Finding Deep Compiler Bugs via Guided Stochastic Program Mutation

Vu Le Chengnian Sun Zhendong Su

Department of Computer Science, University of California, Davis, USA

{vmle, cnsun, su}@ucdavis.edu

Abstract

Compiler testing is important and challenging. Equivalence Modulo Inputs (EMI) is a recent promising approach for compiler validation. It is based on mutating the unexecuted statements of an existing program under some inputs to produce new equivalent test programs *w.r.t.* these inputs. Orion is a simple realization of EMI by only randomly deleting unexecuted statements. Despite its success in finding many bugs in production compilers, Orion's effectiveness is still limited by its simple, blind mutation strategy.

Orion目前存在的问题：随机、盲目

To more effectively realize EMI, this paper introduces a guided, advanced mutation strategy based on Bayesian optimization. Our goal is to generate diverse programs to more thoroughly exercise compilers. We achieve this with two techniques: (1) the support of both code deletions and insertions in the unexecuted regions, leading to a much larger test program space; and (2) the use of an objective function that promotes control-flow-diverse programs for guiding Markov Chain Monte Carlo (MCMC) optimization to explore the search space.

动机：为了更加高效地实现EMI，研究出了基于贝叶斯优化的方法

Our technique helps discover *deep* bugs that require elaborate mutations. Our realization, Athena, targets C compilers. In 19 months, Athena has found 72 new bugs — many of which are deep and important bugs — in GCC and LLVM. Developers have confirmed all 72 bugs and fixed 68 of them.

研究出的Athena

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

Keywords Compiler testing, equivalent program variants, automated testing, Markov Chain Monte Carlo

1. Introduction

Compiler bugs are serious because they potentially affect all programs. They may cause compilers to *silently miscompile* a program, leading to its incorrect executions and even security vulnerabilities in the miscompiled program. Because bugs in production compilers are relatively rare and only trigger under specific circumstances, they may go unnoticed during software development and surface only after deployment. Thus, compiler bugs can have catastrophic consequences, especially for safety-critical software; it is critical to make compilers reliable.

One approach is to develop verified compilers, which guarantee that the compiled executable behaves exactly as defined by its source program's semantics. A notable example in this direction is CompCert [11, 12], a verified compiler for Clight, a subset of the C language. CompCert's semantic preservation is mechanically verified using the Coq proof assistant. However, traditional compilers, such as GCC and LLVM, are still dominant as CompCert is designed for use mostly in the safety-critical domains, where people are more receptive to trading performance and language expressivity for stronger correctness guarantees.

介绍了几种检测编译器bug的方向

Testing remains the major method to validate mainstream compilers. In particular, random program generators (fuzzers) have been developed and demonstrated effective in practice. For instance, jsfunfuzz [8], a popular JavaScript fuzzer, has revealed thousands of bugs in Mozilla Firefox's JavaScript engine. Another example is Csmith [30], which has found hundreds of bugs in production C compilers such as GCC and LLVM. Because the semantics of a fuzzer-generated program is difficult to control, this approach requires the program to be compiled with multiple compiler implementations/configurations to find possible discrepancies (unless the goal is to simply crash a compiler).

随机程序测试一般都需要差分（多个不同的编译器）

To complement existing compiler testing methods, we have introduced Equivalence Modulo Inputs (EMI) [9], a general approach for constructing *valid* test programs from *existing* code. Given an existing program and some of its input, we profile the execution of the program under the given input. We then generate additional test programs (henceforth referred to as *variants*) by randomly pruning unexecuted code.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814319

EMI does not require different compiler implementations because the semantics of the variants is known — they must behave exactly the same as the original program under the same profiling input. Orion, our realization of EMI, has found 147 new bugs, mostly miscompilations, in GCC and LLVM.

这边主要在讲最早一版本的EMI策略的限制

Orion's Limitations Although Orion is effective, it has several technical limitations which mainly concern its simple strategy for generating variants — it only *randomly prunes* unexecuted statements.

1、生成的程序变体数量有限

First, Orion can only generate a limited number of variants from an existing program because there are only a bounded number of possible ways to remove unexecuted statements from a program. This becomes more problematic when a seed program does not contain much dead code. Indeed, it is possible to exhaust all possible variants for almost all the few thousand test cases in GCC's torture test suite.

2、生成程序本身的多样性受限

Second, the control- and data-flow diversity of the Orion-generated variants can be limited. This is again particularly true when the seed program does not contain many unexecuted statements. As EMI's effectiveness depends on having variants with different control- and data-flow to help exercise a compiler's various optimization strategies, this limitation of Orion hinders its capability to thoroughly test the compiler.

3、依赖于种子代码库，随机策略很难触及deep bugs

Third, Orion's generation process is purely random. While this simple strategy has been effective, its detected bugs may saturate because Orion's strategy may help reveal only *shallow* bugs that are within close proximity to the seed program. It is unlikely to find *deep* bugs that only trigger after a sequence of sophisticated mutations on the seed program.

Guided EMI Mutation This paper proposes effective techniques to address these limitations. First, besides deletion, we support *code insertion* into unexecuted program regions. Because we can potentially insert an unlimited number of statements into these regions, we can generate an enormous number of variants. More importantly, the generated variants have substantial different control- and data-flow, therefore helping exercise the compiler much more thoroughly. Our experimental results show that the increased variation and complexity are crucial in revealing more compiler bugs.

Second, we introduce a novel method to guide the generation process to uncover deep bugs. We formulate our bug finding process as an optimization problem whose goal is to *maximize* the difference between a variant and the seed program. By generating substantially diverse variants, we aim at testing more optimization strategies that may not be exercised otherwise. We realize this process using Markov Chain Monte Carlo (MCMC) techniques, which help effectively sample the program space to allow diverse programs. Our evaluation results show that this approach is very effective in finding deep bugs that require long sequences of sophisticated mutations on the seed program. Our results also demonstrate that most of these bugs could not be discovered by Orion, which only uses a much simpler, blind mutation strategy.

Contributions We make the following main contributions:

- We implement a new EMI mutation strategy that allows inserting code into unexecuted regions. This helps generate more diverse variants that have substantially different control- and data-flow.

主要的创新在于实现EMI的方式在于：插入代码

- We propose a novel guided bug-finding process that uses MCMC sampling to find more diverse test variants to trigger deep bugs that otherwise could not be triggered using existing techniques.

另一个创新在于使用MCMC来找到更多的变体

- We realize our technique as the `Athena` tool for validating C compilers. In 19 months, `Athena` has found 72 new bugs in GCC and LLVM. Developers have confirmed all of our reported bugs and fixed 68 of them. Furthermore, 17 of our 40 GCC bugs were marked as P1, the most severe, release-blocking type of bugs. A number of our reported bugs are linked to real-world programs. Our results also demonstrate the effectiveness of our MCMC guided algorithm in finding deep bugs.
- We provide further evidence to demonstrate the effectiveness of the general EMI technique and insight into developing effective EMI mutation strategies.

