Randomized Stress-Testing of Link-Time Optimizers

Vu Le Chengnian Sun Zhendong Su
Department of Computer Science, University of California, Davis, USA
{vmle, cnsun, su}@ucdavis.edu

ABSTRACT

Link-time optimization (LTO) is an increasingly important and adopted modern optimization technology. It is currently supported by many production compilers, including GCC, LLVM, and Microsoft Visual C/C++. Despite its complexity, but because it is more recent, LTO is relatively less tested compared to the more mature, traditional optimizations. To evaluate and help improve the quality of LTO, we present the first extensive effort to stress-test the LTO components of GCC and LLVM, the two most widely-used production C compilers. In 11 months, we have discovered and reported 37 bugs (12 in GCC; 25 in LLVM). Developers have confirmed 21 of our bugs, and fixed 11 of them.

Our core technique is differential testing and realized in the tool Proteus. We leverage existing compiler testing tools (Csmith and Orion) to generate *single-file* test programs and address *two important challenges* specific for LTO testing. First, to thoroughly exercise LTO, Proteus automatically transforms a single-file program into multiple compilation units and stochastically assigns each an optimization level. Second, for effective bug reporting, we develop a practical mechanism to reduce LTO bugs involving multiple files. Our results clearly demonstrate Proteus's utility; we plan to make ours a continuous effort in validating link-time optimizers.

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, link-time optimizer, automated testing

ISSTA'15 , July 12–17, 2015, Baltimore, MD, USA

1. INTRODUCTION

Compilers are among the oldest, most complex and important software. Perhaps the most critical component in any compiler is its optimizer, which determines the overall performance of its generated code. Decades of fruitful research in compiler optimizations have led to many powerful techniques that have helped improve the optimizer performance significantly. We can categorize an optimization as *intraprocedural* (which is performed inside a function), *interprocedural* (which relates several functions), or *whole-program* (which takes into account the entire program).

The traditional workflow of a compiler is to compile and optimize each file or module individually, and link the compiled modules into a single executable file. This approach enables intra- and sometimes also interprocedural optimizations (as a file/module may contain multiple functions), but not whole-program optimizations, because the compiler does not have sufficient information about other modules at compile time. This brings an opportunity for the linker to perform whole-program optimizations at link time, when all information about the program becomes available. This process is referred to as *link-time optimization* (LTO).

Link-Time Optimization LTO is an increasingly important and adopted optimization technology: it is simple to use (without needing to modify one's build process) and effective (providing substantial speedups). Studies [2, 5] have shown that enabling LTO helps reduce code size by 15-20% and increase speed by 5-15% on average. LTO is supported by many modern compilers, *e.g.*, GCC, LLVM, and Microsoft Visual C/C++. Inevitably, it is also very sophisticated as it performs whole-program analyses and optimizations.

To enable LTO, a compiler needs to alter its compilation process. An LTO-compiled object file contains additional intermediate representation besides the normal compiled object code. For example, GCC writes the GIMPLE (GCC's intermediate language) representation of the source file to the output object file when LTO is enabled. Similarly, LLVM allows writing bitcode representation to the object file. At link time, the linker gathers this extra information and performs whole-program optimization, which becomes feasible because all variable and function definitions are available. We enable LTO in GCC and LLVM using the flag `-flto`.

State-of-the-Art Compiler Validation Because optimization is a critical pass, people have devoted considerable efforts to its validation [8, 15, 16, 24, 25]. Two notable examples are Csmith [25] and Orion [8]. To stress-test C compilers, Csmith generates random programs from scratch, while Orion modifies an existing test program to generate

more test variants that are semantically equivalent under a given input. Both have been extremely effective—each has found hundreds of reported bugs in GCC and LLVM.

However, these efforts target only traditional optimizations. There is little attention on validating LTO, the increasingly important component of any modern compiler.

Proteus: Randomized LTO Testing This paper presents Proteus[1], a randomized differential testing technique to stress-test link-time optimizers, and the first extensive effort to stress-test LTO in GCC and LLVM. Two key challenges arise:

1. How to obtain LTO-relevant test programs, which typically involve multiple compilation units?
2. How to effectively reduce the bug-triggering test programs?

