Finding Compiler Bugs via Live Code Mutation

Chengnian Sun Vu Le Zhendong Su

Department of Computer Science, University of California, Davis, USA
{cnsun, vmle, su}@ucdavis.edu

Abstract

Validating optimizing compilers is challenging because it is hard to generate *valid* test programs (*i.e.*, those that do not expose any undefined behavior). Equivalence Modulo Inputs (EMI) is an effective, promising methodology to tackle this problem. Given a test program with some inputs, EMI mutates the program to derive variants that are semantically equivalent *w.r.t.* these inputs. The state-of-the-art instantiations of EMI are Orion and Athena, both of which rely on deleting code from or inserting code into code regions that are *not* executed under the inputs. Although both have demonstrated their ability in finding many bugs in GCC and LLVM, they are still limited due to their mutation strategies that operate only on *dead* code regions.

This paper presents a novel EMI technique that allows mutation in the *entire* program (*i.e.*, both live and dead regions). By removing the restriction of mutating only the dead regions, our technique significantly increases the EMI variant space. It also helps to more thoroughly stress test compilers as compilers must optimize mutated live code, whereas mutated dead code might be eliminated. Finally, our technique also makes compiler bugs more noticeable as miscompilations on mutated dead code may not be observable.

We have realized the proposed technique in Hermes. The evaluation demonstrates Hermes's effectiveness. In 13 months, Hermes found 168 confirmed, valid bugs in GCC and LLVM, of which 132 have already been fixed.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.2 [*Programming Languages*]: Language Classifications—C; H.3.4 [*Programming Languages*]: Processors—compilers

General Terms Algorithms, Languages, Reliability, Verification

Keywords Compiler testing, miscompilation, equivalent program variants, automated testing

1. Introduction

Compilers are among the most important, widely-used system software, on which all programs depend for compilation. A bug in a compiler can have profound impact: it may lead to catastrophic consequences in safety-critical domains. Generally, compiler bugs can be categorized into two classes: crashes and miscompilations. The first class causes the compiler to crash during compilation due to either memory errors or internal assertion failures, which hinders the software development process. The second class causes the compiler to silently compile the program into wrong code. The compiler bug manifests itself indirectly as an application failure. Compared to crashes, this class is more harmful, because (1) compiler bugs are usually rare and application developers usually do not attribute the failure to the compiler, incurring extra debugging effort, and (2) such bugs can escape from in-house software testing and result in field failures.

It is critical to make compilers dependable. In the past decade, compiler verification has been an important, active area for the verification grant challenge in computing research [8]. The main compiler validation techniques include formal verification [12, 13], translation validation [22, 24], and random testing [4, 9–11, 35]. The first two categories try to produce certified compilers or optimization passes. Although promising and encouraging progress has been made (*e.g.*, CompCert [13]), it is still challenging to apply formal techniques to fully verify a production compiler, such as GCC and LLVM. Therefore, random testing remains the dominant approach in compiler validation. A notable example is Csmith [35], which generates random valid programs and relies on multiple compilers as testing oracles to detect possible discrepancies. Recently, Le *et al.* introduced Equivalence Modulo Inputs (EMI) [9] to extend the capability of existing testing approaches. EMI is a general methodology to derive semantically equivalent valid variants from existing test programs. Since the variants and the original test program are expected to behave the same *w.r.t.* some inputs, EMI only needs a single compiler to discover compiler inconsistencies.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984038

Dead-Code EMI Mutation The state-of-the-art EMI instantiations are Orion [9] and Athena [11]. Both mutate test programs by manipulating statements in dead code regions. In particular, given an existing program P and some inputs I, they profile the execution of P under the inputs I. While Orion only randomly prunes the unexecuted statements, Athena also inserts extra code into the dead code regions. Because the mutated code is never executed, the semantics of the variants is equivalent to the original program *w.r.t.* the inputs. Although Orion and Athena have been proved to be effective at finding bugs in mature production compilers such as GCC and LLVM, they are still technically limited by their mutation strategies.

First, the search space of their variant generation is limited by the size of dead code regions. Orion can only output a limited number of variants. Although Athena is able to insert extra dead code to extend the search space, the mutation is still confined within the dead code regions. Second, compilers can identify dead code regions and eliminate them during compilation. If this happens, all mutations in these regions become futile. Third, a miscompilation bug manifesting only in dead code regions is not observable as the wrong code is not executed (note that miscompilations manifest as application failures). For example, if a function with a large body is never called, then any miscompilation bug in the function body cannot be noticed. This happens for test programs that have large chunks of dead code under some inputs.

Live-Code EMI Mutation In this paper, we propose a set of novel EMI mutation strategies that address the aforementioned limitations. Our mutation strategies apply to the live code regions, while still preserving the original semantics *w.r.t.* the inputs. Our approach has the following two steps:

Profiling Stage Instead of profiling which statements are not executed as Orion and Athena do, we record more information during profiling. Given a program point l and a set of live variables V at l, we also record all the valuations of $v \in V$, *i.e.*, all the values that each v ever has during execution.

Mutation Stage Based on the valuations of V, we synthesize a code snippet c at l, such that at the exit of c the program state remains identical to the one prior to the entry of c. A simple example would be a code snippet consisting of only *nop* statements which have no side effects on the runtime. However, in order to stress test compilers, we require that the snippet make changes to the program state, and ensure the semantics preservation at the exit of c by construction.

In this paper, we synthesize the following three types of code snippets to insert into the live code regions: 1) an `if/while` statement with a non-empty randomly generated body and a contradiction predicate, 2) an `if` statement with a tautology predicate to enclose an existing statement in the live code regions, and 3) a chunk of live code mutating existing variables, which restores these values at the exit of the code snippet. We will detail them in Section 4.2.

Compared to the simple mutation strategies of Orion and Athena, the main challenge in this work is how to ensure that the statements synthesized for execution have no undefined behavior. Our key idea is to leverage the valuation of V to ensure the validity of the synthesized code. We propose a bottom-up approach to constructing valid expressions progressively (Section 4.3.2).

We have implemented the proposed technique for validating C compilers in a tool called Hermes. Our evaluation on validating two widely-used mature production compilers GCC and LLVM has strongly positive results. Within only 13 months, Hermes found 168 confirmed, valid bugs in the development trunk of GCC and LLVM. Of these bugs, 29 also affect stable releases, and 132 have already been fixed by compiler developers. Note that 29 of the reported bugs have been latent (bugs found in stable releases of production compilers), thus having slipped through traditional testing and previous compiler testing techniques (*i.e.*, Athena and Orion).

Contributions This paper makes the following main contributions:

- We propose a set of novel EMI mutation strategies, which manipulate both live and dead code regions, to overcome the limitations of existing techniques, such as Orion and Athena.
- We present a bottom-up approach to synthesizing valid (*i.e.*, undefined behavior-free) expressions by leveraging the program states of the original test program *w.r.t.* a set of inputs.
- We realize our approach as the Hermes tool for validating C compilers. Hermes is remarkably effective: it has found 12.9 bugs per month on average during our continuous 13-month evaluation as of the submission date.

Paper Organization The remainder of the paper is structured as follows. Section 2 illustrates our approach via two of our reported bugs. Section 3 briefly introduces the concept of EMI. Section 4 presents the details of our approach. Section 5 describes our extensive evaluation on GCC and LLVM. Section 6 surveys related work and Section 7 concludes.

2. Illustrative Examples

We illustrate Hermes's bug finding process via two concrete bugs: one for GCC and one for LLVM. Both examples show how we use the profiled variable valuations to synthesize code snippets and insert them into live code regions, while still preserving the EMI property. We use Csmith [35] to generate the initial testing programs, from which we derive EMI variants to stress test compilers. The Csmith-generated programs are referred to as *seed* programs in the remainder of this paper.