We structure the remainder of this paper as follows. Section 2 illustrates our approach via two of our reported bugs. Section 3 introduces necessary background. We then present our MCMC-guided compiler testing methodology in Section 4, and discuss the implementation of `Athena` in Section 5. Section 6 discusses our experimental results. We survey related work in Section 7 and conclude in Section 8.

2. Illustrative Examples

This section uses two concrete bug examples to motivate and illustrate `Athena`: one LLVM bug and one GCC bug. Both bugs start from seed programs generated by Csmith [30], and trigger by a sequence of mutations, *i.e.*, inserting and deleting statements derived from the seeds.

2.1 LLVM Crashing Bug 18615

Figure 1a shows the *reduced* EMI variant that triggers the bug. Initially, `clang` compiles the seed program successfully at all optimization levels. However, after `Athena` replaces the original statement `f[0].f0 = b;` in the seed program with another statement `f[0] = f[b];` (which is derived from the seed program), it causes `clang` to crash at optimization `-O1` and above. The bug happens because an assertion is violated. LLVM assumes that array indices used in memory copy operations are non-negative. At line 9 of Figure 1a, the variable `b` is -1, making the array access `f[b]` illegal. However, this should not matter because the code is never executed. The compiler should not crash.

Athena extracts candidate statements for insertion from existing code and saves them into a database. Each database entry is a pair of statement and its required context, where the context specifies the necessary conditions to apply the

从已有的程序中抽取statements放进数据库中

```
int a;
struct S0 {
  int f0; int f1; int f2;
};
void fn1 () {
  int b = -1;
  struct S0 f[1];
  if (a) // the true branch is unreachable
    f[0] = f[b]; // was "f[0].f0 = b;"
}
int main () {
    fn1 ();
    return 0;
}
```

(a) The simplified EMI variant.

数据库中数据样式

```
...
=====
// Required context
g: struct (int x int x int) [1]
c: int
-----
// Statement
g[0] = g[c];
=====
...
```

(b) The database entry used to insert into the variant. Athena renamed g, c to f, b to match the context at the insertion point.

Figure 1: LLVM 3.4 trunk crashes while compiling the variant at -O1 and above (https://llvm.org/bugs/show_bug.cgi?id=18615).

这边我比较好奇的是，插入代码段之后，如何保证不影响输出值的结果（想起来了，插入的块原本就是dead）

statement. Figure 1b shows the database entry that was inserted into the seed program to reveal the bug. To use the statement of this entry, the insertion point must have an array of structs g and an integer c in scope.

While performing insertion, Athena only selects from the database statements whose required contexts can be satisfied by the local context (at the insertion point) to avoid generating invalid programs. Athena also renames constructs in a selected database statement to match the local context when necessary. In this example, we can replace the original unexecuted statement with the statement in Figure 1b because their contexts are compatible under the renaming g $\rightarrow$ f and c $\rightarrow$ b.

Note that the program in Figure 1 is already reduced. The original program and its variant are quite large. Also, the bug was triggered not under one single step but under a sequence of mutations. Athena's MCMC algorithm plays a key role here. It guides the mutation process toward generating programs that are more likely to expose bugs. Orion cannot reveal the bug because it cannot insert new statements to trigger the assertion violation.

将本文的方法和Orion对比，主要突出本文方法中插入新的语句的作用：增大探索bug的机会

```
 1 int a, c, d, e = 1, f;
 2 int fn1 () {
 3   int h;
 4   ...
 5   for (; d < 1; d = e) {
 6     h = f == 0 ? 0 : 1 % f;
 7     if (f < 1)
 8       c = 0;
 9     else // the else branch is unreachable
10       if(h) break; // was "c = 1;"
11   }
12   ...
13 }
14 int main () {
15   fn1 ();
16   return 0;
17 }
```

(a) The simplified EMI variant.

```
...
=====
// Required context
requires_loop
i: int
-----
// Statement
if (i)
  break;
=====
...
```

(b) The database entry used to insert into the variant. Athena renamed i to h. requires_loop requires the statement to be inserted inside a loop.

Figure 2: GCC trunk 4.10 and also 4.8 and 4.9 miscompile the variant at -O2 and -O3 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61383).

也是没有输出，但是比较的是成功运行和提前终止运行

2.2 GCC Miscompilation Bug 61383

Figure 2a shows a simplified version of a GCC miscompilation bug. Because the loop on line 5 executes only once, the expected semantics of this program is to terminate normally. However, the compiled binary of the EMI variant aborts during its execution.

While compiling the variant, GCC identified the expression 1 % f as a loop invariant for the loop on line 5. As an optimization, GCC hoisted the expression out of the loop to avoid redundant computation. However, this optimization is problematic because now the expression is evaluated. Since f is 0, the expression 1 % f traps and aborts the program. This expression should not be evaluated because the conditional expression always takes the "then" branch (f == 0 is true).

By adding the extra statement `if (h) break;` from the database to line 10, `Athena` changes the control-flow of the variant. The extra complexity causes the optimization pass `ifcombine` to miss the check whether the expression `1 % f` can trap. Hence, it concludes that the statement does not have any side effect. The expression is incorrectly hoisted and the bug is triggered.

Because this bug-triggering statement contains a `break` statement, we can only insert it inside a loop. While traversing the source program, `Athena` keeps track of all variables in scope and an additional flag indicating whether the current location is inside the body of a loop. On line 10, `Athena` renames `i` to one of the available variables, `h`, and replaces the unexecuted statement with `if (h) break;`.

3. Background

This section discusses relevant background on Equivalence Modulo Inputs and Markov Chain Monte Carlo.

3.1 Equivalence Modulo Inputs

Le *et al.* [9] have introduced the Equivalence Modulo Inputs (EMI) methodology for testing compilers. Two programs P and Q are equivalent modulo inputs *w.r.t.* an input set I (denoted as $[\![P]\!] =_I [\![Q]\!]$) if they behave exactly the same on I (but may behave differently on other input). Formally,

$$[\![P]\!] =_I [\![Q]\!] \iff \forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i)$$

This relaxed notion of program equivalence is attractive because it enables the testing of a compiler in isolation using test variants generated from existing code.

Given an existing program P and its input I, we profile the execution of P under I. We then generate new test variants by mutating the *unexecuted* statements of P (such as randomly deleting some statements). This is safe because all executions under I will never reach the unexecuted regions. The newly generated variants are EMI variants of P *w.r.t.* I.

EMI can test a compiler in isolation because the expected semantics of the generated variants is known (*i.e.*, producing the same output as P on I). If we do not know the semantics of the program, such as one generated by a fuzzer, we have to rely on multiple compilers to find potential discrepancies among the compiled executables.

Another appealing property of EMI is that the generated variants are always valid provided that the seed program itself is valid. In contrast, randomly removing statements from a program is likely to produce invalid programs, *i.e.*, those with undefined behaviors.

3.2 Markov Chain Monte Carlo

Monte Carlo is a general method to draw samples X_i from a target density distribution $p(X)$ defined on a high-dimensional space $\mathcal{X}$ (such as the space of all possible configurations of a system, or the set of all possible solutions of a problem). From these samples, one can estimate the target density $p(X)$.