To tackle the first challenge, we use Csmith to generate *single-file* test programs. We then automatically transform each test program in two semantics-preserving manners. First, we extend Orion to *inject arbitrary function calls to unexecuted code regions* to increase function-level interprocedural dependencies. The increased dependencies stress-test LTO more thoroughly. Second, we split each test program into separate compilation units. We compile each of these compilation units at a random optimization level, and finally link the object files also at a random optimization level.

As for the second challenge, we develop an effective procedure to *reduce multiple-file test programs* that trigger bugs. The traditional reduction approach for LTO bugs is to reduce each compilation unit individually, which is *extremely inefficient*, and more importantly, *difficult to ensure test program validity* (*i.e.* rejecting tests with undefined behavior). In fact, test-case reduction has often been a neglected real challenge for bug reporting (we believe that more work should be done and encouraged).

Our key observation is that the bug-triggering property of our splitting function is preserved under reduction, which allows us to perform reduction on the original single-file test program. Indeed, after reduction, we split a reduced test program into separate compilation units, and check for bug-triggering behavior. This approach works very well in practice. Most of our tests were reduced within several hours. In comparison, existing reduction techniques take days or weeks, or never terminate, and produce invalid reduced tests.

In 11 months of continuous testing, we have reported 37 bugs, among which 21 have been confirmed, and 11 have been fixed. We are yet to report many others, because they may be duplicate to our reported bugs that are not fixed. We are waiting for the developers to fix our bugs before reporting new ones.

Contributions We make the following main contributions:

- We introduce Proteus, the first randomized differential testing technique to stress-test link-time optimizers, and demonstrate that Proteus is extremely effective in finding LTO bugs in GCC and LLVM.
- We propose a practical procedure to reduce LTO bugs that is efficient (*i.e.* significantly shortening reduction time) and effective (*i.e.* reliably rejecting invalid tests).
- We report our results in finding and reporting LTO bugs. The GCC developers have fixed all but one of our 12 reported bugs, four of which were marked as P1—the most severe, release-blocking type of bugs. The LLVM developers confirmed 9 bugs, but they have not fixed any of them yet.

[1]Proteus is a Greek sea god who can *assume different forms*.

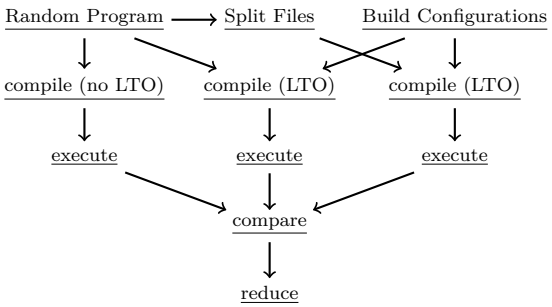

Figure 1: Overview of Proteus's approach.

The remainder of the paper is structured as follows. Section 2 illustrates our approach via two of our reported bugs, one for GCC and one for LLVM. Section 3 discusses the details of Proteus's design and implementation. We present our experimental results in Section 4. Finally, we survey related work (Section 5) and conclude (Section 6).

2. ILLUSTRATIVE EXAMPLES

Generally, we can categorize compiler bugs into two classes: crash/hang and miscompilation. In the first class, the compiler aborts due to unexpected runtime errors or does not terminate due to implementation defects. A compiler crash/hang is undesirable, but a miscompilation can be much more harmful as it causes the compiler to *silently* compile the source code *incorrectly*. These bugs are the most dangerous because the compiler subverts developers' intent, causing their programs to misbehave.

Our tool, Proteus, detects both types of bugs in compilers' LTO components. Unlike traditional optimizations which are performed during compilation, LTO takes place at link time. It further complicates compilers, as they need to write intermediate representations to object files, read them back in, and perform whole-program analyses.

Figure 1 shows the overview of Proteus's approach. It starts with a single-file program p (generated by Csmith or extended Orion), and compiles p in three different ways:

1. p is directly compiled without LTO.
2. p is compiled with LTO under various compilation and linker flags.
3. p is split into multiple compilation units (each corresponds to a function), which are separately compiled under different optimization flags and linked with LTO.

Proteus then executes these compiled programs and compares the execution results. Any inconsistency indicates a bug.

We next demonstrate this process via two concrete bugs, one for GCC and one for LLVM.