Because the original seeds and their bug-triggering variants are large (a few thousands lines of code), in this paper, we only show their *reduced* versions.[1]

2.1 GCC Miscompilation Bug 66186

Figure 1 shows the reduced variant which triggers a GCC miscompilation bug. The executable compiled with GCC 4.9, 5.1, or development trunk 6.0.0 at optimization levels `-O2` or `-O3` throws a segmentation fault. However, this does not conform to the semantics of this program. The `if` statement on line 10 is not executed because `a` is 0. Therefore, the program should terminate normally.

```
1   int a;
2
3   int main () {
4     int b = -1, d, e = 0, f[2] = { 0 };
5     unsigned short c = b;
6
7     for (; e < 3; e++)
8       for (d = 0; d < 2; d++)
9         /* a=0, b=-1, c=65535, d={0,1}, e={0,1,2}, f[0]=0 */
10        if (a < 0) // Inserted code highlighted in gray.
11          for (d = 0; d < 2; d++)
12            if (f[c])
13              break;
14    return 0;
15  }
```

Figure 1: GCC 4.9, 5.1, and development trunk (6.0.0 rev 223265) miscompile the variant at -O3. (`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66186`). The compiled binary throws segmentation fault instead of terminating normally.

The code snippet inserted by Hermes is highlighted in gray between lines 10 and 13. It is synthesized as follows:

1. Hermes profiles the execution of the seed program (*i.e.*, the program in Figure 1 excluding the highlighted code) to determine the possible values of in-scope variables at all execution points. For instance, line 9 shows the observed values of all variables in scope at this location.
2. Hermes synthesizes a statement that does not have any side effect to the program (in this case it is an `if` statement where the guard is `false`[2]), and inserts the statement into the location.

The bug occurs because GCC incorrectly assumes that accesses to stack variables (in this case, access to `f[c]` on line 12) are non-trapping, and therefore hoists `f[c]` out of the `if` statement on line 10, making it unconditionally executed. However, this array access is out-of-bounds as `c` is 65535, and should not be executed as it is in the dead code region. GCC developers have confirmed and fixed this bug. Note that existing tools such as Orion [9] and Athena [11] cannot reveal this bug because the mutation happens in the *live* region of the program.

2.2 LLVM Miscompilation Bug 26266

```
1   extern void abort();
2   char a;
3   int b, c = 9, d, e;
4   void fn1() {
5     unsigned f = 1;
6     int g = 8, h = 5;
7     for (; a != 6; a--) {
8       int *i = &h, *j;
9       for (;;) {
10        /* g = 8, h = 5 */
11        int k = e; // Inserted code highlighted in gray.
12        int l = ~1;
13        if (g && h) {
14          k = g;
15          l = f;
16          f = -(~(c && b) | -~(e * ~l));
17          if (c < f)
18            abort();
19        }
20        g = k;
21        f = l;
22        if (d <= 8)
23          break;
24        *i = 0;
25        for (; *j <= 0;)
26          ;
27      }
28    }
29  }
30  int main() {
31    fn1();
32    return 0;
33  }
```

Figure 2: LLVM development trunk (3.9.0 rev 258508) miscompile the variant at -O1 and above. (`https://llvm.org/bugs/show_bug.cgi?id=25154`). The compiled binary aborts instead of terminating normally.

Figure 2 shows a reduced variant that triggers a miscompilation bug of Clang. Initially, Clang compiles the seed program correctly at all optimization levels — all compiled executables terminate normally. However, after Hermes inserts the *live* `code snippet` (between lines 11 and 21 highlighted in gray) on line 11, Clang trunk 3.9 miscompiles the variant: the compiled binary aborts on line 18, a program point which should not be reachable.

Different from the GCC example above where Hermes only generates an `if` statement whose guard is `false`, in this example, Hermes synthesizes a chunk of code which is

[1] A reduced version is derived from the bug-triggering test program by removing bug-irrelevant code fragments in order to help developers understand and diagnose the bug.

[2] The global variable `a` is not explicitly initialized, and according to the C language standard it is implicitly set to 0 by default. Thus the conditional is always false.

executed at runtime. By instrumenting the program, Hermes knows that the values of `g` and `h` on line 10 are `8` and `5` respectively. Then it synthesizes a block of live code to mutate `f`. Specifically, we use `g` and `h` to construct a true `if` statement on line 13. In the body, we first save the value of `f` on line 15, later change its value and use the new value (lines 17). This live code snippet is generated by our mutation strategy *Always True Conditional Block* which will be detailed in Subsection 4.2.

This bug is in the demanded-bits analysis in Clang. When computing the demanded bits for the comparison `c < f` on line 17, Clang should consider both operands `c` and `f`. However, in the buggy version of this analysis, the first operand is accidentally ignored, leading the compiler to conclude that this comparison `c < f` is always true (actually it should be always false), and consequently making the `abort()` unconditionally executed. Again, this bug cannot be triggered by Orion and Athena because the insertion happens in the live region.

3. Equivalence Modulo Inputs

This section briefly introduces the formal definition of Equivalence Modulo Inputs (EMI) [9]. Let $\mathcal{L}$ denote a generic programming language $\mathcal{L}$ with deterministic semantics $[\![\cdot]\!]$. Repeated executions of a program $P \in \mathcal{L}$ on the same input i[3] always yield the same output $[\![P]\!](i)$.

Given two programs $P, Q \in \mathcal{L}$ and a set of common inputs I (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$[4]), P and Q are equivalent modulo inputs *w.r.t.* I (denoted as $[\![P]\!] =_I [\![Q]\!]$) iff

$$\forall i \in I \quad [\![P]\!](i) = [\![Q]\!](i).$$

Given a program $P \in \mathcal{L}$, any input set $I \in \mathsf{dom}(P)$ naturally induces a collection of programs $Q \in \mathcal{L}$ such that $[\![P]\!] =_I [\![Q]\!]$. We refer to this collection as P's EMI variants. EMI relaxes the notion of program equivalence, and provides a practical way to generate test programs from existing code for compiler testing.

The state-of-the-art instantiations of EMI are Orion and Athena. Both operate on unexecuted code regions. Given a program P and its inputs I, Orion and Athena first profile the execution of P and identify the dead code regions. Then they delete statements from or insert additional statements into the dead code regions. As these modified regions remain unexecuted under the same inputs I, the mutated programs are EMI variants of P.

The main benefit of EMI is the preservation of the validity of P *w.r.t.* I after mutation. Another attractive property of EMI is that it does not need a reference compiler for differential testing, because EMI provides an explicit testing oracle, that is, all EMI variants are expected to output the same result *w.r.t.* the same input. In contrast, other approaches such as Csmith [35] relies on an extra reference compiler to cross-check the consistency of the compiled code, as the semantics of the randomly generated test programs is unknown.

Without loss of generality, we assume that the input set I has only one element in order to facilitate the presentation. Thus in the remainder of this paper, we use I to represent a single input, rather than a set of inputs.

4. Approach

Algorithm 1 shows the general process of using EMI. Given a compiler under test, we first compute the testing oracle O on line 2 by caching the output of compiling and running the test program P. In each of the main iterations on line 4, we generate an EMI variant P', and compare the output of compiling and running P' (*i.e.*, O') with the oracle O.

Algorithm 1: Hermes's process for testing compilers

```
1  procedure Test(Compiler $C$, Program $P$, Input $I$):
2      $O \leftarrow C$.Compile($P$).Execute($I$)            // oracle
3      $P' \leftarrow P$                                     // initialization
4      for $i \leftarrow 1$ to MAX_ITER do
           // create a random EMI variant
5          $P' \leftarrow$ EMI($P', I$)
            $O' \leftarrow C$.Compile($P'$).Execute($I$)
6          if $O' \neq O$ then                            // inconsistent outputs
7              ReportBug($C, P'$)

8  function EMI (Program $P$, Input $I$):
9      $p \leftarrow$ Profile($P, I$)
10     $P' \leftarrow P$                                     // initialization
11     foreach $s \in P'$.Statements() do
12         Mutate($P', s, p$)
13     return $P'$

14 function Mutate (Program $P'$, Statement $s$, Profile $p$):
15     $env \leftarrow p$(Loc($P', s$))
16     if dom($env$) $\neq \varnothing$ and FlipCoin() then
           // SynCode randomly selects FCB, TG or TCB
17         $code \leftarrow$ SynCode($env$)
18         $P'$.InsertBefore($code, s$)
19     foreach $c \in s$.ChildStatements() do
20         Mutate($P', c, p$)
```

4.1 Program Profiling

Our EMI variant generation (the function `EMI` on line 8 in Algorithm 1) starts on line 9 with profiling the test program to record necessary program states. Our profiling schema is more sophisticated than the one used in both Orion and Athena. While Orion and Athena only need the coverage information to identify dead code regions, our approach, in addition to the coverage information valuation of variables

[3] For a closed program (*e.g.*, a Csmith-generated program), we assume that it only has one type of inputs, *i.e.*, $\varnothing$.