A stochastic process $\{X_0, X_1, X_2, \dots\}$ is a Markov chain if the next state X_{t+1} sampled from a distribution $q(X_{t+1} \mid X_t)$ only depends on the current state of the chain X_t. In other words, X_{t+1} does not depend on the history of the chain $\{X_0, X_1, \dots, X_{t-1}\}$.

Markov Chain Monte Carlo (MCMC) draws samples X_i in the space $\mathcal{X}$ using a carefully constructed Markov chain, which allows more samples to be drawn from important regions (*i.e.*, regions having higher densities). This nice property is obtained when the chain is *ergodic*, which holds if it is possible to transition from any state to any other state in the space $\mathcal{X}$. The samples X_i mimic samples drawn from the target distribution $p(X)$. Note that while we cannot sample directly on $p(X)$ (we are simulating this unknown distribution, which is why we use MCMC in the first place), we should be able to evaluate $p(X)$ up to a normalizing constant.

The Metropolis-Hasting algorithm is the most popular MCMC method. This algorithm samples the candidate state X^* from the current state X according to the proposal distribution $q(X^* \mid X)$. The chain accepts the candidate and moves to X^* with the acceptance probability as follows:

$$\mathcal{A}(X \to X^*) = \min\left(1, \frac{p(X^*)q(X \mid X^*)}{p(X)q(X^* \mid X)}\right) \tag{1}$$

Otherwise, the chain remains at X, and a new candidate state is proposed. The process continues until a specified computational budget is reached.

The Metropolis algorithm is a simple instance of the Metropolis-Hasting algorithm, which assumes that the proposal distribution is symmetric, *i.e.* $q(X^* \mid X) = q(X \mid X^*)$. Our acceptance probability simplifies to the following:

$$\mathcal{A}(X \to X^*) = \min\left(1, \frac{p(X^*)}{p(X)}\right) \tag{2}$$

While MCMC techniques can be used to solve many problems including integration, simulation and optimization [2, 6], our focus in this paper is optimization.

4. MCMC Bug Finding

In our setting, the search space $\mathcal{X}$ is the space of all EMI variants of a seed program P. Because our computational budget is limited, we want to sample more "interesting" variants in this space $\mathcal{X}$. For this reason, we need to design an effective objective function that determines if a variant is interesting and worth exploring.

4.1 Objective Function

Our key insight for a suitable objective function is to *favor variants having different control- and data-flow* as the seed program. When compiling a program, compilers use various static analyses to determine — based on the program's control- and data-flow information — which optimizations are applicable. By generating variants with different control-

and data-flow, we are likely to exercise the compilers more thoroughly by forcing them to use various optimization strategies on the variants. In particular, we use *program distance* to measure the difference between two programs.

Definition 4.1. (*Program Distance*) *The distance Δ between an EMI variant Q and its seed program P is a function of the distance between their control-flow graph (CFG) nodes (*i.e.*, basic blocks), the distance between their CFG edges, and their size difference. Specifically,*

程序距离

$$\Delta(Q;P) = \alpha \cdot d(V_Q, V_P) + \beta \cdot d(E_Q, E_P) - \gamma \cdot |Q - P|$$

where

- $d(A,B) = 1 - \frac{A \cap B}{A \cup B}$ *is the Jaccard distance [29];*
- V_Q, V_P *are Q and P's CFG node sets respectively;*
- E_Q, E_P *are Q and P's CFG edge sets respectively; and*
- $|Q - P|$ *is the program size difference of Q and P.*

Two nodes are different if their corresponding statements are different. If we modify a node in the variant, the node will be different from the original one. Two edges are different if their corresponding nodes are different.

优点是可以提升蜕变程序的复杂性，触发更多的优化

Intuitively, our notion of program distance measures the *changes* in the variant. It is capable of capturing simple changes that do not alter the control flow, such as deleting and inserting straight-line statements, via the node distance. It also captures complex changes that modify the control and data flow considerably, such as deleting and inserting complicated statements, via both the node and edge distance.

另一个优点是不会使得生成的测试程序过大或者过小：过大编译时间过长；过小很难触发bug

Our program distance metric disfavors changes in program size. This helps avoid generating too small or too large variants. Small variants are less likely to reveal bugs. Large variants may take significant amount of time to compile, thus may prevent us from sampling many variants.

4.2 MCMC Sampling

When applied to optimization, an MCMC sampler draws samples more often from regions that have *higher* objective values. We leverage this crucial property to sample more often the program space that produces more different EMI variants (*i.e.*, ones with larger program distances Δ).

To calculate the transition acceptance probability, we need to evaluate the density distribution $p(.)$ at any step in the chain. According to [6, 23], we can transform any arbitrary objective function into a density distribution as follows

$$p(Q;P) = \frac{1}{Z} \exp\left(\sigma \cdot \Delta(Q;P)\right) \tag{3}$$

where σ is a constant and Z a normalizing partition function.

Algorithm 1: MCMC algorithm for testing compilers

```
1  procedure BugFinding(Compiler $C$, Seed test $P$, Input $I$):
2      $O := C$.Compile($P$).Execute($I$)    /* ref. output */
3      $Q := P$                               /* initialization */
4      for 1 .. MAX_ITER do
5          $Q^* :=$ Mutate($Q, I$)            /* propose candidate */
6          $O^* := C$.Compile($Q^*$).Execute($I$)
7          if $O^* \neq O$ then
8              ReportBug($C, Q^*$)
9          if Rand(0,1) $< \mathcal{A}(Q \to Q^*; P)$ then
10             $Q := Q^*$                     /* move to new state */
```

Deriving from (1), the probability to accept the proposal $Q \to Q^*$ is given below:

$$\begin{aligned} \mathcal{A}(Q \to Q^*; P) &= \min\left(1, \frac{p(Q^*;P) \cdot q(Q \mid Q^*)}{p(Q;P) \cdot q(Q^* \mid Q)}\right) \\ &= \min\left(1, \exp\left(\sigma \cdot \left(\Delta(Q^*;P) - \Delta(Q;P)\right)\right) \cdot \frac{q(Q \mid Q^*)}{q(Q^* \mid Q)}\right) \end{aligned} \tag{4}$$

where $q(\cdot)$ is the proposal distribution, $q(Q \mid Q^*)$ is the probability of transforming Q^* to Q, and $q(Q^* \mid Q)$ is the probability of transforming Q to Q^*.

We develop our bug finding procedure based on Metropolis-Hasting algorithm. Algorithm 1 illustrates this procedure. We start with the seed program (line 3). The loop on lines 4-10 simulates our chain. At each iteration, based on the current variant Q, we propose a candidate Q^*, which we use to validate the compiler. The chain moves to the new state Q^* with the acceptance probability described in (4). Otherwise, it stays at the current state Q.

4.3 Variant Proposal

We generate a new candidate Q^* by removing existing statements from and inserting new statements into the current variant Q. All mutations happen in the unexecuted regions under the profiling inputs I to maintain the EMI property.

Removing unexecuted statements is straightforward. We can safely remove any of these statements from the program without affecting its compilability. We only need to be careful not to remove any declaring statements whose declarations may be used later in the program.