GCC Bug #60404 Figure 4c shows Proteus's steps to find this bug. The original program, generated by Orion, prints 0 — the expected value of `a[b]` — and terminates (Figure 2a). Note that after the call `fn2(0)` in the function `main`, the value of variable `b` remains unchanged (*i.e.*, 0).

```c
/*** small.c ***/
#include <stdio.h>
int a[1] = { 0 }, b = 0;

void fn1 (int p) { }
void fn2 (int p) {
  b = p++;
  fn1 (p);
}
int main () {
  fn2 (0);
  printf ("%d\n", a[b]);
  return 0;
}
```

(a) A simplified version of a program generated by Orion (the original version has 2367 lines of code). All compilers under test compile it correctly.

```
/*** configuration ***/
gcc -flto -O1 -c fn1.c
gcc -flto -O1 -c fn2.c
gcc -flto -O1 -c main.c
gcc -flto -O1 -c t.c
gcc -flto -O0 fn1.o fn2.o main.o t.o
```

(b) A build configuration that triggers a GCC LTO bug on split files in Figure 2c.

```c
/*** small.h ***/
#include <stdio.h>
int a[1], b;
void fn1 (int p);
void fn2 (int p);

/*** small.c ***/
#include "small.h"
int a[1] = { 0 }, b = 0;

/*** fn1.c ***/
#include "small.h"
void fn1 (int p) { }

/*** fn2.c ***/
#include "small.h"
void fn2 (int p) {
  b = p++;
  fn1 (p);
}

/*** main.c ***/
#include "small.h"
int main () {
  fn2 (0);
  printf ("%d\n", a[b]);
  return 0;
}
```

(c) Files split from the code in Figure 2a.

```c
/*** fn1.c ***/
void fn1 (int p) { }

/*** fn2.c ***/
extern int b;
extern void fn1 (int);
void fn2 (int p) {
  b = p++;
  fn1 (p);
}

/*** main.c ***/
int printf (const char *, ...);
extern void fn2 (int);
int a[1], b;
int main () {
  fn2 (0);
  printf ("%d\n", a[b]);
  return 0;
}
```

```
/*** configuration ***/
gcc -flto -O1 -c fn1.c
gcc -flto -O1 -c fn2.c
gcc -flto -O1 -c main.c
gcc -flto -O0 fn1.o fn2.o main.o
```

(d) Cleaned up files with the bug-triggering configuration for bug reporting.

Figure 2: Proteus's workflow: from original file (in Figure (a)) to split files (in Figure (c)) to reported files (in Figure (d)). GCC revision 208268 miscompiles these files. The compiled program returns 1 instead of 0. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=60404`)

However, the GCC development trunk (revision 208268) miscompiles the files that are split from the original file by Proteus (Figure 2c), under the build configuration shown in Figure 2b. The compiled program in this case prints 1 instead of the expected 0. In this program, `b` was incorrectly assigned the value 1 after invoking `fn2(0)`. The printed value `a[b]` (or `a[1]`) is the memory location right after the boundary of the array `a`, which is coincidentally `b` (or 1). Figure 2d shows the files and the build configuration that we used for reporting after some cleaning up.

The bug happens because while coalescing SSA (single static assignment) parameter variables, the GCC developer mistakenly assumes that intermediate representation (IR) in object files are compiled without optimization, or without LTO. When the assumption is invalid, GCC misinterprets the IR stored in object files and generates incorrect code.

In this example, GCC compiles `fn2.c` at `-O1` to produce `fn2.o`, whose *optimized* IR is similar to the following:

```c
void fn2 (int p) { p_1 = p + 1; b = p; fn1 (p_1); }
```

When GCC links the program with `-flto -O0`, it assumes that the IR of `fn2.c` is unoptimized and that all the SSA variables originated from the parameter `p` are already coalesced into a single partition. As the assumption is wrong, GCC fails to allocate a memory location for `p_1`, and consequently both `p` and `p_1` share the same address. The miscompiled code is similar to the following:

```c
void fn2 (int p) { p = p + 1; b = p; fn1 (p); }
```

GCC correctly compiles the original single-file program and its split files without LTO because in these cases, the complication involving writing/reading/linking multiple IRs do not arise.

```c
/*** small.c ***/
int a, b = 1, c;
int fn1 (unsigned char p1, int p2) {
  return p2 || p1 > 1 ? 0 : p2;
}
int main () {
  int d = 0;
  for (; a < 1; a++) {
      c = 1;
      fn1 ((b &= c) | 10L, d);
  }
  return 0;
}
/*** configuration ***/
clang -flto -O0 -c small.c -o small.o
clang -flto -O0 small.o
```