[4] $\mathsf{dom}(f)$ denotes the domain the function f.

at certain program points so that we can mutate the live code safely without introducing any undefined behavior. We formally define the execution profile as follows:

Definition 4.1 (Execution Profile)**.** *Given a program P and an input I, the execution profile of running P with input I records all the values observed at runtime for each variable at each program point. Let $\widehat{Var}$ denote all the in-scope variables (including fields of structures and elements of arrays), $\mathbb{Z}$ be the integers, and $\widehat{Loc}$ denote the program points in P. The profile $\widehat{Profile}$ is defined as a mapping from a program point $l \in \widehat{Loc}$ to an environment $e \in \widehat{Env}$, which maps a variable $v \in \widehat{Var}$ to the set of its observed values.*

$$\widehat{Profile} = \widehat{Loc} \to \widehat{Env} \qquad \text{(Profile)}$$

$$\widehat{Env} = \widehat{Var} \to \mathcal{P}(\mathbb{Z}) \qquad \text{(Environment)}$$

Take Figure 1 as an example. After running the program, we collect a *profile*, which associates line 10 with the following environment:

$$\mathit{profile}(10) = \left\{ \begin{array}{ll} a \to \{0\} & d \to \{0,1\} \\ b \to \{-1\} & e \to \{0,1,2\} \\ c \to \{65535\} & f[0] \to \{0\} \end{array} \right\}$$

Figure 3: The environment on line 10 in Figure 1

For a program point l in dead code regions, the profile has an empty environment, namely, $\mathit{profile}(l) = \varnothing$. Note that in this paper, we only use integer values (*i.e.*, $\mathbb{Z}$) in execution profiles. This is mainly due to the inherent inaccuracy of floating-point arithmetic, which makes differential testing of compilers intractable [9, 11, 35]. In other words, it is difficult to predict the results of comparisons on close floating point values. We leave this as future work.

4.2 Live Code EMI Mutation

Our approach relies on mutating live code regions in a test program. Given a test program P, its inputs I, and a statement s executed by running P with I, we formally define the mutation as a program synthesis problem as follows.

Definition 4.2 (Live Code EMI Mutation)**.** *Let C be a code snippet to be inserted right before s, resulting in a variant program P'. C is an EMI mutation in P' if when P' is executed with I the program state before the entry of C and the program state after the exit of C are the same.*

To stress test compilers, the code snippet C should have side effects, otherwise a compiler may be smart enough to safely remove C from P'. The extra side effects complicate data- and control-dependencies so that compilers may optimize P' differently from P.

After obtaining a profile, we randomly insert a code snippet into a live code region. Specifically, on line 16 in Algorithm 1, if the statement s is executed (by checking whether its associated environment is not empty, *i.e.*, $\mathsf{dom}(\mathit{env}) \neq \varnothing$) and `FlipCoin` returns true,[5] we then randomly synthesize a code snippet via calling `SynCode`() and insert the snippet into the program right before s on line 17. This procedure is repeatedly applied to the child statements of s on line 20.

We have designed the following strategies in `SynCode`() to synthesize live code EMI snippets. In each invocation to `SynCode`(), the following strategies are randomly selected with the same probability, *i.e.*, 1/3 each.

Always False Conditional Block (FCB) We generate an `if`/`while` statement, of which the body is not empty and the conditional predicate of this statement is always evaluated to false under the program input I. We synthesize the false predicate based on the variable valuations in the environment at the program point.

The body is randomly generated by unrolling the C grammar. Concretely, during code generation, we randomly pick a production rule from the grammar, and instantiate it by recursively instantiating its sub-production rules with the in-scope variables available at the program point. We try to reuse the existing variables as much as possible in order to maximize the data dependencies between the code in the seed program and the synthesized code snippet. To limit the size of the generated code, we set an upper bound of the depth for derivation of production rules.

```
10  if (a < 0) // inserted code
11    for (d = 0; d < 2; d++)
12      if (f[c])
13        break;
```

For example, the above code snippet (extracted from lines 10–13 of Figure 1) is synthesized based on the environment in Figure 3. As `a` is `0` at runtime, the predicate `(a < 0)` is false, and the body will not be executed. The body is synthesized by instantiating the production rule of the `for` loop, which requires instantiations of four more production rules (*e.g.*, the loop body, initializer, condition) and uses three existing variables `d`, `f` and `c`. Note that since the body is not executed, we do not need to ensure the validity of the code during program generation.

Always True Guard (TG) For an existing executed statement s in the original program, we introduce an `if` statement to guard s, of which the predicate p is always true, (*i.e.*, `if (p) s;`). This strategy alters the control flow but still preserves the original semantics.

Always True Conditional Block (TCB) We synthesize an `if` statement with a non-empty body with side effects, where the guard is always true based on the environment. This is the opposite of **FCB** because the body of this statement will be executed. Therefore, we need to ensure that the body does not exhibit any undefined behavior (*i.e.*, well-defined *w.r.t.*

[5] `FlipCoin` returns true with a tuned probability. In a different context, the probability can be different.

```
1 int backup_v = ⟨synthesized valid expression⟩;
2 if (⟨synthesized true predicate⟩) {
3   backup_v = v;
4   v = ⟨synthesized valid expression⟩;
5   if/while(⟨synthesized false predicate⟩) {
6     print v;
7   }
8 }
9 v = backup_v;
```

Figure 4: Skeleton to synthesize TCB with one variable v. The highlighted text will be replaced by synthesized expressions and predicates.

to the C standard). In order to stress the compiler, the body should have side effects on the program state. But these side effects should be reverted at the exit of the TCB block to maintain the EMI property.

The process of generating TCB snippets takes as input a set of in-scope variables. Without loss of generality, Figure 4 only takes a single variable v of type `int`. It is straightforward to extend the template to multiple variables of different types.

In general, the template first creates a backup variable to store the value of v on line 3, then changes v on line 4. The print statement '`print v`' on line 5 ensures that the new value of v is used, so that the compiler cannot optimize away the synthesized code. Finally, the template restores the variable v to its original value on line 9. This template does not change program state after it is fully executed, and therefore preserves the EMI property.

4.3 Predicate, Expression and TCB Synthesis

This subsection describes the building blocks of realizing our three mutation strategies: synthesizing a predicate with a given truth value, and synthesizing an expression without undefined behavior for TCB. We also discuss how we generate a TCB block via speculative execution.

4.3.1 Predicate Synthesis

Algorithm 2 presents the process to generate a predicate with a given truth value. The parameter depth is used to limit the size of the generated predicate.

A predicate is built top-down. At the top we first randomly choose a logical operator (*i.e.*, conjunction, disjunction, and negation), then carefully randomize and compute the target truth values of the children so that the whole predicate evaluates to the expected truth value. We then proceed to build the children. Specifically in Algorithm 2, the function `SynNeg` synthesizes a negation predicate, `SynCon` generates a conjunction, and `SynDis` generates a disjunction.

An atomic predicate (one with depth 0) checks the relation between a variable and a constant, or between two variables. It is constructed with a set of rules (*i.e.*, the function `SynAtom`). Specifically, as we know all values of each variable, we can create tautologies or contradictions by construction by selecting the right constants or relational operators. For example, in Figure 1, because a is 0 on line 10, we can create a contradiction $a < 0$, $a > 0$, $a < d$ and many others.