However, inserting new statements into unexecuted regions is not as straightforward. In particular, we need to construct the set of all statements suitable for insertion. Also, while inserting new statements, we need to take extra care to make sure the new variants compile.

这边讲的是如何从已有的程序中找到可以用来做插入的程序片段

Extracting Statement Candidates We extract statement candidates from *existing code*. Given a corpus of existing programs, we construct the database of all available statements by traversing the ASTs of these programs and extract all statements at all levels. These statements have different complexities, ranging from a single-line statement to a whole function body.

这边也加了一些条件限制来确保插入后的变体可以被编译

For each statement, we determine the *context* required to insert the statement. When we perform insertion into a location, we only insert statements whose required contexts can be satisfied by the local context at the location. This is to guarantee that the new variant compiles after insertion.

Generating New Variant We use our statement database to facilitate variant mutation. At an unexecuted statement, we can perform the following actions according to the proposal distribution $q(\cdot)$:

Delete Similar to Orion, we keep track of two probabilities p^d_{parent} and p^d_{leaf}. We delete a parent statement, those that contain other statements, with probability p^d_{parent}. We delete a leaf statement with probability p^d_{leaf}. We distinguish these two kinds of statements because they have different effects on the new variant.

Insert We also have two probabilities p^i_{parent} and p^i_{leaf} for inserting at parent or leaf statements. We can insert a new statement either before or after the unexecuted statement (with the same probability). If the unexecuted statement does not directly belong to a compound statement, we promote it to a compound statement before inserting the new statement. This is to make these statements share the same parent.

We perform a breadth-first traversal on the AST of the current variant Q. During this traversal, we maintain a *context table* that contains necessary information (such as the variables in scope) to help select compatible statements from the database. At each unexecuted statement, we delete the statement or insert a new statement according to the probabilities defined above. It is possible to do both, in which case the unexecuted statement is replaced by the new statement.

Maintaining Ergodicity Our mutation must satisfy the ergodicity property described in Section 3.2, which states that we should be able to walk from one state to any other state in our search space. If this property is violated, we cannot perform the walk because the search space is disconnected.

这边主要是想解决从Q*恢复成Q需要解决的两个问题：一个是dead code位置；一个是dead code原始的语句保留策略

Let us first see if we can revert our proposal Q^* to Q. We can easily revert an inserted statement by deleting it. However, it is impossible to revert a deleted statement. As the statement is removed, our coverage tool cannot mark the location as unexecuted, which is the necessary condition for insertion. Moreover, if the deleted statement does not exist in our database, our insertion process will not be able to recover this statement.

引入了标记以便在dead block上留下记号

To address the first problem, we leave a syntactic marker in place of the deleted statement. While mutating the variant, we replace these markers with new statements as if they are unexecuted statements. In our implementation Athena, we use the statement "`while(0);`". Comments do not work because our tool only visits statements. Although these markers do not have any effects on the variant semantics, they may affect the compilers under test. Hence, we remove them from the variant before testing.

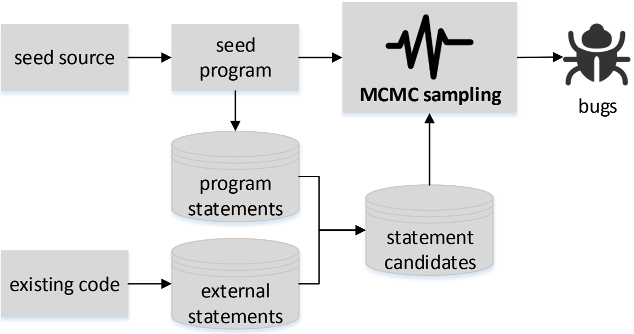

Figure 3: The high-level architecture of Athena. We extract statement candidates from both existing code and the seed program. We then perform MCMC sampling on the seed program to expose compiler bugs.

To solve the second problem, we allow code from the seed program. Before sampling, we extract statements from the seed and add them to the statement database. Because the deleted statement is either from the seed program or external code, we will be able to reconstruct it from the updated database.

Our process is now ergodic, and hence applicable for sampling. We can transform any EMI variant of a program to any other EMI variant of that program.

5. Athena

Athena is a concrete realization of our MCMC bug finding procedure targeting C compilers. We highlight its architecture in Figure 3. This section discusses various design choices we made in implementing Athena.

5.1 Extracting Statement Candidates

Athena uses the tool `stmt-extractor` to extract statement candidates from existing code corpus. We implement `stmt-extractor` using LLVM's Libtooling library [27]. For each program in the corpus, `stmt-extractor` visits all statements at all levels in the program's AST. It then determines the context required to apply such statements and inserts these $\langle$context, statement$\rangle$ pairs to the database.

A context contains the following information:

- The variables (including their types) used in the statement but defined outside. These variables must be in scope at the insertion point when we perform insertion (otherwise we would have inserted an invalid statement). We exclude variables that are defined and used locally.
- The functions (including their signatures) used in the statement.
- The user-defined types used in the statement such as structures or unions, and their dependent types.

- The (`goto`) labels *defined* in the statement. If a function has already defined these labels, we have to rename the labels in the inserted statement.
- The labels *used* in the statement. We have to rename these labels to match those defined in the function.
- A flag indicating whether the statement contains a `break` or `continue` statement. If this is the case, we can only insert this statement inside a loop.

To construct the context, we parse the statement and collect the information listed above. In particular, we find the required variables by finding all variables used in the statement, and subtracting them with those that are defined inside the statement. During this process, we also collect all used functions, user-defined types, labels, and break and continue statements. If the statement uses any user-defined type, we recurse into the type and construct its dependencies. Type dependencies are cached to avoid redundant computation.

In Figure 3, we only need to calculate the database of candidate statements *once* and the process happens *offline*. We also use `stmt-extractor` to extract statements from the seed program.

5.2 Proposing Variants

这个组件的主要作用就是将从数据库提取出来的stmt经过合适于目标程序的适配转化，变成可以插入到目标程序中的stmt

We implement a tool called `transformer` to transform variants, also using LLVM Libtooling. The tool takes as input a program (which is either the seed or one of its EMI variants in the chain), the program's coverage information (obtained using GNU gcov [7]), and the four deletion/insertion probabilities mentioned in Section 4.3.

Transformation The `transformer` tool performs a breadth-first traversal on the program. It keeps a context table that stores variables and labels in scope, the available functions and user-defined types, and a flag indicating whether the current statement is in a loop. It deletes unexecuted statement or inserting new ones according to the given probabilities. If a statement is deleted, we will stop traversing into it. We do not traverse into newly inserted statements.

While it is possible to keep the deletion/insertion probabilities unchanged during the mutation, it is better to randomize them in our experience. Hence, we shuffle these probabilities after each deletion or insertion.

Statement Renaming Because external code has different semantics and naming convention, there are usually not many statement candidates compatible with the local program context at insertion points. For this reason, we relax the context matching condition to accept matches under *renaming*. In particular, we allow the renaming of variables in the statement candidates to those in the local context that have compatible types. For example, if the context of a statement candidate requires an integer variable b, but the local context only has an integer variable a, we can rename all occurrences of b in the candidate to a. Similarly, we can rename labels, functions and user-defined types to those having compatible signatures.

