin{aligned}(-\texttt{real}(b + 1) * 2) + g * 2 - 1 &= (-1 * 2) + g * 2 - 1\\ &= (-1 * 2) + 4 * 2 - 1\\ &= 5 = \overline{\texttt{val}}\end{aligned}$$

The synthesized expression *expr′* fuses the function call with variables in the seed program. This fusion builds rich dependencies between the seed program and the inserted functions. The original expression *expr* is then replaced by *expr′*.

3.6 Implementation

We implemented the proposed program generator in Creal. Taking a seed program $\mathcal{P}$ and a function database $\mathcal{D}_F$ as inputs, Creal utilizes Clang's Libtooling [LLVM 2023a] to instrument profiling code into $\mathcal{P}$ and collects its execution profile. When generating a mutant, Creal replaces multiple expressions in the seed program with different synthesized function call expressions.



```
int g = 1;
int* foo(int *a,char b)
{
  static int c = 1;
  a[b] = bar(g, c++);
  return a[b];
}
```

```
int g = 1;
int* foo(int *a,char b)
{
  static int c = 1;
  a[b] = bar(g, c++);
  return a[b];
}

int foo_proxy (char b){
  int a[2] = {0, 0};
  return *foo(a, b);
}
```

```
int g = 1;
int* foo(int *a,char b)
{
  static int c = 1;
  a[b] = 1;
  return a[b];
}

int foo_proxy (char b){
  int a[2] = {0, 0};
  return *foo(a, b);
}
```

```
int* foo(int *a,char b)
{
  int g = 1;
  int c = 1;
  a[b] = 1;
  return a[b];
}

int foo_proxy (char b){
  int a[2] = {0, 0};
  return *foo(a, b);
}
```

Fig. 6. Step-by-step transform an invalid function into a pure function.

Type conversion. In the above presentation, we assumed that there is only one numeric type in the program. In practice, the C language supports many types such as `int`, `char`, and `unsigned`. When an operation happens between values of different types, the compiler will do implicit type conversions. For example, the evaluation results of "$a + b$" are different when "a" has different types:

$$\begin{aligned} \texttt{int}\ \ a = 0; \texttt{int}\ \ b = -1; \quad &\Rightarrow \quad a + b = -1\ \ (\texttt{int}) \\ \texttt{unsigned}\ \ a = 0; \texttt{int}\ \ b = -1; \quad &\Rightarrow \quad a + b = 4294967295\ \ (\texttt{unsigned}) \end{aligned}$$

For the second case, b is cast into `unsigned` and thus its value becomes 4294967295 (the maximum value for a 32-bit unsigned integer). In Creal, we carefully evaluate the expression values when type conversion is possible. We also add explicit casts to make sure the type of $expr'$ is the same as the original expression $expr$.

Avoiding undefined behavior. When synthesizing expressions in Creal, it is important to guarantee that the synthesized expressions are well-formed. Since our expressions involve only arithmetic operations, we only need to avoid integer overflow. This can be easily achieved because we know the concrete evaluation values of all involved variables.

Generated programs. Our approach is general and can in principle take programs from any generators as input. In our implementation, we use Csmith as default generator to generate seed programs. In this case, every seed program is in a single source file. After injecting function calls, Creal includes function definitions for all used functions in the source file. The generated program by Creal is thus a single source file, as exemplified in Figure 2c.

4 CONSTRUCTING FUNCTION DATABASE

This section presents details about constructing a function database to fuel our approach.

4.1 Extracting and Transforming Functions

We design a function extractor to obtain functions from source code files. We utilize Clang's Libtooling [LLVM 2023a] for implementation. The extractor finds all function definitions in a pre-processed input program, *e.g.*, programs processed by "`clang -E`". The principal selection criteria are (a) the function only involves primitive types, and (b) the function should be pure, *i.e.*, with deterministic outputs and no side effects. The first requirement allows us to compile and execute functions without dealing with user-defined types. The second requirement allows us to model the execution of a function purely by its I/O behavior. In the context of C, this means

```c
#include <stdio.h>

/* Placeholder for function definition */
F.def;


int main() {
  /* randomly select values for arguments */
  F.arg_type[0]  p1 = RAND();
  F.arg_type[1]  p2 = RAND();
  ...
  /* invoke the function */
  F.ret_type    ret = F.name(p1, p2, ...);

  /* output the IO */
  LOG(p1); LOG(p2); ...;
  LOG(ret);
}
```

(a) Driver template.

```c
#include <stdio.h>
/* function definition */
int add(char a, int b) {
   int r = a + b;
   return r;
}
int main() {
  /* randomly select values for arguments*/
  char   p1 = RAND(char);
  int    p2 = RAND(int);

  /* invoke the function */
  int   ret = add(p1, p2);

  /* output the IO */
  LOG(p1); LOG(p2);
  LOG(ret);
}
```

(b) The driver function for "add"

Fig. 7. Template for driver functions and an example driver function.