Figure 3: LLVM 3.4 and trunk revision 204228 miscompile this program at -O0. The compiled executable hangs instead of terminating. (`http://llvm.org/bugs/show_bug.cgi?id=19201`)

LLVM Bug #19201 Figure 3 shows an LLVM LTO bug, triggered by a test program generated by Orion. The test program is clearly well-defined according to the C standard. It should execute and terminate normally. However, both LLVM 3.4 and its development trunk miscompile the code with LTO enabled at `-O0`, resulting in a non-terminating program. With LTO disabled the program is correctly compiled.

It is evidently a bug in LLVM, because the program is valid and the semantics of the program compiled with LTO is inconsistent with that of its non-LTO counterpart. The LLVM developers have yet to comment on the root cause of the bug. To shed some light on this bug, we have disassembled the miscompiled program and inspected its assembly code. LLVM compiles the program into an infinite loop with an empty body, similar to the following:

```
int main() {  while(1) {}; return 0; }
```

We suspect that LLVM mistakenly concludes that the program contains undefined behavior, and consequently generates a non-terminating loop. This happens because there is no restriction on compilers for compiling programs having undefined behaviors. Compilers are only obligated to consider valid programs.

3. DESIGN AND REALIZATION

This section describes our approach and realization of Proteus. At the high level, Proteus first leverages Csmith and Orion to generate single-file test programs to enable later phases of our differential testing of LTO. In particular, we use the generated programs to seed the following two-step process: (1) we modify Orion to insert arbitrary function calls to unexecuted code regions to increase function-level dependencies; and (2) we divide a single test program into separate compilation units. Both steps are semantics-preserving and designed to specifically target LTO testing. Our goal is to find build configurations that lead to deviant behavior.

DEFINITION 3.1 (SPLIT FUNCTION). *The function Split takes as input a single program and divides it into the followings: (1) a header file that contains all type definitions and global variable/function declarations; (2) an initialization source file that contains all initializations of global variables (this file includes the header file); (3) a set of source files, each of which contains one function definition from the original source file (these files also include the header file).*

For example, Split divides a single file in Figure 2a to a header file, an initialization file, and a set of function files in Figure 2c. The transformation imposed by Split does not change the semantics of the original program.

DEFINITION 3.2 (BUILD CONFIGURATION). *A build configuration specifies how to compile and link a set of source files into a single executable. It compiles each source file into an object file and links all the object files into a final executable. Each compilation and linking step is parametrized over a set of optimization flags.*

For instance, Figure 2b is a build configuration that triggers a bug in GCC while compiling the files in Figure 2c.

3.1 Differential Testing of LTO

Proteus uses differential testing to find LTO bugs. The traditional view of differential testing [12] is quite simple: If two systems under test behave differently on some input, it indicates a bug in one of the systems, or both. Csmith has implemented this view [25]. It generates random C programs and seeks for deviant behavior in different C compilers while compiling/running the same source program.

Orion introduces an alternate view on differential testing [8]. It profiles the execution of a program P under some input I. It then generates many variants of P by randomly pruning unexecuted statements in P. Since these variants are equivalent *w.r.t.* I (*i.e.*, they produce the same output under the input I), Orion seeks for deviant behavior in a compiler while compiling/running these variants on I.

We take yet another view of differential testing, based on the observation that all compiled programs built from different build configurations are semantically equivalent. Proteus seeks for deviant behavior in a compiler while compiling/running the program (and its split version) under different LTO build configurations. Similar to the view in Orion, this view has some attractive advantages over the original (and Csmith's) approach. Proteus can operate on existing code base (either real or randomly generated), and it can validate a single compiler in isolation (where competing compilers do not exist).