5.3 Discussion

Although our algorithm satisfies the ergodicity property, it is not symmetric. That is, we cannot walk back to Q^* from Q in one step in general (but we may via several steps). This is because a deleted statement from Q may be constructed from several steps, and does not exist in our database. For example, we may insert a large statement s into Q_k, and since s is also unexecuted, some of its children are deleted in Q_{k+1} (let us call the updated statement s'). If we delete s' in Q_{k+2}, we cannot go back to Q_{k+1} in a single step because the database does not have s'.

One possible solution is to extract the statements from every variant, and update the database after each transition. We have decided not to do this because many statements are redundant. It is quite challenging to distinguish these statements since many of them have been renamed to match the local context. 一个局限性：不对称

As a result, our proposal distribution is not symmetric. To simplify our implementation, we assume that this distribution is symmetric (*i.e.*, it is equally likely to generate a new variant Q^* from the current variant Q and vice versa). The consequence of this assumption is that we may not sample the program space proportionally to the value of the objective function. For instance, we may not sample as often (or more often than required) the program space that has high objective values.

This is an example of the trade-offs between precision and efficiency. Making our process more precise may incur major performance degradation. On the other hand, having a less precise process helps sample many more variants in the same unit of time. This justifies the lack of sampling precision.

Based on this assumption, our algorithm turns into Metropolis algorithm, which has the following simpler acceptance probability:

$$\mathcal{A}(Q \to Q^*; P) = \min\big(1, \exp\big(\sigma \cdot (\Delta(Q^*; P) - \Delta(Q; P))\big)\big) \tag{5}$$

6. Evaluation

We focus our testing efforts on two open-source compilers GCC and LLVM because of their openness in tracking and fixing bugs. We summarize below our results from the end of January 2014 to the end of August 2015: 实验耗时1年半

- **Many detected bugs:** In 19 months, Athena has revealed 72 new bugs. Developers confirmed *all* of our bugs and fixed nearly all of them (68 out of 72).
- **Many serious bugs:** GCC developers marked 17 out of 40 GCC bugs as P1, the most severe kind of bugs that must be fixed before a new GCC release can be made. Three of our GCC bugs were linked to subsequent bugs exposed while compiling real-world code, namely `gcc`, `qtwebkit`, and `glibc`.
- **Many deep bugs:** Our experiments show that Athena is capable of detecting both shallow and deep bugs. The later

requires sophisticated mutation sequences that could not be done using Orion.

- **Many long-latent bugs:** Although our focus is to test the development trunks of GCC and LLVM, we have found 15 latent bugs in old versions of the two compilers. These bugs had resisted traditional validation approaches, which further illustrates Athena's effectiveness.

6.1 Testing Setup

We first describe our testing setup before presenting our detailed results.

Sources of Seed Programs Athena is capable of using existing open-source projects as seed programs. However, it is challenging to reduce bugs triggered by these projects because the projects typically involve multiple directories and multiple files [9]. Therefore, in our evaluation, we only use programs generated from the random program generator Csmith [30]. Existing reduction tools such as Berkeley's Delta [15] and CReduce [21] can reduce these programs effectively.

Sources of Statement Candidates Athena is also capable of using existing code as candidates for insertion. As part of our evaluation, we built a database of all statements extracted from the SPEC CINT2006 benchmarks [25]. Because the database contains a huge number of statements, it is very expensive to perform a linear scan on it. Note that we cannot look up by required context because a local context is different from these required contexts, and it may satisfy not one but multiple of them. Our solution is to repeatedly draw a random statement from the database until we find one that satisfies the local context. If we cannot find any satisfying statement after some constant k attempts, we conclude that no satisfying candidate exists.

Unfortunately, our experiments show that inserting real-world code into Csmith-generated seeds is ineffective. This is because Csmith can only generate limited forms of constructs, making the contexts at insertion points incapable of accepting the more diverse real-world code. One way to mitigate this problem is to merge the required constructs from external projects to the current variant. This is quite challenging because these constructs may depend on other constructs, or locate in a different location. We leave this for future work.

The seed program turns out to be a great source for statement candidates. A seed program can yield hundreds of statements that have diverse complexities: statements range from one line to hundred lines of code. Moreover, these statements are well connected to the variants, which helps increasing the ratio of satisfying statements. Our evaluation uses only statements from the seed programs.

Selecting Statement Candidates For each unexecuted location, there are potentially multiple satisfying statements from the database. A good strategy to select the "best" statement from this satisfying set may yield a better result overall.

	GCC	LLVM	TOTAL
Fixed	38	30	68
Not-Yet-Fixed	2	2	4
WorksForMe	0	3	3
Duplicate	3	4	7
Invalid	1	0	1
TOTAL	44	39	83

Table 1: Reported bugs.

We hypothesize that the best statement is one that uses the most information from the local context. For instance, it uses the most number of variables defined in the variant. Using this statement puts more constraints on the compiler because it increases the dependencies between existing code and external code.

Under this strategy, we have to scan the database to find the best statement candidate. Our experiments show that this strategy is two orders of magnitude slower than the random sampling strategy. Because speed is key in testing, we adopt the random sampling strategy and use it in our evaluation.

Testing Infrastructure Our testing focuses on the x86-linux platform due to its popularity and ease of access. We conduct our experiments on two machines (one 18 cores and one 6 cores) running Ubuntu 12.04 (x86_64). While calculating the program distance Δ, we value equally changes in CFG nodes and edges ($\alpha = \beta = 0.5$). We fine-tuned γ to avoid generating programs larger than 500KB, a threshold at which we observed a significant degradation in compilation time for both GCC and LLVM.

As in the work on Csmith and Orion, we have focused on the five standard options, "-O0", "-O1", "-Os", "-O2" and "-O3", because they are the most commonly used. Athena tests only the daily-built development trunks of GCC and LLVM. Once it finds a bug in a compiler, Athena also validates the bug against other major releases of that compiler.

6.2 Quantitative Results

We next present some general statistics on our reported bugs.

Bug Count Table 1 summarizes our bug results. In 19 months, we reported 83 bugs, which are roughly equally divided between GCC (44 bugs) and LLVM(39 bugs). Developers confirmed 72 valid bugs and fixed 68 of them.

Not-Yet-Fixed Bugs Among two GCC bugs that have not been fixed, one was just reported recently. The other one (bug 62016) is a performance bug, which affects GCC 4.8.X, 4.9.X, and 4.10 trunk (at that time). GCC took a few minutes to compile a small program due to problems in inlining. Developers discussed about backporting some code across versions to fix the problem. Unfortunately, this was quite challenging because of a cross-version design gap. A few months later, GCC moved to version 5.0 and some design changes fixed the problem. The bug therefore remains unfixed, although it still affects earlier versions. One of the two not-yet-fixed LLVM bugs is a complicated one. Developers have not found a solution to fix it yet. The other bug triggers only in debug mode, which perhaps is the reason why developers have not fixed it.