(1) *The function does not modify its arguments.* Some functions take pointers as arguments, the memory locations pointed to by which can be modified after invoking the function. Instead of discarding such functions, we synthesize proxy functions for them to avoid possible modifications to arguments. For example, Figure 6 (1) shows how we use a proxy function `foo_proxy` to hide the pointer parameter in the original function `foo`. Note that the proxy function also dereferences the return pointer to make it also a primitive type.

(2) *The function does not call other functions that have side effects.* Real-world functions may call other functions. Analyzing other functions to avoid side effects is theoretically feasible but would complicate our implementation and make it fragile. Instead, we choose to remove all other function calls from a function definition. To maintain syntactic correctness, we replace each call with a randomly chosen yet type-compatible value. For instance, Figure 6(2) shows that the extractor replaces the function call "`bar(g, c++)`" with an integer value "`1`".

(3) *The function does not modify global or static variables.* The value of global or static variables may be different when calling the same function multiple times. Such side effects break our intention of modeling a function by its I/O behaviors. Our extractor will remove all "static" keywords from a function definition and then lower all global variables into local variables. For example, the extractor changes variables `g` and `c` as shown in Figure 6(3).

Since we utilize a function's I/O for program generation, we also exclude functions that take no inputs or return void. With the designed function extractor, we can extract a set of functions from real-world projects. We denote the constructed function database as $\mathcal{D}_F = \{F_1, F_2, \cdots, F_n\}$, where each F_i contains the static information regarding the function and has the following information:

- F_i.`name`: the unique name of the function in the database, *e.g.*, "`foo_proxy`" in Figure 6.
- F_i.`arg_type`: the types of each argument, *e.g.*, ["`char`"].
- F_i.`ret_type`: the return type, *e.g.*, "`int`".
- F_i.`def`: the function definition, *e.g.*, "`int* foo(...){...} int foo_proxy(...){...}`".

4.2 Constructing Function Database with I/O

Caching the input/output (I/O) behaviors of a function can help us precisely synthesize a function call and avoid ill-formed functions prior to synthesis. For each function in $\mathcal{D}_F$, we utilize a driver function to learn (1) its I/O pairs, and (2) whether or not the function is well-formed, *i.e.*, whether it contains undefined behavior. Figure 7a shows the template for the driver function, and Figure 7b shows an example driver function for the function “add()”. For a function $F_i \in \mathcal{D}_F$, the general I/O construction works as follows:

(1) Include the function definition F_i.def in the driver template, *i.e.*, line 4 in Figure 7a and lines 3-6 in Figure 7b.

(2) For each argument of F_i, synthesize a statement to randomly pick a value with this type, *i.e.*, lines 9-11 in Figure 7a and lines 9-10 in Figure 7b. “RAND()” function only returns values within the valid range for the given type. For instance, RAND("char") returns values between -128 and 127.

(3) Synthesize a function call statement and put the return value into a variable with type F.ret_type, *i.e.*, line 13 in Figure 7a and line 13 in Figure 7b.

(4) Log the inputs and output, *i.e.*, lines 16-17 in Figure 7a and lines 16-17 in Figure 7b.

The synthesized driver function is then compiled and executed. We use sanitizers [Team 2023] and CompCert [Leroy 2009, 2023] to examine whether or not the driver function is well-formed. If all examinations are successful, we save the logged input/output pair into the database. For instance, one valid input/output pair of Fig. 7b is {input=[1,2], output=3}. In practice, we will try to run each driver function multiple times to save more than one input/output pair into the database. Now, new items F_i.input and F_i.output are added to the database. *At this point, all functions in the database are free of undefined behaviors when calling them with the inputs cached in F_i.input.*

5 EVALUATION

This section presents the details of our extensive evaluation of Creal, demonstrating its practical effectiveness. In a nine-month period until mid-November 2023, we ran Creal to test two open-source compilers, GCC and LLVM. Our testing results are summarized as follows:

- Creal *has found many new bugs.* Within 9 months of testing, Creal has detected 132 bugs, of which compiler developers have confirmed 121 bugs and fixed 97 bugs.
- Creal *has found many miscompilation bugs.* Of all 132 detected bugs, 79 (60%) of them are miscompilation bugs, the most severe and hard-to-detect compiler bug type.
- Creal *has found many latent bugs.* We reported 41 latent bugs in GCC and LLVM. These bugs have escaped all previous generation- and mutation-based compiler testing techniques.