Algorithm 2: Synthesize a predicate

```
function SynPred (Env env, Bool expected, Int depth):
    if depth = 0 then
        return SynAtom(env, expected)
    switch Random(4) do
        /* synthesize a negation */
        case 1 do return SynNeg(env, expected, depth)
        /* synthesize a conjunctive predicate */
        case 2 do return SynCon(env, expected, depth)
        /* synthesize a disjunctive predicate */
        case 3 do return SynDis(env, expected, depth)
        /* synthesize an atomic predicate */
        case 4 do return SynAtom(env, expected)
function SynNeg (Env env, Bool expected, Int depth):
    return Expr('!', SynPred(env, !expected, depth − 1))
function SynCon (Env env, Bool expected, Int depth):
    if expected then left ← true, right ← true
    else if FlipCoin() then left ← true, right ← false
    else left ← false, right ← FlipCoin()
    left_pred ← SynPred(env, left, depth − 1)
    right_pred ← SynPred(env, right, depth − 1)
    return Expr('&&', left_pred, right_pred)
function SynDis (Env env, Bool expected, Int depth):
    if ! expected then left ← false, right ← false
    else if FlipCoin() then left ← false, right ← true
    else left ← true, right ← FlipCoin()
    left_pred ← SynPred(env, left, depth − 1)
    right_pred ← SynPred(env, right, depth − 1)
    return Expr('||', left_pred, right_pred)
function SynAtom (Env env, Bool expected):
    /* Rule 1: randomly pick one variable v and
       construct a relational predicate with expected
       truth value over v and a constant */
    /* Rule 2: randomly pick two variables v1 and v2,
       and construct a relational predicate with
       expected truth value over v1 and v2 */
```

4.3.2 Bottom-Up Valid Expression Synthesis

In order to realize the TCB mutation strategy, we need to synthesize valid expressions without undefined behavior, a challenging synthesis problem in compiler testing for decades. In this paper, we leverage the fact that we know the values of all variables at runtime *w.r.t.* the program inputs I, and propose a bottom-up expression building algorithm that safely avoids undefined behaviors. Figure 5 shows the grammar of expressions we support in this work.

```
⟨expr⟩    ::= ⟨c⟩ | ⟨v⟩ | '(' ⟨expr⟩ ')' | ⟨uop⟩ ⟨expr⟩
           |  ⟨expr⟩ ⟨bop⟩ ⟨expr⟩
⟨c⟩       ::= ℤ
⟨v⟩       ::= variables
⟨uop⟩     ::= '!' | '~' | '-'
⟨bop⟩     ::= '+' | '-' | '*' | '/' | '%' | '<<' | '>>' | '&' | '|' | '^'
           |  '>' | '>=' | '==' | '!=' | '<=' | '<'
```

Figure 5: BNF Grammar of Synthesized Expressions

Algorithm 3: Synthesize valid expressions

```
1  function SynExpr (Env env):
2      worklist ← Sample(env, Random(|dom(env)|))
3      while |worklist| $> 1$ do
4          if FlipCoin() then                        // unary expression
5              $v$ ← Sample(worklist, 1)
6              uop_list ← a shuffled list of unary operators
7              foreach $uop \in$ uop_list do
8                  if IsUndefined(env, $v$, $uop$) then
9                      continue
10                 worklist ← (worklist \ $\{v\}$)
11                 worklist ← worklist ∪ {Expr($uop, v$)}
12                 break
13         else                                      // binary expression
14             $\{u, v\}$ ← Sample(worklist, 2)
15             bop_list ← a shuffled list binary operators
16             foreach $bop \in$ bop_list do
17                 if IsUndefined(env, $u, v, bop$) then
18                     continue
19                 worklist ← (worklist \ $\{u, v\}$)
20                 worklist ← worklist ∪ {Expr($bop, u, v$)}
21                 break
22     return the only expression in worklist
```

Algorithm 3 shows the process to build valid expressions with unary and binary operators. The variable worklist stores a set of valid expressions, which are gradually merged together with various operators and form a valid single compound expression.

The function Sample samples elements from a set without replacement. Initially, we sample a random number of variables from env on line 2 and use them as the leaves of the expression to build. We then make a random choice to generate either a unary or a binary expression on line 4.

Building Binary Expressions If we choose to construct a binary expression, we first sample two elements $\{u, v\}$ from worklist as child expressions on line 14, then iterate through all the available binary operators in a randomized order until we find an operator *bop* that does not introduce undefined behaviors according to the valuations of u and v. Note that some operators are always well defined independently of their arguments according to the C language standard (*e.g.*, negation (!), bit and/or/xor, and relational operators), so this step will always succeed. The algorithm removes u and v from worklist and adds the new binary expression Expr(bop, u, v) to worklist.

Building Unary Expressions This process is similar to constructing binary expressions. However, because building unary expressions does not shrink worklist, our algorithm may not terminate. To deal with this problem, a counter records the number of consecutive unary constructions (This counter is not shown in Algorithm 3 for clear presentation of the algorithm). If a threshold is reached, we force the algorithm to build binary expressions in the next iteration.

Checking Undefined Behavior To evaluate the validity of an expression e, we need all the possible values of its child expression(s). We perform a speculative execution *Spex*(env, e) over e with the given environment env to compute an over-approximation of the environment as follows:

- $v \in \widehat{Var}$: If e is a variable expression v, its values can be obtained from env, *i.e.*,

$$Spex(\mathsf{env}, e) = \mathsf{env}(v)$$

- Expr(uop, e'): If e is a unary expression, its values are computed by applying *uop* to each value of its child expression e', *i.e.*,

$$Spex(\mathsf{env}, e) = \{uop\ x | x \in Spex(\mathsf{env}, e_1)\}$$

- Expr(bop, e_1, e_2): If e is a binary expression, its values are the application of *bop* to the elements of the Cartesian product of *Spex*(env, e_1) and *Spex*(env, e_2), namely,

$$\{x_1\ bop\ x_2 | (x_1, x_2) \in Spex(\mathsf{env}, e_1) \times Spex(\mathsf{env}, e_2)\}$$

To check for undefined behaviors, we emulate the semantics of unary and binary operators defined in the C language standard over all the possible values of child expressions. If no values trigger undefined behaviors, the function IsUndefined returns false.

Theoretically, it is possible that *Spex* can become very large for expressions with many nested binary operators. In practice, when synthesizing an expression, we limit the depth of expressions, and prefer variables with a few values as expression leaves over ones with a large number of values.

4.4 TCB Synthesis with Speculative Execution

Hermes generates TCB code snippets by instantiating the template in Figure 4. Specifically, it replaces the highlighted text in Figure 4 with synthesized expressions and predicates by calling SynExpr and SynPred, as shown in Figure 6.

Both functions SynExpr and SynPred require the environment of the insertion point (*i.e.*, the program point at which the synthesized code will be inserted). However, because the whole TCB code snippet is statically synthesized (not executed yet), the environment of each program point in the template except the entry is $\varnothing$. The environment of the entry env is the one associated to the insertion point of the whole TCB code in the original test program, which is accessible from the execution profile.

Similar to checking undefined behaviors, we perform speculative execution to over-approximate environments for each program point of a TCB code block. Figure 6 shows how we instantiate the TCB template. The template is interpreted line by line from top to bottom. For each line, the plain text will be output directly as a part of the final code snippet. The highlighted text within ⟨⟩ will be executed first and its result will be a part of the final code snippet.