WorksForMe Bugs It may take a while before developers consider our reported bugs. During this time, the trunk has changed and it is possible that these changes suppress the bug. Developers mark bugs that no longer trigger "WorksForMe". We have three LLVM bugs of this kind. We do not have this kind of bugs in GCC because GCC developers responded to our bugs very quickly. Moreover, even when this happens, GCC's policy recommends going back to the affected revision and check if the root cause has been properly fixed. If not, the bug may be latent and is likely to trigger later. Indeed, one LLVM WorksForMe bug (bug 21741) re-triggers in a later revision. We reopened the bug, but LLVM developers have not yet responded.

Duplicate Bugs Before reporting a bug, we ensure that it has different symptoms from the previously reported and not yet fixed bugs. However, reporting duplicated bugs may be unavoidable because compilers are complex. Bugs having different symptoms may turn out to have the same root cause. During our evaluation, we reported 7 duplicated bugs (3 GCC and 4 LLVM).

As an example, GCC bugs 64990 and 645383 are duplicates of bug 61047. Bug 61047 only triggers at `-O1` on GCC 4.9 and 4.10 trunk. Bug 64990 triggers at all optimization levels on all versions of GCC from 4.6 to 5.0 trunk. Bug 65383 only triggers at `-O2` and `-O3` from GCC 4.7 to trunk, and the program looks very different from bug 64990. Although these bugs affected different versions and triggered at different optimization levels, they turned out to have the same root cause. Developers marked two bugs reported later as duplicates.

Invalid Bugs We reported one invalid bug (bug 63774) for GCC, in which a function returns the address of a local variable. We were unsure whether this behavior was implementation defined or undefined. It turned out that the behavior was undefined, and the bug was subsequently marked as invalid.

Bug Type We classify bugs into two main categories: (1) ones that manifest when we compile programs, and (2) ones that manifest when we execute the compiled binaries. A compile-time bug can be a *crashing bug* (*e.g.*, internal compiler errors), or *performance bug* (*e.g.*, the compiler hangs or takes a very long time to compile the program). A runtime bug happens when the compiled program behaves abnormally *w.r.t.* its expected behavior. For example, it may crash, hang, or produce wrong output. We call these bugs *miscompilation* bugs.

Table 2 classifies our 72 confirmed and valid bugs according to the above taxonomy. These bugs are quite diverse, illustrating the power of Athena in finding all kinds of bugs. A significant chunk of these bugs are miscompilation, the most serious kind among the three.

	GCC	LLVM	TOTAL
Miscompilation	11	17	28
Crash	26	13	39
Performance	3	2	5

Table 2: Bug classification.

Importance of Reported Bugs Developers took our bugs seriously. They have confirmed all of our bugs and fixed nearly all of them. GCC developers are generally more responsive in fixing bugs — they fixed most of our reported bugs within several days.

Three of our GCC bugs were linked to bugs triggered while compiling real-world projects. Bug 63835 is related to a bug that crashes GCC while compiling GCC itself. Bugs 61042 and 61684 crash GCC while compiling our variants. Later, people reported similar bugs while compiling `qtwebkit` and `glibc`. Subsequently, these bugs were marked as duplicates. Such bugs are rare because developers usually fix our bugs very quickly, leaving only a small time window for people to rediscover these bugs using real-world projects.

Another way to measure the importance of our bugs is via the "Importance" field set by developers in the bug reports. Developers marked 17 of our 40 GCC bugs as "P1", the highest bug priority (the default is "P3"). Developers must fix all P1 bugs before they can release a new version of GCC. LLVM developers marked all our bugs using the default value "P normal". We do not know whether these bugs have normal severity or LLVM developers did not classify the bugs.

6.3 Effectiveness of MCMC Bug Finding

Athena is effective because it is able to find many bugs after developers have fixed numerous bugs reported by Csmith and Orion. However, it is unclear how many of these bugs are deep bugs and could not be found by Orion. We conduct another experiment to compare Athena directly with Orion to answer this question.

We run Athena and Orion in parallel for a certain amount of time using the same seed programs that trigger our bugs. If Athena finds a bug, we reset the chain to the seed program and continue. This does not apply to Orion because it always operates on the seed program.

We limit this experiment to bugs that affect previous stable releases. It is because if the bug only affects the trunk, we need to build the compiler at that revision. Since the number of bugs is large, it is quite expensive to build a revision for each of them.

Table 3 shows the results of running Athena and Orion in parallel on 15 of such bugs for one week. In the table, we assume that all bugs rediscovered under a same seed file are the same. From our experience with both Orion and Athena, it is unlikely for the same seed Csmith program to reveal multiple bugs in the same compiler revision.

Bug ID (1)	Type (2)	Affected Compilers (3)	Optimizations (4)	Seed (5)	Variant (6)	Report (7)	DB (8)	# Bugs (9)	# Variants (10)
gcc-59903	Crash	4.8, 4.9 (trunk)	-O3	4,694	6,238	38	1,723	14	23,479
gcc-60116	Mis.	4.8, 4.9 (trunk)	-Os	11,596	11,843	25	3,092	367	20,082
gcc-60382	Crash	4.8, 4.9 (trunk)	-O3	6,151	21,903	19	1,989	19	21,267
gcc-61383	Mis.	4.8, 4.9, 4.10 (trunk)	-O2, -O3	3,298	3,567	22	1,272	106	32,981
gcc-61452	Hang	4.8 - 5.0 (trunk)	-O1, -Os	3,308	3,474	17	885	0	49,158
gcc-61917	Crash	4.9, 4.10, 5.0 (trunk)	-O3	11,820	11,226	7	3,066	2	32,562
gcc-64495	Crash	4.8 – 4.10, 5.0 (trunk)	-O3	2,767	1,951	20	517	4	45,896
gcc-64663	Crash	4.6 – 4.10, 5.0 (trunk)	-O1, -Os, -O2, -O3	11,118	12,160	9	2,875	0	26,626
llvm-20494*	Mis.	3.2 – 3.4, 3.5 (trunk)	-O2, -O3	8,080	11,009	23	1,683	2,660	24,588
llvm-20680	Mis.	3.5, 3.6 (trunk)	-O3	6,250	7,584	15	1,753	22	23,438
llvm-21512*	Crash	3.5, 3.6 (trunk)	-O1, -Os, -O2, -O3	8,455	5,087	11	3,081	988	21,882
llvm-22086	Crash	3.5, 3.6 (trunk)	-Os, -O2, -O3	5,220	8,495	27	1,711	0	29,279
llvm-22338	Crash	3.5, 3.6, 3.7 (trunk)	-O2, -O3	2,923	7,197	23	1,302	13	19,469
llvm-22382	Crash	3.2 – 3.6, 3.7 (trunk)	-Os, -O2, -O3	4,813	2,147	19	1,432	0	29,805
llvm-22704	Crash	3.6, 3.7 (trunk)	-O1, -Os, -O2, -O3	3,684	23,250	11	981	12	28,740

Table 3: The result of running Athena and Orion on the bugs that affect stable releases for one week. Columns (5), (6) and (7) show the SLOC (in BSD/Allman style) of the original seed program, the bug-triggering variant, the reduced file used to report the bug. Column (8) shows the size of the database (*i.e.*, the number of <context, statement> pairs) constructed from the seed program. Column (9) shows the number of bugs Athena rediscovered. Column (10) shows the total number of variants Athena generated. Orion rediscovered two *shallow* bugs: LLVM 20494 and 21512.