5.1 Experimental Setup

Compiler versions. Our evaluation focuses on widely used and mature production C compilers, GCC and LLVM. To avoid reporting previous known bugs, we used Creal to test the latest development versions of both GCC and LLVM. We used all five standard optimization levels, *i.e.*, -O0, -O1, -Os, -O2, and -O3, in both compilers for extensive testing.

Seed programs. We used Csmith [Yang et al. 2011], a mature random C program generator, to generate seed programs for Creal. Csmith has been widely adopted by many compiler testing techniques [Chen et al. 2020; Even-Mendoza et al. 2021; Lidbury et al. 2015] and has been *de facto* default program generator for C compiler testing. There are three main strengths in using Csmith: (1) it can generate complex programs, providing Creal rich opportunities to mutate on; (2) all Csmith-generated programs are self-contained, meaning that they do not take external inputs and can be executed; and (3) Csmith-generated programs and their mutants can be effectively reduced with existing tools like C-Reduce [Regehr et al. 2012].

Function database. We used our function extractor to extract functions from open-source projects. AnghaBench [da Silva et al. 2021] provides intermediate access to the source files of 146 most

Table 1. Status of the reported bugs.

Status	GCC	LLVM	Total
Reported	3	1	4
Confirmed	13	11	24
Fixed	38	59	97
Duplicate	5	2	7
Total	59	73	132

Table 2. Type of found bugs.

Symptom	GCC	LLVM	Total
Crash	22	31	53
Miscompilation	37	42	79

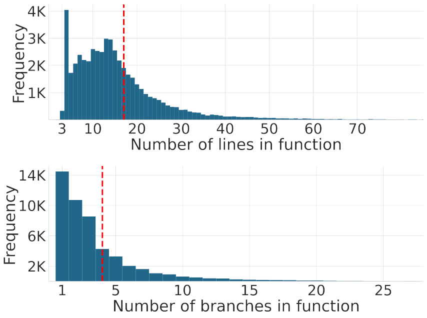


Fig. 8. Statistics about #lines and #branches of functions in the constructed database.

starred open-source C projects on GitHub, such as OpenSSL, PHP, Linux kernel, to name a few. On top of AnghaBench, we crawled over one million functions from these 146 projects. After extraction and I/O generation as described in Section 4, we collected a function database of ***51,356 functions***. The majority of crawled functions are discarded due to unsupported input or return types. We count the number of lines and branches in our collected functions. Figure 8 shows the distributions. We can see that most functions have fewer than 30 lines and 10 branches. On average, the number of lines in a function is 17 while the number of branches in a function is 4.

Hardware. We conducted all our evaluations on two Linux servers running Ubuntu 20.04 LTS. Both are equipped with an AMD EPYC 7742 64-core CPU and 256GB RAM.

Testing process. Our testing process runs continuously and is fully automated. We first use Csmith to generate a seed program, then apply Creal to generate 10 mutants. The `FlipCoin()` probability in Section 3.2 is set to 20%. Then, we use both GCC and LLVM with different optimization levels to compile and run these programs. Since all mutants are semantics-preserving, we use the output from the seed program as the reference oracle. Once a compiler crash or miscompilation is observed, we use C-Reduce to reduce the bug-triggering program into a small reproducing program and then make a bug report with it.

5.2 Quantitative Results: Bug-Finding

Number of bugs. Table 1 summarizes the status of all found bugs in GCC and LLVM. As of mid-November, 2023, we have filed a total of 132 bug reports to GCC and LLVM. The compiler developers have acknowledged, *i.e.*, confirmed or fixed, 92% of them (121/132) as previously unknown and new bugs, while they have fixed 73% of them (97/132). In particular, GCC and LLVM developers confirmed or fixed respectively 86% and 96% of our filed bugs. As mature and production compilers, both GCC and LLVM have been extensively tested in industry and academia. The significant number of new bugs highlights the substantial bug-finding ability of Creal. The "Reported" bugs are still waiting for developers' confirmation. Since there are other compiler developers, users, and testers reporting bugs, 5 bugs in GCC and 2 bugs in LLVM are marked as "Duplicate".

Types of bugs. Table 2 summarizes the types of found bugs. All bugs can be categorized into the following two types: (1) "Miscompilation": The compiler incorrectly compiles the program and produces a wrong executable code, which has a different semantics from the source program, and (2) "Crash": The compiler crashes when compiling the program due to either run-time failures or

Table 3. Affected LLVM components.