The environment env is updated along the live path of the template. After each assignment to a variable, we update env with the new valuation. For example, after creating a new variable backup_v and initializing it with a random expression on line 2, we update env by incorporating backup_v and its valuation on line 3. The updated env is used later to synthesize other predicates and expressions.

```
1  // env is obtained from the execution profile
2  int backup_v = ⟨SynExpr(env)⟩
3  ⟨add backup_v and its valuation to env⟩
4  if (⟨SynPred(env, true, depth)⟩) {
5    backup_v = v;
6    ⟨update backup_v in env with v's valuation⟩
7    v = ⟨SynExpr(env)⟩;
8    ⟨update v in env with its new valuation⟩
9    if/while(⟨SynPred(env, false, depth)⟩) {
10     print v;
11   }
12 }
13 v = backup_v;
```

Figure 6: Synthesizing TCB with speculative execution

4.5 Implementation

We have implemented the proposed technique in Hermes for testing C compilers. Hermes uses Clang's LibTooling library [30] to instrument profiling code into the program to obtain the execution profile, and to mutate the program based on the obtained profile.

In addition to the algorithmic complexity, we also face the challenge of handling the intricate type conversions defined in the C language when performing speculative execution. This is crucial as any inconsistency with the C standard will lead the synthesized code to deviate from our expectation, which breaks the mutated program's EMI property. For example, given two variables 'int a=0' and 'short b=-1', the sum 'a+b' will be -1. However if the type of a is unsigned, then 'a+b' will be the maximum value of unsigned type (4294967295 if unsigned has 32 bits) as b is converted to unsigned first.

Therefore, although our speculative execution operates on concrete values in execution profiles, we also need to take their types into consideration when performing unary and binary operations. Hermes carefully implements these conversions including integer promotion, usual arithmetic conversion and implicit conversion in C. Note that an alternative to implement *Spex* is creating tiny C programs with the desired fragments of code and running them. However, this alternative may incur significant overhead as it involves very frequent compilation, execution, and process-level communication, compared to our current implementation by simulating the semantics of the C language.

4.5.1 Sparse Profiling

Profiling all program points to collect the valuations of all in-scope variables is usually not feasible or practical, as it can consume much time and huge storage space. In the following, we describe an optimization technique to reduce the profiling overhead.

The optimization is based on an insight that not all collected environments in the profile are used for EMI mutation. Note line 16 in Algorithm 1. For each program point with non-empty environment (checked by $\mathsf{dom}(env) \neq \varnothing$), Hermes makes a random decision to mutate this program point (determined by FlipCoin()). If Hermes decides to skip this point (*i.e.*, FlipCoin() returns false), then the effort to collect the environment for this point is wasted.

Based on this observation, we propose a sparse profiling schema by moving the non-determinism of code synthesis at the mutation stage to the profiling stage. Let s denote all the eligible program points for profiling. Instead of instrumenting profiling code into each point in s, we introduce a profiling probability $\mathcal{P}_{profile}$, and sample a subset $s' \subseteq s$ with $\mathcal{P}_{profile}$ for profiling. Concretely, when we visit each point in s, we randomly select the point and instrument the profiling code with the probability $\mathcal{P}_{profile}$.

Next on line 16 in Algorithm 1, we remove the call to FlipCoin(), and deterministically synthesize a code fragment for the program point l if l has a non-empty environment.

5. Evaluation

This section discusses our testing efforts over 13 months, from middle May 2015 to June 2016. We focus on testing two open-source compilers GCC and LLVM because of their openness in tracking and fixing bugs. Some highlights of this process follows:

- **Many detected bugs:** In 13 months, we have reported 168 new and valid bugs. Developers have confirmed all these bugs and fixed 132 of them.
- **Many long-latent bugs:** We have found 29 latent bugs in stable releases of GCC and LLVM. These bugs have slipped through traditional testing and previous compiler testing techniques, *e.g.*, Athena.

Developers are generally responsive in confirming and fixing our bugs, which is a strong indication that they take our bugs seriously. To quote one developer:

> *"Wow, this was a *horrible* failure. Thanks for the testcase! We were invalidating a cached reference under our own feet."* [6]

5.1 Testing Setup

We ran Hermes on two machines (one has 18 cores and the other 6 cores) running Ubuntu 14.04 (x86_64).

Compiler Versions We built the development trunks of GCC and LLVM daily and tested them on their five standard options, "-O0", "-O1", "-Os", "-O2" and "-O3". The reason for testing development trunks is mainly that developers fix bugs primarily in truck revisions rather than stable releases. Fixes for bugs in stable releases are only available much later in subsequence stable releases. This long gap between when a bug is found and when the fix is released makes it difficult to identify whether a newly found bug is a duplicate to an existing unfixed bug. We also lose the opportunity of finding bugs in the trunk as early as possible if we only focus on stable releases.

Seed Programs In this paper, we use Csmith [35] as the seed program generator, because Csmith-generated programs can be effectively reduced using existing reduction tools such as Berkeley's Delta [19] and CReduce [25].

Note that Hermes is orthogonal to and independent of the seed programs. We can use the vastly available open-source code as seeds. However, the major obstacle of doing this is that those programs are usually large and consist of multiple files, and thus cannot be effectively reduced by the state-of-the-art reduction tools.

Testing Process Our testing process is fully automated and runs continuously. In each iteration, we first use Csmith to generate a seed program. We then apply Hermes to derive 10 variants from the seed, and use then to test GCC and LLVM.

This process involves little human intervention. In addition to test program generation, we have also automated the test case reduction (using Delta and CReduce). The only manual effort is to confirm that the reduced test programs are indeed valid and report them. Concretely, if the compiler bug is miscompilation, we ensure that the test program does not contain undefined behavior. Moreover, we check whether the new bug is a duplicate to any existing bugs in compilers' Bugzilla databases.

Profiling and Synthesis Parameters We set the profiling probability $\mathcal{P}_{profile}$ to 0.1 to perform sparse profiling (*cf.* Section 4.5.1). The Csmith-generated programs usually have around 1000 ~ 2000 lines of code, so the profiler will select around 100 ~ 200 program points to instrument. Note that some of these instrumented points might be in dead code regions, hence the final number of executed profiling points can be fewer, not zero according to our experiences.

	GCC	LLVM	TOTAL
Fixed	85	47	132
Not-Yet-Fixed	10	26	36
Duplicate	28	20	48
Invalid	1	0	1
TOTAL	124	93	217

Table 1: Reported bugs.

When synthesizing a predicate, we set its maximum depth to 2. In detail, before each call to `SynPred` in Algorithm 2, we randomize a number $d \in [0, 2]$ and pass d as the argument depth.

5.2 Quantitative Results

We next present some statistics on our reported bugs.

Bug Count Table 1 summarizes our bug results. In 13 months, we reported in total 217 bugs (124 in GCC and 93 in LLVM). Developers confirmed 168 valid bugs and fixed 132 of them; 49 have yet to be confirmed as they were reported very recently.

Not-Yet-Fixed Bugs All these bugs were reported recently. Developers have confirmed these bugs, and are discussing how to fix them.

Duplicate Bugs Although we only considered bugs that have different symptoms from that of the not-yet-fixed bugs, we reported 48 duplicated bugs (all are GCC bugs) during this testing period. These bugs turned out to have the same root cause.

Invalid Bugs We reported one invalid bug in GCC, in which the variant has an undefined behavior, in particular, sequence point violation [34]. This was caused by a bug in our Hermes tool, which we later fixed.

Bug Type Compiler bugs can be generally classified into the following three categories.

Miscompilation The compiler silently miscompiles the program by producing a wrong executable that alters the semantics of the source program.

Crashing The compiler crashes when compiling the program, due to either runtime errors (*e.g.*, segmentation fault, floating-point exceptions) or assertion failures.

Performance The compiler hangs or takes a very long time to compile the program.

Miscompilation bugs are run-time bugs because they manifest when the compiled binaries are executed. Crashing and performance bugs are compile-time bugs because they manifest during compilation.

Table 2 partitions our 168 confirmed and valid bugs into the above three categories. The category *performance* represents bugs that cause the compiler to not terminate within a

[6] `https://llvm.org/bugs/show_bug.cgi?id=25225`

certain amount of time when compiling a program (five minutes in our experiments). As the table shows, nearly half of our bugs are miscompilation, the most serious kind.