Algorithm 1: Proteus's main procedure

```
1  procedure Validate (Compiler Comp, TestProgram P,
   InputSet I):
2      begin
           /* Calculate expected output */
3          P_exe := Comp.Compile(P)
4          IO := {⟨i, P_exe.Execute(i)⟩ | i ∈ I}
           /* Perform differential testing */
5          DiffTest(Comp, P, IO)
6          DiffTest(Comp, Split(P), IO)
7  procedure DiffTest (Compiler Comp, TestPrograms 𝒫,
   InputOutputSet IO):
8      begin
           /* Generate configs and verify */
9          for 1..MAX_ITER do
10             σ := GenerateRandomBuildConfig(𝒫)
11             P'_exe := Comp.Compile(𝒫, σ)
12             if ∄P'_exe then
13                 ReportCrashHang(Comp, σ, 𝒫)
14             else
15                 foreach ⟨i, o⟩ ∈ IO do
16                     if P'_exe.Execute(i) ≠ o then
17                         ReportMiscomp(Comp, σ, 𝒫, i)
```

3.2 Implementation

Algorithm 1 describes the main procedure for finding LTO bugs from a test program in Proteus. It takes as input a compiler under test $Comp$, a program P and a set of its input I, and searches for build configurations that trigger LTO bugs on P (or its split files) under some input in I.

The algorithm consists of two main steps. First, Proteus calculates the expected output of the original program built without LTO (lines 3–4). It then searches for inconsistent behaviors in the compiler when LTO is enabled while compiling/running the program and its split files (lines 5–6). The loop on line 9 randomly generates build configurations (line 10) and checks for any deviance from the expected behavior (lines 12–17). Proteus reports a crashing or hang bug, if $Comp$ crashes or hangs during the build process (line 13), or a miscompilation bug, if the running output is different from the expected output on some input (line 17).

Algorithm 1 is realized as a shell script. We implement the function Split in C++ using LLVM's LibTooling library [22]. This implementation follows the function's de-

scription in Definition 3.1. Similarly, the implementation of `GenerateRandomBuildConfig` follows the description of build configuration in Definition 3.2. We assign a random optimization flag to each compilation or linking step to generate a random build configuration.

Proteus is simple to realize, yet very effective in finding LTO bugs (see Section 4). Our implementation contains only approximately 300 lines of bash scripts and 200 lines of C++ code. Its simplicity makes Proteus general and applicable to other language settings.

3.3 Bug Reduction

Once Proteus finds a bug, we need to reduce it before filing a report in the affected compiler's bug database. This step is important because developers usually ignore large test cases or specifically ask reporters to reduce their test cases further.

We can automate this step using delta-debugging. At a high-level, delta-debugging works by gradually reducing the input program and ensuring that the reduced program still triggers the bug and is valid (*i.e.*, does not contain undefined behavior). State-of-the art delta-debugging reducers include Berkeley Delta [13] and C-Reduce [19]. Unfortunately, these reducers support reducing only a single file.

Traditional Approach to Reducing LTO Bugs Because LTO bugs normally involve multiple files, reducing them is quite challenging, especially for miscompilation bugs. In fact, the GCC official guide to reduce bugs does not even have any instruction for reducing LTO miscompilation bugs [6], forcing the developers to rely on their experience to craft their own reduction strategies.

The standard approach to reducing miscompilation LTO bugs is to reduce each file individually (*e.g.* with Delta or C-Reduce). This is very inefficient and error-prone as these files are normally interdependent. While reducing a file, reduction tools cannot remove constructs used in other files, as this would invalidate the integrity of the program. Therefore, the reduction results are usually unsatisfactory.

Moreover, we do not have a reliable way to detect undefined behavior in multiple-file programs. For example, the C interpreter in CompCert, which can detect undefined behavior, supports only single-file programs. This makes reducing LTO bugs even more challenging. Because we cannot check for program validity during reduction, the reduced program may contain undefined behavior and become invalid.

Reducing LTO Bugs in Proteus Fortunately, in our settings, by design the splitting function Split has a special property that allows us to perform reduction on a single file, which significantly improves reduction effectiveness.

PROPOSITION 3.1. *The bug-triggering property of Split is preserved under reduction. That is,*

$$\forall P \forall Comp \forall \sigma \forall i : Bug(Comp, \mathit{Split}(P), \sigma, i) \rightarrow \exists P' : P' = \Delta(P) \wedge Bug(Comp, \mathit{Split}(P'), \sigma, i)$$

where:

P is the program whose split files trigger the bug,
$Comp$ is the compiler affected by the bug,
σ is the bug-triggering build configuration,
i is the bug-triggering input, and
Δ is the reduction function.

The above claim states that, if the split files of a program P trigger a bug under some build configuration σ and some input i, the reduction tool Δ will produce a reduced (*i.e.* smaller) program P' such that its split files also trigger the same bug—to be precise the *same manifested bug characteristics*, *i.e.* crash or miscompilation—under the same configuration and input.