Component	#Bugs
Peephole Optimizations	17
Loop Transformations	11
Backend	6
Induction Variable Trans.	6
Scalar Evolution Analysis	5
Global Value Numbering	4
SLP Vectorization	4
Selection DAG	3
Alias Analysis	2
Induction Variable Analysis	2
CFG Transformations	1
Dead Store Elimination	1
Dominance Optimizations	1
Escape Analysis	1
Pass Management	1
Vectorization Optimizations	1

Table 4. Affected GCC components.

Component	#Bugs
Loop Transformations	10
Peephole Optimizations	9
CFG Transformations	7
Loop Analysis	3
Value Range Analysis	3
Vectorization	3
Constant Propagation	2
IR Data Structures	2
Dead Store Elimination	1
Jump Threading	1
Liveness Analysis	1
Predictive Commoning	1
Redundancy Elimination	1

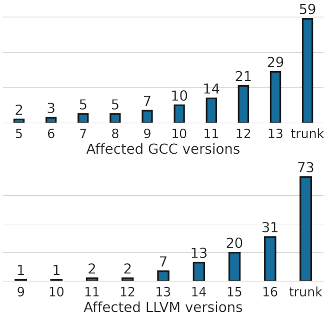


Fig. 9. Stable compiler versions that are affected by our reported bugs.

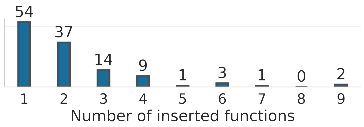


Fig. 10. Number of inserted functions in reduced bug-triggering programs.

assertion failures. As the table shows, more than half of the bugs (79 out of 132) are miscompilations, the most critical and hard-to-detect bugs [Chen et al. 2020; Sun et al. 2016].

Importance of bugs. The fix of 97 out of 132 bugs, predominantly miscompilations, has highlighted the critical impact of the bugs. For GCC bugs, developers will classify bugs according to their priority and severity. P3 is the default while P1 is the highest priority. *Developers have to fix all P1 bugs before making the next release.* Out of the 59 GCC bugs, 19 (32%) of them are assigned as P1.

To understand the impact of found bugs, we ran each bug-triggering test case on stable compiler versions since GCC-5 and LLVM-9. Figure 9 shows for each compiler version, how many bugs affect it. It indicates that Creal can find many *long-latent* bugs. Specifically, there are 14 bugs affecting GCC versions earlier than GCC-11 and 7 bugs affecting LLVM versions earlier than LLVM-13. These compiler versions have been released for 2 ∼ 8 years. Since these long-latent bugs have escaped all existing testing techniques, it further confirms the significant bug-finding capability of Creal.

Affected compiler components. We studied all fixed bugs on which compiler developers provided detailed diagnoses and fix messages, allowing us to identify the affected compiler components accurately. Tables 3 and 4 list all the affected compiler components in LLVM and GCC. One may be concerned that Creal can only find control-flow related bugs as we only inject function calls into seed programs. However, from the tables, it is evident that these bugs impact a diverse range of compiler components. In both GCC and LLVM, loop transformations and peephole optimizations account for more bugs than other components. This finding is aligned with the existing empirical study [Zhou et al. 2021] that instruction combination (peephole optimization) is one of the most buggy optimizations and loop optimizations are more bug-prone than other optimizations.

5.3 Bug Characteristics

For all the unique and new bugs (121 bugs), we identified what real-world functions account for triggering the bug. Because Creal injected multiple functions into a seed program, we automatically removed all irrelevant functions and only kept the functions that were essential to trigger the underlying bug. Note that, it is possible that more than one function is required to trigger a bug. With this information, we answer the following questions:

➤ **How many functions are in bug-triggering programs?** Figure 10 summarizes the number of real-world functions in each reduced bug-triggering program. The first bar shows that out of all 121 bug-triggering programs, 54 of them have only one single real-world function inserted, demonstrating the ability of one real-world function to enhance seed programs' features. The rest half, nearly all of them, have ≤ 4 inserted functions, with only a few having more than 5 functions. This means that the combination of a small number of functions can further enhance the expressiveness of seed programs.

➤ **Can real-world functions alone trigger bugs?** Figure 10 shows that 54 out of 121 bug-triggering programs contain only one real-world function. One may wonder if the seed programs are necessary, *i.e.*, if we can find many compiler bugs without using Csmith-generated seed programs. To answer this question, we used the driver function from Section 4 as the seed to build a well-formed program for each function in the database. Then, we compiled and ran these programs with different compilers to test if we could find compiler bugs. *The result is that we did not find any compiler bugs.* This emphasizes the importance of incorporating complex run-time semantics from seed programs.