	GCC	LLVM	TOTAL
Miscompilation	42	34	76
Crash	52	38	90
Performance	1	1	2

Table 2: Bug classification.

Mutation Cost and Throughput Given a seed program, the time cost of Hermes to derive an EMI variant is low. Thanks to the sparse profiling and efficient implementation of the code synthesizer, the time is negligible. On average, it took Hermes only 1.7 seconds to complete both the profiling and code synthesis. Table 3 shows the statistics of the time cost.

In the setting of a single run (a single thread on a single CPU core), our testing process can test around 400 programs per hour. Most of the CPU time is taken up by the compilation by GCC and LLVM at different optimization levels. The throughputs of Hermes and Athena are similar, as the mutation overhead of Hermes is quite small although Hermes has an extra step of profiling.

Min	Mean	Median	Max	StdDev
0	1.7	1	6	1.2

Table 3: Time of Generating an EMI variant (seconds).

5.3 Comparison with Athena

This section presents our in-depth comparison evaluation of Hermes and Athena. In general, we performed the comparison in three different ways: 1) running Athena concurrently with Hermes, 2) running Athena with the same seeds to re-trigger the bugs found by Hermes in Table 4, and 3) measuring the covered lines of compiler code. These three evaluations complement each other, and together offer a comprehensive comparison between Athena and Hermes. Since Le *et al.* has shown in [11] that Athena outperforms Orion at finding bugs in GCC and LLVM, we exclude Orion from this study.

5.3.1 Effectiveness

Based on the data reported in [11], Athena took 19 months to find 72 confirmed bugs, whereas Hermes has already found 168 confirmed bugs within only 13 months. Moreover, since last May, we have launched a parallel run of Athena along with Hermes. During the same period, Athena has found only a few bugs.

We have also conducted an evaluation to compare Hermes and Athena directly. In particular, we selected all the confirmed bugs (28 bugs in total) that were found by Hermes and reported in November and December 2015, and tried to re-trigger them using Athena with the same seed programs used by Hermes. For each bug, we used Athena to derive a sequence of variants from the same seed program. Considering the stochastic characteristics of Athena, we set the number of variants per seed to 200 (in comparison to 10 variants per seed of Hermes), and repeated the generation process 100 times. Table 4 lists the results. Only three bugs (GCC-68327, GCC-68870, and LLVM-25538) can be re-triggered by Athena.

	BUG ID	Type	Optimization	Athena
1	GCC-68194	Mis.	-Os,-O2,-O3	
2	GCC-68240	Crash	-O2, -O3	
3	GCC-68248	Crash	-O2, -O3	
4	GCC-68285	Mis.	-O3	
5	GCC-68305	Crash	-O1,-Os,-O2,-O3	
6	GCC-68327	Crash	-O3	✓
7	GCC-68376	Mis.	-O1,-Os,-O2,-O3	
8	GCC-68506	Mis.	-O3	
9	GCC-68520	Crash	-O3	
10	GCC-68570	Crash	-O1,-Os,-O2,-O3	
11	GCC-68624	Mis.	-O2,-O3	
12	GCC-68691	Crash	-O3	
13	GCC-68721	Mis.	-Os,-O2,-O3	✓
14	GCC-68730	Mis.	-O3	
15	GCC-68906	Crash	-O3	
16	GCC-68909	Crash	-O3	
17	GCC-68911	Mis.	-O3	
18	GCC-68951	Mis.	-O3	
19	GCC-68955	Mis.	-O3	
20	GCC-69030	Crash	-O2,-O3	
21	GCC-69083	Crash	-O3	
22	GCC-69097	Wrong	-O1,-Os,-O2,-O3	
23	LLVM-25372	Crash	-O1,-Os,-O2	
24	LLVM-25538	Crash	-O1,-Os,-O2	✓
25	LLVM-25629	Mis.	-O1,-Os,-O2,-O3	
26	LLVM-25754	Crash	-O1,-Os,-O2,-O3	
27	LLVM-25900	Wrong	-O1,-Os,-O2,-O3	
28	LLVM-25988	Wrong	-O2,-O3	

Table 4: The results of running Athena to re-trigger the confirmed bugs that were reported in November 2015. The first column shows the bug ID. The second column is the type of the bug, either miscompilation or crash. The third column lists the optimization levels to trigger the bug, and the forth shows the reproduction result.

The various comparisons above demonstrate that the mutation strategy of Hermes is more effective at finding bugs than that of Athena.

5.3.2 Line Coverage

We now evaluate the line coverage improvement of Hermes on GCC and LLVM compared to Athena. The baseline is the coverage of compiling 100 Csmith-randomly-generated

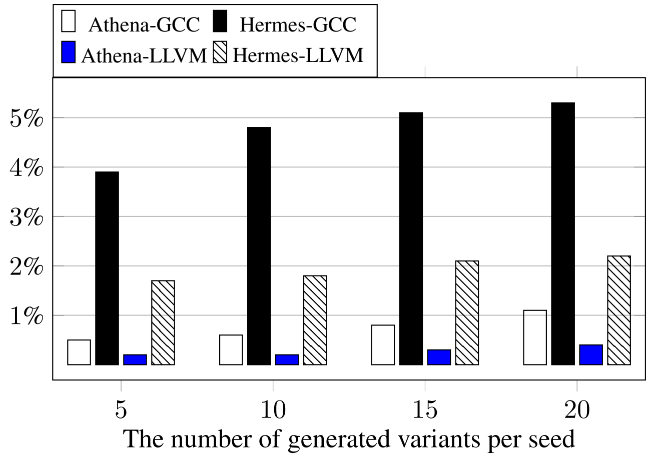

Figure 7: Improvement in line coverage of Athena and Hermes. The baseline is the coverage of compiling 100 seed programs by GCC and LLVM (34% coverage ratio for GCC and 21.1% for LLVM). The x-axis is the number of generated variants per seed program, and the y-axis is the absolute coverage ration improvement over the baseline.

seed programs — the coverage ratio of GCC is 34% and that of LLVM is 21.1%.

We then use Athena and Hermes to derive EMI variants from these 100 test programs, compile the variants with GCC and LLVM, and lastly measure the coverage of compilers. We vary the number of variants to generate from each seed program, and obtain the coverage data shown in Figure 7. The x-axis is the number of generated variants per seed program, and the y-axis is the absolute coverage ratio improvement over the baseline.

Figure 7 demonstrates that Hermes significantly improves line coverage of both GCC and LLVM over Athena. Take the second group of vertical bars at `x=10` as an example. Hermes improved the coverage over the baseline by 4.8% (24,612 lines) for GCC and 1.8% (12,672 lines) for LLVM, while Athena did by 0.6% (3,111 lines) for GCC and 0.2% (1,753 lines) for LLVM. The additionally covered code is mainly in the middle-end and the back-end of each compiler.

5.4 Comparison of Mutation Strategies

This section presents the comparison of the three mutation strategies (*i.e.*, FCB, TG, TCB) in terms of numbers of bugs detected.

We use the 28 bugs in Table 4 as samples for this analysis. Each bug-revealing test program P' is derived from the seed program P by injecting a set S of code snippets, of which each is synthesized by a strategy of Hermes. Then we iteratively remove elements from S until we obtain a minimal $S' \subseteq S$ such that $P + S'$ (*i.e.*, an EMI variant by inserting the mutation code S' into P) still trigger the same bug.

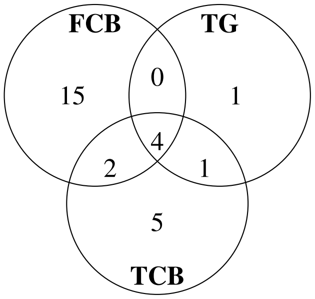

Figure 8: Distribution of bugs found by different mutation strategies. Intersection represents the number of bugs found by the combination of strategies. For example, the intersection of the three circles means that four bugs are found by a set of code snippets synthesized by FCB, TG and TCB together.