We leverage this property to reduce LTO bugs found by Proteus. Our reduction script first applies delta reduction on the original single file. It then uses Split to separate the reduced file, and checks for bug-triggering behavior on the split files. We carefully design the build configuration so that it is always valid for the reduced split files, although our reduction tool may eliminate several functions (and thus their corresponding split files).

Test Reduction Efficiency Our delta tool is a meta-reducer. It involves a nested loop that invokes Berkeley Delta followed by C-Reduce. The loop terminates if a fixpoint is reached (*i.e.*, the file is not reduced any further). Our meta-reducer strikes a nice balance of Berkeley Delta's efficiency and C-Reduce effectiveness. It has another crucial benefit, to maintain test case validity, which we discuss next.

Reduced Test Validity During reduction, we need to reject reduced programs that contain undefined behavior. We follow C-Reduce's and Orion's approaches in rejecting these programs. In particular, we leverage GCC and LLVM warnings and Clang's undefined behavior sanitizer. Whenever applicable, we also heavily rely on CompCert and its C interpreter because the compiler has added support for more language features. In the process, we also found two interesting bugs in CompCert (see Section 4.4). Since our reduction does not require human intervention, we chain it with Proteus to create a unified automatic bug finding and reducing process, which we discuss next.

3.4 Automatic Bug Finding and Reducing

Our design goal is to minimize human involvement in the process of finding and reducing bugs. To that end, we create a script to automate this process.

Our script involves a loop, where in each iteration, it invokes Csmith/Orion to generate a random C program, and uses Proteus to find bugs derived from this program. If Proteus finds a bug, the script generates the bug's reduction script, and invokes the reducer. The script sends a notification when the reduction process terminates.

During this process, it is important to avoid reporting duplicated bugs. Two bugs are duplicate if they share the same root cause. Unfortunately, it is difficult to decide if two bugs have the same cause because it requires deep knowledge of the compilers' internals. As an approximate duplicate check, we use bug signatures that involve the bug-triggering compiler setups. We cluster bugs *w.r.t.* their signatures and report only one representative from each cluster. After a bug has been fixed, we run the compiler again on other bugs from the same cluster. Any bugs that do not manifest anymore are likely duplicates. If a bug still manifests, it is different. We then reduce it and file a separate report.

Bug signature A bug signature is a vector of binary elements, each corresponding to the result (*i.e.*, trigger bug/-does not trigger bug) of running a particular compiler setup.

The first kind of signature is *compiler signature*, in which setups are obtained by varying the compilers and compiler versions. We use two compilers (GCC and LLVM) and their three most recent releases in addition to the development trunk. We check in both 32-bit (`-m32`) and 64-bit (`-m64`) modes, with LTO enabled/disabled (use `-flto` or not).

For each compiler setup (compiler, version, mode, LTO enabled/disabled), we (1) test the original program in all optimization levels (`-O0`, `-O1`, `-O2`, `-Os`, `-O3`), (2) test split files using the same optimization level over all levels, and (3) test split files using the randomly generated build configuration.

The second kind of signatures is *optimization signature*, which is obtained by varying the optimization flags in the build configuration after reduction is complete. In particular, we enumerate all possible combinations of optimization flags for all compilation and linking steps in the build configuration. This is feasible because normally after reduction, the number of functions is very small (from 2 to 4 functions). The byproduct of this step is the *minimal* build configuration that triggers the bug (*i.e.*, the build configuration that uses the least optimization). We report the bug with this build configuration. We also check previous compiler versions for potential regression bugs.

Note that these two signatures may be imprecise. They may misclassify bugs and cause real bugs to slip through. However, they help us prioritize our resources on more interesting bugs in case there are many more inconsistencies than what we can feasibly process.

4. EVALUATION

We started our experiments with Proteus from the end of February 2014. We focus on testing two mainstream open-source C compilers—GCC and LLVM—because of their open bug tracking systems. This section describes the results of our testing effort in about 11 months.