We also checked if there are functions that lead to to multiple bugs. We identified that 5 real-world functions contribute to more than one bug, highlighting the importance of injecting real-world functions into seed programs: *different injections may uncover different bugs.*

➤ **What are the unique features in bug-triggering functions?** We manually analyzed each minimized bug-triggering program to figure out what unique code features are in the real-world functions. We find two notable features that appear in 21 bugs:

- *Unbounded complex loop*, where the loop conditions are not bounded by a constant value and have complex dependency with the loop body. This feature is present in 11 bug-triggering programs. As discussed before, generating complex loops is difficult for random program generators, due to the need to guarantee termination and avoid undefined behavior.
- *Employing switch/case statements*, where the function employs "`switch...case...`" statements to build control flow. This feature is present in 10 bug-triggering programs. Each `switch` often comes with multiple ($5 \sim 21$) `case`s conditioning on the same expression. Csmith does not yet support this feature.

The remaining bug-related functions have various features. They typically have either unique syntax such as the above two constructs or unusual real-world semantics such as *flipping bits* and *rotating arrays.* Although random program generators support the syntax in these functions, the underlying semantics are nearly impossible to be from random generation. We will include more sample bugs in Section 5.7 to further demonstrate this point.

5.4 Code Coverage and Generation Speed

We conduct code coverage analysis to understand if Creal can increase code coverage. We randomly generated 1,000 seed programs with Csmith. Then, we use Creal to generate 10 mutants per seed, which leads to a total of 10,000 programs from Creal. For a fair comparison against Csmith, we also used Csmith to generate 10,000 random programs. For each set of programs, we collected their function, line, and branch coverage on GCC and LLVM[3]. Table 5 reports the results. It is clear that Creal achieves significantly higher code coverage. Compared to the seed programs, Creal remarkably increased line coverage in GCC by 6.3% (55,761 more lines) and in LLVM by

[3] We used `gcov` for GCC coverage collection and `llvm-cov` for LLVM coverage collection.

Table 5. Line coverage (LC), function coverage (FC), and branch coverage (BC) of GCC and LLVM.

Compiler	Generator	LC	FC	BC
GCC	Seeds (1,000)	30.9%	34.3%	19.2%
	Csmith (10,000)	33.6% (+23,897)	35.5% (+1,055)	21.4% (+24,055)
	Creal	**37.2% (+55,761)**	**37.5% (+2,813)**	**24.0% (+52,485)**
LLVM	Seeds (1,000)	33.3%	24.9%	18.2%
	Csmith (10,000)	34.7% (+22,788)	25.8% (+801)	20.3% (+14,166)
	Creal	**35.9% (+42,952)**	**26.9% (+1,864)**	**22.1% (+27,382)**

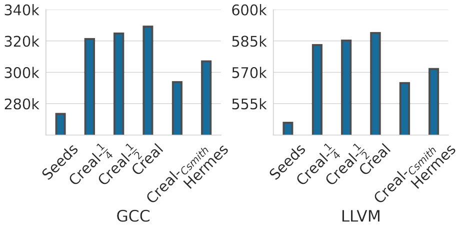


Fig. 11. Line coverage of different variants of Creal and Hermes.

2.6% (42,952 more lines). Creal also shows significantly higher coverage compared to the same amount of Csmith-generated programs. Overall, the large amount of extra code regions signifies that Creal-generated programs cover more code features.

With the same set of seeds, we measured the generation speed of Creal. Generating 10,000 mutants took 8,710 seconds, an average of 0.87 seconds per mutant. During our testing process, we observed that compiling and executing programs consumed most of the time (≥ 3 seconds per mutant), while program generation was not a bottleneck.

5.5 Significance of Function Database

To evaluate the impact of real-world functions, we construct three variants of Creal:

- *Creal-$\frac{1}{2}$*: We halve the size of the real-world function database used in Creal. We randomly selected half of the functions and let Creal use this smaller database during generation.
- *Creal-$\frac{1}{4}$*: We further halve again the size of the real-world function database used in Creal.
- *Creal-$_{Csmith}$*: Instead of using real-world functions, we extract functions from Csmith-generated programs. We build a new function database of the same size as in Creal, *i.e.*, 50,000 functions from Csmith-generated programs. Then, we set up Creal to use this new function database.