Then we classify S' based on the mutation strategies that synthesize the elements in S'. The result is shown in Figure 8 as a Venn diagram. Each single strategy can find bugs alone. But there are two bugs found by the combination of FCB and TCB, one bug by the combination of TG and TCB, and four bugs by the three strategies together.

5.5 Sample Bugs

This section illustrates the diversity of our bugs via six bug samples. Note that all these bugs are reduced from (much) larger mutated programs.

Figure 9a The expected behavior of this program is to print 0 (the value of `d` assigned inside the loop). Note that because the conditional statement at line 13 is not executed (`b` is 0), none of its printf statements are executed.

However, the program compiled with GCC trunk prints 1. The reason is that GCC decides to perform loop unswitching [33] to move the conditional at line 15 outside the loop. This optimization is incorrect because the now-used local variable `j` was not initialized: the optimized program contains undefined behavior. Note that the original program is valid (no undefined behavior) because `j` is not evaluated.

Figure 9b The expected behavior of this program is to terminate normally. Because the value of `a` after exiting the loop at line 5 is 1, the program does not abort at line 21.

However, the program compiled with GCC trunk aborts. The ifcombine optimization mistakenly moves the uninitialized variable `k` at line 9 before the guard at line 8, introducing undefined behavior (`k` is now evaluated).

```c
int a;
int b;
short c;

int main () {
  int j;
  int d = 1;

  for (; c >= 0; c++) {
    a = d;
    d = 0;

    if (b) {
      printf ("%d", 0);
      if (j) {
        printf ("%d", 0);
      }
    }
  }

  printf ("%d\n", d);
  return 0;
}
```

(a) GCC development trunk (6.0.0 rev 228389) miscompiles the variant at `-O3`. The compiled binary prints 1 instead of 0. **`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=67828`**

```c
int a = 2, b = 1, c = 1;

int fn1 () {
  int d;
  for (; a; a--) {
    for (d = 0; d < 4; d++) {
      int k;
      if (c < 1)
        if (k) c = 0;
      if (b) continue;
      return 0;
    }
    b = !1;
  }
  return 0;
}

int main () {
  fn1 ();
  if (a != 1)
    __builtin_abort ();
  return 0;
}
```

(b) GCC development trunk (6.0.0 rev 229251) miscompiles the variant at `-O3`. The compiled binary aborts instead of terminating normally. **`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68083`.**

```c
1   int a[1], b, d, e, f, h, i, j;
2   volatile int c = 1;
3   char g;
4   void fn1 (int p1) { b = a[p1]; }
5   void fn2 () {
6     for (h = 15; h < 22; h += 5)
7       if (c) {
8         d--; f--; g--;
9         if (d) {
10          j = f < 0 || (f >= 0) > f;
11          i = e ^= 1;
12        }
13        e = 0;
14      }
15      else g = 0;
16  }
17  int main () {
18    a[0] = 1;
19    fn2 ();
20    fn1 (g & 1);
21    if (b != 1) __builtin_abort();
22    return 0;
23  }
```

(c) LLVM development trunk (3.8.0 rev 248820) miscompiles the variant at `-O2` and `-O3`. The compiled binary aborts instead of terminating normally. **`https://llvm.org/bugs/show_bug.cgi?id=24991`**

```c
int printf (const char *, ...);
int a[1], b[1][1], c, d;

void fn1 (int p1) {
  for (; d; d++)
    goto lbl;
  if (0)
lbl:
    c--;
  else
    b[p1][0] = 0;

  printf ("%d\n", a[p1]);
  goto lbl;
}
```

(d) LLVM 3.6, 3.7, and development trunk (3.8.0 rev 250927) crash while compiling the variant at `-Os` and above. **`https://llvm.org/bugs/show_bug.cgi?id=25291`**

```c
int a, b, c;

void fn1 () {
  short d = 0;
  for (; d < 2; d++) {
    for (b = 0; b < 1; b++)
      for (; c; c++)
        a = d;
    for (; b < 1; b++)
      for (c = 0; c;)
        ;
    if (c)
      break;
  }
}
```

(e) LLVM development trunk (3.8.0 rev 253143) crashes while compiling the variant at `-O2`. **`https://llvm.org/bugs/show_bug.cgi?id=25538`.**

```c
struct S {
  int f0;
  int f1;
};
int b;
int main () {
  struct S a[2] = { 0 };
  struct S d = { 0, 1 };
  for (b = 0; b < 2; b++) {
    a[b] = d;
    d = a[0];
  }
  if (d.f1 != 1)
    __builtin_abort ();
}
```

(f) GCC 4.9.2, 5.1, and development trunk (6.0.0 rev 223630) miscompiles the variant at `-O3`. The compiled binary aborts instead of terminating normally. **`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66272`.**

Figure 9: Some sample variants that trigger a variety of GCC and LLVM bugs.

Figure 9c The expected behavior of this program is to terminate normally. In the function `fn2`, the for loop executes twice, each time decreasing the value of `g` by 1. While calling `fn1` at line 20, the argument is 0 because `g` is -2. This function assigns `b` to `a[0]`, which is 1. Therefore, the `if` statement at line 21 is not executed, and the program does not abort. However, the program compiled by Clang trunk aborts because of a bug in calculating liveness.

Figure 9d Clang 3.6, 3.7, and trunk crashes while compiling this program at `-Os` and above because of an issue in global value numbering [32].

Figure 9e Clang trunk crashes while compiling this program at `-O2`. The bug happened because the developer did not handle a special case while avoiding recomputing the expensive loop-closed SSA form [15].

Figure 9f The expected behavior of this program is to terminate normally. After the first iteration of the loop at line 9, `a[0]` has the same value as `d`. After the second iteration, `a[1]` also has the same value as `d`. The program does not abort at line 14 because `d.f1` is 1.

However, the program compiled with GCC 4.9.2, 5.1, and trunk aborts. The predictive commoning pass [7] mistakenly hoisted the store `d.f1 = a[0].f1;` at line 11 out of the loop because it incorrectly assumes that the store is independent of the previous store to a[b].

6. Related Work

Our work is closely related to research on compiler testing and verification. In this section, we survey the three most related lines of work: compiler testing, compiler verification and translation validation.

6.1 Compiler Testing

Due to its practicability, compiler testing is still the dominant technique to assure compiler quality. For example, mature production compilers (*e.g.* GCC and LLVM) have their own regression test suites. Commercial test suites are also available such as PlumHall [23] and SuperTest [1] for language conformance and correctness checking. Most of these test suites are written manually.

An alternative to manually written test suites is random testing, which can further stress test compilers by providing extra testing coverage. Zhao *et al.* proposed a tool JTT to test the EC++ embedded compiler [36]. Nagai *et al.* [20, 21] proposed a technique to generate random arithmetic expressions to find bugs in the arithmetic optimizers of GCC and LLVM. CCG is another random C program generator that targets compiler crashing bugs [2]. Sun *et al.* proposed an automated approach to finding bugs in compiler warning diagnostics [27]. Chen *et al.* proposed a guided approach to detecting discrepancies between different implementations of Java Virtual Machine [3]. Sun *et al.* proposed a simple but effective program fuzzer to find crashing bugs in GCC and LLVM based on an observation that most of bug revealing test programs are small [28].

Csmith Csmtih [35] is one of the most successful random program generator for testing C compilers. It targets crashing and miscompilation bugs. Over the last several years, Csmith has made significant contribution to improve the quality of GCC and LLVM. Csmith is well-known for its careful control to avoid generating test programs with undefined behaviors and the large set of C language features it supports. In addition to compiler testing, Csmith has also been applied to test static analyzers such as Frama-C [5], and CPU emulators [18].

Csmith [35] and all these efforts [2, 20, 21, 36] generate test programs from scratch. We propose a technique to derive EMI variants from existing tests. [2, 20, 21, 36] only found several bugs. Csmith found hundreds of bugs before, but the current production compilers (*e.g.*, GCC and LLVM) are already resilient to it. We also used Csmith to generate seed programs, but rarely found bugs triggered directly by the seeds.