Shallow Bugs During this period, Orion was able to only discover two LLVM bugs (20494 and 21512). Athena also rediscovered these bugs. Interestingly, these bugs were rediscovered most *often* (2,660 and 988 times). This indicates that they are shallow bugs, which can be triggered using simple mutations. Orion failed to find bugs in the other seed programs. These bugs are deep bugs, which require sophisticated sequences of mutations.

Deep Bugs Despite generating 28K variants on average for a seed program, Athena was unable to rediscover four of our bugs. Some other bugs were rediscovered only a few times. These bugs are indeed quite deep and require specific sequences of mutations. This is understandable because the search space is vast and our process is nondeterministic.

The sizes of the triggering variants vary. The variants are often larger than the original programs. Some are significantly larger because we may happen to insert some large chunks of code (such as bugs gcc-60382 and llvm-22704). On the other hand, some are smaller because we delete some large chunks of code (such as bugs gcc-64495 and llvm-22382).

This experiment confirms that Athena is more powerful than Orion in terms of bug detection. Indeed, Orion is Athena taking away insertion and limiting the length of random walks to one. However, Athena may take more time than Orion to find bugs. If the bug is shallow, Orion may find it faster because it only explores the smaller nearby neighborhood of a seed program.

6.4 Coverage Improvement

We now evaluate the line coverage improvement of Athena on GCC and LLVM in comparison with Orion. The baseline is the coverage of executing 100 Csmith seed programs, on which GCC and LLVM achieve 34.9% and 23.5% coverage ratios respectively. We measure coverage using variants produced by Orion and Athena from these seeds. To evaluate the impact of the number of variants on code coverage, we also vary the number of variants for each seed.

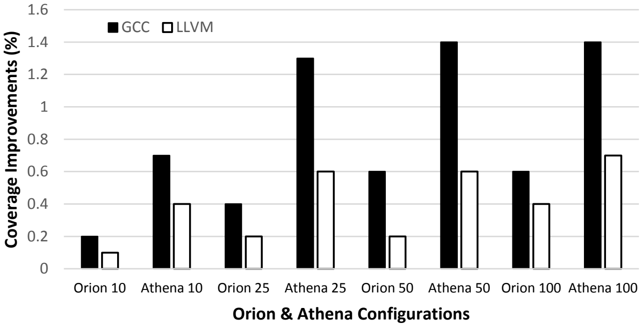

Figure 4: Improvement in line coverage of Orion and Athena while increasing the number of variants. The baseline is the coverage of executing 100 Csmith seeds, where GCC and LLVM have respectively 34.9% and 23.5% coverage ratios.

Figure 4 shows the coverage improvement of Orion and Athena over the baseline. Although both Athena and Orion help increase line coverage, Athena is strictly better than Orion. In particular, 10 Athena variants yields slightly better coverage than 100 Orion variants. This is expected because Athena generates more diverse test programs.

6.5 Assorted Sample Bugs Found by Athena

We now present six GCC and LLVM bugs to illustrate the diversity of our reported bugs.

```c
int printf(const char *, ...);
static short a = 0;
short b = -1;
static unsigned short c = 0;

int main (){
  if (a <= b)
    printf ("%d\n", c);
  return 0;
}
```

(a) GCC trunk miscompiles this program at `-Os` and above in both 32-bit and 64-bit modes. The compiled executable prints `0`, which is unexpected.

```c
int a;
int b;
int c[1];
int d;
int f;

void foo () {
  for (; b; b++)
    c[0] = c[f] && (d = a);
}
```

(b) GCC trunk crashes with a segmentation fault when compiling the program at `-O3` in both 32-bit and 64-bit models.

```c
int a, b, d;
void fn1 () {
  for (b = 0; b < 9; b++) {
    int e;
    for (d = 0; d < 5; d++) {
      a &= 231; a ^= 14;
    }
    e = (a ^= 1) < 0;
  }
}
```

(c) This is a performance regression bug. It takes GCC trunk several minutes to compile this small program if flags `-O3` and `-g` are specified.

```c
int a;
int b;
int d;
void foo () {
  for (b = 0; b < 9; b++) {
    int e;
    for (d = 0; d < 5; d++) {
      a &= 231;
      a ^= 14;
    }
    e = (a ^= 1) < 0;
  }
}
int main() {
  foo(); return 0;
}
```

(d) Clang trunk miscompiles this program at `-O3` in both 32-bit and 64-bit modes. The compiled executable aborts with a floating point exception (modulo by zero).

```c
int a;
int b;
int fn1 () {
  if (b)
    goto lbl;
  fn1 ();
  return 0;
lbl:
  a = 0;
  fn1 ();
  return 0;
}

int main() {
  return fn1();
}
```

(e) Clang trunk crashes when compiling the test program at `-O1` and above in both 32-bit and 64-bit modes. The bug is in tail call elimination pass.

```c
int a, c, d;
unsigned int b;
static int e[1];
long long f;
long long fn1 () {return a;}
void fn2 () {
  int g;
  while (c) {
    f = b; g |= f;
    while (c) g = e[0];
    g |= fn1 ();
  }
}
int main () {
  fn2 (); return 0;
}
```

(f) Clang hangs when compiling this test program at `-Os` and above in both 32-bit and 64-bit modes.

Figure 5: Example test programs uncovering a variety of GCC and LLVM bugs.

Figure 5a The GCC trunk miscompiled the program at `-Os` and above in both 32-bit and 64-bit modes. The miscompiled executable prints `0`. This behavior is unexpected because the print statement is guarded by the predicate `a <= b`, which is evaluated to false.

The bug is in the ICF (interprocedural constant folding) pass. Both variables `a` and `c` are read-only and have the same value `0`. GCC folds them by converting `a` to an alias to `c`. However, `c` is unsigned whereas `a` is signed. This signedness difference later confuses the pass *conditional constant propagation*, making it incorrectly conclude that the conditional expression `a <= b` is always `true`.

Figure 5b The test program crashes the GCC trunk with a segmentation fault when it is compiled at `-O3` in both 32-bit and 64-bit modes. The bug is due to a missing check in the tree optimization pass IFCVT that transforms the body of a loop to vectorize it. This pass seeks a certain ϕ statement (one containing two operands: a value defined by a statement within the loop body, and another ϕ value), and converts the matched ϕ statement into an if-expression. The compiler crashes when it is checking the ϕ statement `d.2 = `ϕ`(a.1, d.1)` that is generated from the assignment `d = a`. The first operand `a.1` is from an assignment `a.1 = a` before the loop. Because the optimization pass assumes that `a.1` is generated within the loop, a memory error occurs.