Comparing against Creal-$\frac{1}{2}$ and Creal-$\frac{1}{4}$ helps us understand if more real-world functions in the database can offer richer code features. Comparing against Creal-$_{Csmith}$ helps us understand the necessity and importance of using functions from real-world projects.

Code coverage analysis. We ran these variants on the same 1,000 seeds as in Section 5.4. We let each tool generate 10 mutants per seed. Figure 11 shows the covered lines by each variant. Overall, our default Creal achieves the highest line coverage in both GCC and LLVM. Compared to both Creal-$\frac{1}{4}$ and Creal-$\frac{1}{2}$, we can see that with the increase of database size, the line coverage also increases. This confirms the necessity of using a large set of real-world functions. Compared to Creal-$_{Csmith}$, which has the same number of functions as Creal, all variants based on real-world functions significantly surpass it. We can further confirm that real-world functions enhance the functionality and expressiveness of seed programs.

Bug-finding analysis. Our analysis of bug-triggering programs in Section 5.3 has shown that *nearly all of the bugs are triggered by different real-world functions* except for 5 functions. When using a smaller database such as Creal-$\frac{1}{4}$, it is very likely to exclude many bug-relevant functions. Using such a small database would not be able to detect as many bugs as the full-size database. It is thus clear that *reducing the number of real-world functions hinder the bug-finding capability of Creal.*

We also evaluated the bug-finding capability of Creal-$_{Csmith}$. To this end, (1) we ran Creal-$_{Csmith}$ for one week on the same machine to see if it could find any new bugs, and the result shows that Creal-$_{Csmith}$ failed to find any new bugs. (2) we selected all fixed bugs (97 bugs) found by Creal, then applied Creal-$_{Csmith}$ to mutate each seed program that was used by Creal to trigger the bug.

Considering the stochastic nature of Creal, we let Creal-$_{Csmith}$ mutate for one hour, generating more than 200 mutants per seed (instead of 10 used in Creal). In the end, Creal-$_{Csmith}$ re-discovered 3 bugs in LLVM and 0 in GCC, which are all recent regressions. All three bugs are crash bugs, and no miscompilation was found.

5.6 Comparison of Creal v.s. Hermes

We also compared Creal against Hermes [Sun et al. 2016], which is the most recent and powerful EMI-based compiler testing tool.

Code coverage analysis. We used the same 1,000 seeds as in Section 5.4 and applied Hermes to generate 10 mutants per seed. In the end, we obtained 10,000 Hermes-generated programs, the same number as Creal. The last bars in Figure 11 show the achieved line coverage of Hermes on GCC and LLVM. Hermes improved line coverage compared to the seed programs, but the improvement is not as significant as Creal. Since all Hermes-injected code snippets are derived from a relatively simple grammar, it is expected to exercise much fewer code features than Creal.

Bug-finding analysis. According to EMI's project website [Group 2023], Hermes has been constantly used for finding compiler bugs until now: "*We have maintained our continuous, extensive effort in stress-testing GCC and LLVM to benefit the entire community*". Despite their efforts, the 121 new compiler bugs demonstrate at least the complementary bug-finding capability of Creal. To understand if Hermes can re-discover the bugs found by Creal, as in Section 5.5, we selected all fixed bugs (97 bugs) and applied Hermes to mutate each seed program that was used by Creal to trigger the bug. For each seed, we ran Hermes for one hour, generating more than 100 mutants per seed (instead of 10 used in Creal). The result shows that Hermes was able to re-discover 3 bugs.

Despite the evaluation favoring Creal, we stand by the fact that the strengths of both Creal and Hermes are complementary. Similar to Csmith, Hermes can be used as a program generator, on top of which Creal can bring in real-world code features.

5.7 Case Study

Figure 12a: This program triggers a miscompilation bug in GCC. The function `real` is extracted from `FreeBSD` [FreeBSD 2023], which converts an input numeric variable `dest` into a pointer and then operates over this pointer. GCC incorrectly handles the `while` loop by asserting that this loop must be executed at least once. The reason is that `*ptr` and `len` are from the same variable `j` and the pointer `*ptr` is used in line 7, which leads GCC to conclude that `len` must not be zero otherwise `*ptr` would deference a null pointer. Line 5, where an integer is converted to a pointer, is essential to trigger this bug. Such corner functionality cannot be covered by generators.

Figure 12b: This program triggers an assertion failure at Clang -O1: *"TableSize >= Values.size() && "Can't fit values in table!""*. The function `real` is extracted from `GBStudio` [Studio 2023] and has more than 20 `case` branches. We omit the majority of them for easier presentation. The compiler incorrectly infers the value range of `effectnum` when simplifying the control flow graph. The multiple branches conditioning on the same variable is important to trigger the bug.