EMI Recently, Le *et al.* introduce equivalence modulo inputs (EMI), a technique that allows creation of many variants from a single program that are semantically equivalent under some inputs to stress test compilers [9]. They have developed a series of tools to instantiate EMI. Orion [9] deletes unexecuted statements randomly. Proteus [10] stress tests the link-time optimizers of GCC and LLVM by splitting a translation unit into multiple ones and randomizing optimization levels. Athena [11] randomly inserts code into and removes statements from dead code regions, and uses the Markov Chain Monte Carlo (MCMC) method to guide EMI variant generation.

Hermes complements all the three tools above, as our mutation strategies operate on live code regions, which overcome the limitations of mutating dead code regions as mentioned in Section 1. Moreover, our approach is also orthogonal to Athena because it can be easily integrated into the MCMC process of Athena to propose EMI variants.

CLsmith CLsmith is a testing system built on top of Csmith to validate OpenCL compilers [4] by leveraging differential testing [9, 35] (*i.e.*, using a reference compiler to serve as the test oracle) and EMI. As stated in [4], the dynamically-dead code regions are scarce in practical kernels, and therefore they injected dead-by-construction code into kernels. Specifically, they construct an always false conditional block by comparing the values of two global variables that, by construction, have values that are unknown at compile time but which are set to fixed and unchanging values at runtime. Later in [6], Donaldson and Lascu discussed a broader way to generate EMI variants by mutating existing expressions with identify functions. For example, an identify function can be defined as $id(\mathtt{e}) = \mathtt{e} + \mathtt{e}_0$, where $\mathtt{e}$ is an existing expression in the seed program, $\mathtt{e}_0$ is a synthesized expression whose value is always zero *w.r.t.* the same input, and the result expression $\mathtt{e} + \mathtt{e}_0$ is equivalent to $\mathtt{e}$ but in a different syntactical form.

Hermes differs from CLsmith as our mutation strategies interact with the local context more as CLsmith's false predicate relies on the comparison between two global variables, whereas ours are randomly synthesized over all in-scope variables. Moreover, we also generate live code blocks (TCB) which is not supported by CLsmith. All our mutation strategies are ready to be incorporated into CLsmith, which we believe can improve the diversity of its test programs.

6.2 Verified Compilers

A verified compiler ensures that the compilation from source code is semantics preserving, *i.e.*, the compiled code is semantically equivalent to the source code. This goal is achieved by accompanying a correctness proof with the compiler that guarantees semantic preservation.

The most notable verified compiler is CompCert [13, 14], a certified optimizing compiler for a subset of C language. The compiler with its proof has been developed using Coq proof assistant. Zhao *et al.* proposed a new technique to verify SSA-based optimizations in LLVM with the Coq proof assistant [37]. Malecha *et al.* applied the idea of verified compilers to the database domain and built a verified relational database management system [17]. Lopes *et al.* proposed a domain-specific language to automatically prove and generate peephole optimizers [16].

The benefit of verified compilers is clearly their guarantee of compilation correctness. For example, years of testing with Csmith, Orion and Athena have not revealed a single bug in the optimizer component of CompCert. This correctness guarantee is crucial to safety-critical domains. However, for general application domains, verified compilers have not been widely accepted due to their limited types of optimizations compared to the production compilers, *e.g.*, GCC and LLVM.

6.3 Translation Validation

Translation validation aims to verify that the compiled code is equivalent to its source code and find any compilation errors on the fly. The motivation originates from the fact that it is usually easier to prove the correctness of a specific compilation than to prove the correctness of compiling every input program.

Hanan Samet first introduced the concept of translation validation in [26]. Pnueli *et al.*'s work [24] proposed to use translation validation to validate the non-optimizing compilation from SIGNAL to C. Later, Necula [22] extended the concept to directly validate compiler optimizations and successfully validated four optimizers in GCC 2.7. Tate *et al.* introduced a translation validation framework for JVM based on equality saturation [29]. Later, Tristan *et al.* validated intraprocedural optimizations in LLVM [31] by adapting the work on equality saturation [29].

7. Conclusion

We have presented a class of novel EMI mutation strategies for compiler testing. Rather than mutating dead code regions like the state-of-the-art tools Orion and Athena, our technique directly manipulates live code regions, which significantly increases the space of EMI variants and testing efficiency. Within only 13 months, our realization Hermes has found 168 confirmed bugs in GCC and LLVM, of which 132 have already been fixed.

We keep actively testing GCC and LLVM with Hermes, and have still been able to find around 10 bugs per month even after many months of testing. For future work, we plan to design more live code EMI mutation strategies by leveraging the insights of this paper — investigating observed program states by running programs with a set of inputs in order to fabricate EMI mutation code.

8. Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments. This research was supported in part by the United States National Science Foundation (NSF) Grants 1117603, 1319187, 1349528 and 1528133, and 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.

References

[1] ACE. SuperTest compiler test and validation suite. http://www.ace.nl/compiler/supertest.html, accessed: 2016-03-20.

[2] A. Balestrat. CCG: A random C code generator.

[3] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of jvm implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 85–99, New York, NY, USA, 2016. ACM.

[4] N. Chong, A. Donaldson, A. Lascu, and C. Lidbury. Many-core compiler fuzzing. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[5] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer Berlin Heidelberg, 2012.

[6] A. F. Donaldson and A. Lascu. Metamorphic testing for (graphics) compilers. In *1st International Workshop on Metamorphic Testing, in concunction with the 38th International Conference on Software Engineering*, 5 2016.

[7] GCC Wiki. Predictive Commoning. https://gcc.gnu.org/wiki/PredictiveCommoning, accessed: 2016-03-20.

[8] T. Hoare. The verifying compiler: A grand challenge for computing research. In *Modular Programming Languages*, pages 25–35. Springer, 2003.

[9] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[10] V. Le, C. Sun, and Z. Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2014.

[11] V. Le, C. Sun, and Z. Su. 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, OOPSLA 2015, pages 386–399, New York, NY, USA, 2015. ACM.

[12] X. Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP'06)*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2006.

[13] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, 2006.

[14] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[15] LLVM's Analysis and Transform Passes. Loop-Closed SSA Form Pass. `http://llvm.org/docs/Passes.html#lcssa-loop-closed-ssa-form-pass`, accessed: 2016-03-20.

[16] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 22–32, New York, NY, USA, 2015. ACM.

[17] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–248, 2010.

[18] L. Martignoni, R. Paleari, A. Reina, G. F. Roglia, and D. Bruschi. A methodology for testing CPU emulators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):29, 2013.

[19] S. McPeak, D. S. Wilkerson, and S. Goldsmith. Berkeley Delta. `http://delta.tigris.org/`, accessed: 2016-03-20.

[20] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.

[21] E. Nagai, A. Hashimoto, and N. Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.

[22] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2000.

[23] Plum Hall, Inc. The Plum Hall Validation Suite for C. `http://www.plumhall.com/stec.html`, accessed: 2016-03-20.

[24] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166, 1998.

[25] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.

[26] H. Samet. *Automatically proving the correctness of translations involving optimized code*. PhD Thesis, Stanford University, May 1975.

[27] C. Sun, V. Le, and Z. Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 203–213, 2016.

[28] C. Sun, V. Le, Q. Zhang, and Z. Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 294–305, 2016.

[29] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 264–276, 2009.

[30] The Clang Team. Clang 3.4 documentation: LibTooling. `http://clang.llvm.org/docs/LibTooling.html`, accessed: 2016-03-20.

[31] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 295–305, 2011.

[32] Wikipedia. Global Value Numbering. `https://en.wikipedia.org/wiki/Global_value_numbering`, accessed: 2016-03-20.

[33] Wikipedia. Loop unswitching. `https://en.wikipedia.org/wiki/Loop_unswitching`, accessed: 2016-03-20.

[34] Wikipedia. Sequence point. `https://en.wikipedia.org/wiki/Sequence_point`, accessed: 2016-03-20.

[35] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[36] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test (AST)*, pages 36–43, 2009.

[37] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 175–186, 2013.