Figure 5c This small test program reveals a severe performance regression in GCC. It takes GCC several minutes to compile the program at `-O3` with debugging information retained (specified by the flag `-g`). While GCC unrolls

the loop body, it keeps the debugging information in all unrolled instances.

Figure 5d The Clang trunk miscompiles the test program at -O3. The compiled executable aborts with a floating-point exception (modulo by zero).

This bug involves the loop vectorizer pass and the peephole optimization pass of LLVM. Initially, the loop body is vectorized including the modulo statement (a % d). However, in the later peephole pass (*i.e.*, instcombine), vector shuffle instructions that are necessary to perform the modulo operation are eliminated, leading to the trap.

Figure 5e The test program crashed Clang trunk at -O1 and above in both 32-bit and 64-bit modes.

The cause of this bug is in the tail call elimination pass of LLVM. After internal transformation (*i.e.*, folding return statements into unconditional branches), some basic blocks are cleared but later requested to provide their terminator instructions. This causes an assertion failure.

Figure 5f The test program triggered Clang to hang with the flags -Os and above in both 32-bit and 64-bit modes.

The bug is in the peephole optimizer of LLVM. The specific optimizer FoldOpIntoPhi creates an infinite loop. The optimization it performs triggers other optimizations in instcombine, which in turn leads back to FoldOpIntoPhi.

7. Related Work

This section surveys representative, closely related work.

Compiler Testing Compiler testing still remains the dominant technique for validating production compilers. Every major compiler (such as GCC and LLVM) has its own regression test suite maintained along with its development. There are also some commercial test suites available for checking compiler conformance and correctness such as PlumHall [19], SuperTest [1].

The alternative is to use random testing to complement these manually written test suites. Recent work by Nagai *et al.* [16, 17] generates random arithmetic expressions to find bugs in C compilers' arithmetic optimizations. They have found several bugs in GCC and LLVM. CCG is a random C program generator that focuses on finding compiler crashing bugs [3]. Csmith is another C program generator that can find both crashing and miscompilation bugs [4, 21, 30]. Based on the idea of differential testing [14], Csmith randomly generates C programs and checks for deviant behavior across compilers or compiler versions. Csmith is very successful: it has found a few hundred GCC and LLVM bugs over the last several years. Athena complements the above techniques.

要不要尝试专攻一下编译器某些类别的bug
？
算数表达式

CLsmith is a system built on top of Csmith to validate OpenCL compilers [5]. It also applies EMI to enrich the generated test programs. Proteus [10] is another system based on Csmith and Orion to specifically stress test the link-time optimizers of compilers. Although we only demonstrate the effectiveness of our technique in testing ordinary C compilers, the mutation strategy in Athena is general, orthogonal and can also be applied to both domains to generate more effective test programs.

Link-time

Markov Chain Monte Carlo MCMC sampling is a popular algorithm that has played a significant role in science and engineering. People have applied MCMC techniques to solve a large number of problems in statistics, economics, physics, biology, and computer science [2, 6].

In computer science, Schkufza *et al.* recently applied MCMC sampling to perform superoptimization tasks, which transform a loop-free sequence of binary statements into a more optimized sequence [23, 24]. They propose a cost function that captures both correctness and performance, and some basic operations to transform the program such as replacing the opcode or operand, or swapping the statements. The cost function guides their program space exploration toward lower cost (*i.e.*, more optimized) programs.

MCMC在软件工程领域的应用

Our technique also uses MCMC sampling to explore the space of programs but with several distinct differences: (1) we target a different domain (*i.e.*, compiler testing); (2) the cost function in our work is to maximize the diversity of generated test programs; and (3) the eligible transformations generate EMI variants instead of instruction sequences.

Verified Compilers A verified compiler guarantees that the compiled code is semantically equivalent to its source program. To achieve this goal, each verified compiler is accompanied by a correctness proof that ensures semantic preservation. CompCert [11, 12] is the most notable example of verified compilers. CompCert is mostly written in the Coq specification language, and its correctness is proved by the Coq proof assistant. Zhao *et al.* proposed a new technique to verify SSA-based optimizations in LLVM — a production compiler — using the Coq proof assistant [31]. Malecha *et al.* applied the idea of verified compilers to the database domain and demonstrated preliminary results on building a verified relational database management system [13].

Translation Validation It is usually easier to prove that a particular translation of a source program is correct than to verify the correctness of all possible translations (of all possible source programs). This is the motivation behind translation validation, which aims to verify that the compiled code is equivalent to its source on-the-fly. Hanan Samet introduced the idea of translation validation in his PhD thesis [22]. Pnueli *et al.* did an early work on translation validation to validate the non-optimizing compilation from SIGNAL programs to C programs [20]. Subsequently, Necula [18] extended it to handle optimizing transformations and validates four optimizations in GCC 2.7. Tristan *et al.* adapted the work on equality saturation [26] to validate intraprocedural optimizations in LLVM [28].

8. Conclusion

We have presented a novel EMI mutation strategy to expose deep bugs in production compilers. Our technique uses the MCMC algorithm guided by an objective function to sample the space of test programs effectively. Athena, our realization targeting C compilers, has found 72 GCC and LLVM bugs in 19 months, most of which have been fixed.

We continue actively using Athena to stress-test GCC and LLVM, and plan to extend our testing to programs derived from real-world projects. The key challenge is to effectively reduce such large test programs after bugs are found, which typically involve multiple directories and source files.

9. Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback on an earlier draft of this paper. Our special thanks go to the GCC and LLVM developers for analyzing and fixing our reported bugs. Our evaluation also benefited significantly from Berkeley Delta [15], and University of Utah's Csmith [30] and C-Reduce [21] tools.

This research was supported in part by National Science Foundation (NSF) Grants 1117603, 1319187, 1349528, and 1528133. 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. URL http://www.ace.nl/compiler/supertest.html.

[2] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1):5–43, Jan. 2003.

[3] A. Balestrat. CCG: A random C code generator. URL https://github.com/Merkil/ccg/.

[4] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208, 2013.

[5] 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.

[6] W. R. Gilks. *Markov Chain Monte Carlo In Practice*. Chapman and Hall/CRC, 1999. ISBN 0412055511.

[7] GNU Compiler Collection. Gcov - Using the GNU Compiler Collection (GCC). URL http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

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

[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] 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.

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

[13] 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. .

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

[15] S. McPeak, D. S. Wilkerson, and S. Goldsmith. Berkeley Delta. URL http://delta.tigris.org/.

[16] 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.

[17] 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.

[18] 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. .

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

[20] 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.

[21] 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.

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

[23] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316, 2013.

[24] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 53–64, 2014.

[25] Standard Performance Evaluation Corporation. SPEC CINT2006 Benchmarks. URL https://www.spec.org/cpu2006/CINT2006/.

[26] 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*, pages 264–276, 2009.

[27] The Clang Team. Clang 3.4 documentation: LibTooling. URL `http://clang.llvm.org/docs/LibTooling.html`.

[28] 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.

[29] Wikipedia. Jaccard index. URL `http://en.wikipedia.org/wiki/Jaccard_index`.

[30] 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.

[31] 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.