Figure 12c: This program triggers a latent miscompilation bug since GCC-10. The function `real` is extracted from `QMKFirmware` [Firmware 2023], which rotates the bit stream of the input `x`. GCC has built-in rotation optimizations which incorrectly handle this case. The rotation functionality, critical to triggering this bug, is almost impossible for a random program generator to generate.

Figure 12d: This program triggers a miscompilation bug in LLVM. The root cause is in the backend, which incorrectly rewrites the expression at line 7. The operand "`low`" and the last "`-low`" are essential to trigger this bug. The function `real` is extracted from `FreeType` [FreeType 2023].

```
int a, f, i, h;
static int *e, *g;

long real (long dest, int val, long len){
    char *ptr = (char*)dest;
    while (len-- > 0)
        *ptr++ = val;
    return dest;
}

int main() {
    unsigned j = i, c = 0;
    for (a=1; a<2; a++) {
        j = real(j, c, j);
        if (e) break;
    }
    const int **k = &e;
    j && (*g)--; *k = 0;
    printf("%d\n", f);
}
```

(a) GCC at -O1 miscompiles this code. The compiled binary produces 2 instead of 1.

```
static int a=0, *b, *c = &a;  int d, e;
int real(int effectnum,int *converted_num){
    switch (effectnum) {
        case 0x0:
            *converted_num = effectnum;
            return 1;
        case 0x4: return 1;
        case 0x5:
            *converted_num = effectnum+1;
            return 0;
        default: return 0;
    }
}
int main() {
    unsigned h=0;
    for (; a + h <= 6; h = h + 3)
        for (; d; d++);
    int j, *k = c, d = &k != &b;
    int g = real(h, &j);   e = g;
}
```

(b) Clang -O1 crashes on this code. It triggers an assertion failure in SimplifyCFG component.

```
int a, b, c, d;
int *e = &c;
unsigned real(unsigned char x) {
  x = ((x >> 1) & 0x55) | ((x << 1) & 0xaa);
  x = ((x >> 2) & 0x33) | ((x << 2) & 0xcc);
  x = ((x >> 4) & 0x0f) | ((x << 4) & 0xf0);
  return x;
}
void h(unsigned g) {
    *e = 8 > real(g + 86) - 86;
}
int main() {
  d = a && b;
  h(d + 4);
  printf("%d\n", c);
}
```

(c) GCC at -O1/2/3/s miscompiled the code. The compiled binaries output 0 instead of 1.

```
static int a = -3, b;
static char c;
int d;
int real(int low, int high) {
    if (low - high < 0x10000L)
        return low;
    return low + (1 % (- low));
}
int main() {
    int *h[] = {&a, &a};
    for (; c <= 8; ++c) {
        int *i = &b;
        *i |= real(a, 8) + d;
    }
    printf("%d\n", b);
}
```

(d) Clang at -Os miscompiles this code. The compiled binary produces -1 instead of -3.

Fig. 12. Sample reduced programs that trigger compiler bugs.

Although it seems that such a function has trivial syntax, the semantics that the same operand "low" has to appear twice with negated values is non-trivial.

6 RELATED WORK

In this section, we discuss related work in the context of compiler testing.

Generative compiler testing. Generative compiler testing utilizes random programs produced by a well-engineered generator to test compilers. Csmith [Yang et al. 2011] is by far the most impactful program generator for finding C/C++ compiler bugs. It can produce a large amount of well-formed programs and found hundreds of compiler optimization bugs when it was launched more than ten years ago. Csmith has been also adapted for other compiler testing scenarios. For example, CLsmith [Lidbury et al. 2015] modifies Csmith to test OpenCL compilers. YARPGen [Livinskii et al. 2020] and its follow-up second version YARPGen v2 [Livinskii et al. 2023] are more recent re-designed program generators targeting scalar and loop optimizations, respectively. Together, they have discovered hundreds of optimization bugs in a few years.

A key challenge for generative tools is to guarantee the validity of the generated programs. Extensive engineering efforts are an absolute necessity to support a wide range of yet incomplete language features. Due to the stochastic nature of random generation, certain code semantics such as the ones discussed in § 5.7 have low probabilities of being generated. Creal tackles this challenge by embracing the power of rich open-source projects and requires relatively low engineering efforts. Conceptually, our solution can be effortlessly adapted to any generator-based approach.

Mutation-based compiler testing. Another line of compiler testing is based on mutation. The most effective approaches are the series of *equivalence modulo inputs* (EMI) work [Le et al. 2014, 2015; Sun et al. 2016]. Given a seed program, EMI mutates the seed program by removing/changing dead regions or inserting code into live regions. EMI guarantees that the mutated program has the same semantics *w.r.t.* the same input. GrayC [Even-Mendoza et al. 2023] mutates seed programs with coverage guidance. This approach is effective in finding compiler crashes, however, no miscompilation could be found.

Similar to the generators, they are also limited by the designed mutation rules and the seed programs. Although our approach is based on mutation, the large amount of available real-world code provides us with more elevated diversity in generating mutants.

Using real-world code for compiler testing. There are indirect usages of real-world code by using machine learning models for compiler testing. DeepSmith [Cummins et al. 2018] trains an OpenCL program generator on a large set of real-world code. Fuzz4All [Xia et al. 2023] utilizes large-language models to produce programs. One fundamental limitation of these approaches is no guarantee of the validity of generated programs. Instead of using models, LangFuzz [Holler et al. 2012] aims at fuzzing JavaScript interpreters and uses code fragments from bug reports. FreeFuzz [Wei et al. 2022] collects API information from deep learning libraries and then synthesizes API calls to test deep learning compilers. Our work differentiates from these efforts by semantically fusing real-world code with the seed program and ensuring program validity. **Program synthesis.** Much work has been done in the domain of program synthesis [Gulwani et al. 2017]. The goal is to automatically find a program that satisfies the user-intended specifications such as logical constraints [Manna and Waldinger 1980] and input/output examples [Osera and Zdancewic 2015]. The purpose of program synthesis is not testing compilers but rather aiding human programmers, education, *etc.*

7 CONCLUSION

We have presented our idea of using real-world code for boosting compiler testing. Our method first constructs a function database by collecting functions from real-world applications. We then augment each function with input/output pairs to model function semantics precisely. With the function database, we synthesize function calls and inject them into seed programs. We have demonstrated the effectiveness of our tool, Creal, by identifying 132 bugs with 121 confirmed and 97 fixed in the two most popular C compilers, GCC and LLVM.

Our innovative approach of blending real-world code with synthetic programs offers a compelling and optimistic research direction for testing compiler implementations. From a technical standpoint, discovering more effective methods for extracting code could enhance the variety and versatility of generated programs. From an application standpoint, further research is needed to explore how this approach can be applied to other language compilers.

REFERENCES

Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. 2015. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems* (Florence, Italy) *(MOBILESoft '15)*. IEEE Press, 33–43.

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1, Article 4 (2020), 36 pages. https://doi.org/10.1145/3363562

Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2021. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative about Undefined Behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1219–1223. https://doi.org/10.1145/3324884.3418933

Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3597926.3598130

Project QMK Firmware. 2023. Quantum Mechanical Keyboard Firmware. https://github.com/qmk/qmk_firmware/blob/gb_port/keyboards/bfake/matrix.c.

Project FreeBSD. 2023. FreeBSD. https://github.com/freebsd/freebsd-src/blob/stable/10/contrib/binutils/libiberty/memset.c.

Project FreeType. 2023. FreeType. https://github.com/freetype/freetype/blob/master/src/tools/ftrandom/ftrandom.c.

GNU GCC. 2023. Passes and Files of the Compiler. https://gcc.gnu.org/onlinedocs/gccint/Passes.html.

Zhendong Su's Group. 2023. EMI Compiler Testing. https://people.inf.ethz.ch/suz/emi/index.html.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2023. CompCert. https://compcert.org/index.html.

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. *SIGPLAN Not.* 50, 6 (2015), 65–76. https://doi.org/10.1145/2813885.2737986

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (2020), 25 pages. https://doi.org/10.1145/3428264

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (2023), 22 pages. https://doi.org/10.1145/3591295

LLVM. 2023a. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

LLVM. 2023b. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html.

Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121. https://doi.org/10.1145/357084.357090

Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter? *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (2019), 29 pages. https://doi.org/10.1145/3360581

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*.

Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

Project GB Studio. 2023. GBStudio. https://github.com/chrismaltby/gb-studio/blob/develop/buildTools/linux-x64/mod2gbt/mod2gbt.c.

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. *SIGPLAN Not.* 51, 10 (oct 2016), 849–863. https://doi.org/10.1145/3022671.2984038

The Clang Team. 2023. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728

Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 995–1007. https://doi.org/10.1145/3510003.3510041

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. *arXiv preprint arXiv:2308.04748* (2023).

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. https://doi.org/10.1016/j.jss.2020.110884

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s, Article 230 (2022), 36 pages. https://doi.org/10.1145/3512345