Result Summary Proteus is very effective:

- *Many detected bugs*: Proteus has detected 37 bugs in GCC and LLVM. Developers have confirmed 21 of our bugs. Eight out of the 12 GCC bugs were discovered from split programs, while Csmith and Orion alone would fail to discover these. Thus, the results highlight the utility and effectiveness of Proteus.
- *Many long-latent bugs*: Many of the detected bugs have been latent in old versions of GCC and LLVM. These bugs had resisted all traditional validation approaches. This further emphasizes Proteus's effectiveness.
- *All but one reported GCC bugs are fixed*: So far, 11 out of our 12 reported GCC bugs have already been fixed.
- *Diversified bugs*: Proteus has found many kinds of bugs in GCC and LLVM's LTO components. The majority are miscompilations, the most serious kind.

4.1 Testing Setup

We first describe our set up for Proteus to find LTO bugs before discussing our empirical results.

Hardware and Compilers We focus our testing on the x86-linux platform due to its popularity and our ease of access. We perform our testing on two machines (one 18 core and one 6 core) running Ubuntu 12.04 (x86_64). We only test the daily-built development trunks of GCC and LLVM. Once Proteus has found a bug in a compiler, we also validate the bug against other major versions of that compiler.

Test Programs We build our LTO test corpus by leveraging Csmith-generated programs and their variants generated by Orion. Traditionally, Orion only generates variants by removing unexecuted statements. We modify Orion to also allow inserting arbitrary function calls to these unexecuted area. We then further split these programs into separate compilation units. We run Csmith in its "swarm testing" mode [7] to maximize its effectiveness.

Real-world projects are an interesting source to test compilers. We can certainly apply Proteus on these projects to detect LTO bugs. However, reducing these projects is very challenging as they usually involve many files, each of them can be very large. It is also more difficult to detect undefined behavior in these projects. This explains why we have only focused on randomly generated programs. We leave as future work how to test with real-world projects and how to reduce any detected bugs.

Build Configurations While generating build configurations, we only use the popular compiler optimization flags (*i.e.* `-O0`, `-O1`, `-Os`, `-O2`, and `-O3`). Each compilation and linking step is assigned with the `-flto` flag to enable LTO. We consider generating build configurations for both 32-bit (`-m32`) and 64-bit (`-m64`) environments.

For a single program, we are able to enumerate all possible optimization flags to generate build configurations. However, this is infeasible in case we split the program into separate files, because the number of files is usually large (10+ files). We need to sample this large search space and select only a certain number of build configurations to perform testing.

The number of build configurations generated for each program is a trade-off between the depth and scope of Proteus's testing. If we generate many build configurations, we may test that program more thoroughly, but we may lose the opportunity to test other programs. On the other hand, generating a few build configurations helps Proteus cover more programs, but it may not be sufficient to trigger the buggy behavior of each of the generated programs. Our empirical experience suggests that 8 build configurations strikes a good balance. We control a random parameter whose expected value is 8 and use it to generate build configurations.

4.2 Quantitative Results

Having described our testing setup, we are now ready to discuss our results using Proteus to find LTO bugs. Table 1 shows the details of our reported bugs.

Bug Count We have reported 37 bugs: 12 in GCC and 25 in LLVM. Till end of January, 2015, GCC developers have fixed 11 bugs. The LLVM developers have confirmed 9 of our bugs, but they have not fixed any of them. A number of private communications suggested that they were busy fixing internal bugs and working on Swift.

Before reporting a bug, we ensure that it has a different symptom from the previously reported bugs. However, reporting duplicate bugs is unavoidable, as compilers are complex, and bugs having different symptoms may turn out to have the same root cause. So far, we only reported one duplicated GCC bug (we did not include it in our results). Our LLVM bugs may contain duplicates, but from our experience on previous work [8], the duplication rate is low.

Importance of Reported Bugs Because we use randomly generated programs offered by Csmith and Orion to find LTO bugs, it is reasonable to ask if these bugs really matter in practice. The related discussions in Csmith [25] and Orion [8] are quite relevant here.

First, developers have acknowledged and fixed these bugs. GCC developers are impressively responsive; they generally confirmed our bugs within one day, and fixed them after three

Table 1: The valid reported bugs for GCC and LLVM.

Bug ID	Bug Type	Status	Orig. SLOC	Reduc. SLOC	Test Program	Affected Versions	Modes
gcc-60319	Miscomp.	Fixed	1818	11	Csmith + Split	4.6, 4.7, 4.8, 4.9-trunk	m32, m64
gcc-60404	Miscomp.	Fixed	2367	28	Orion + Split	4.9-trunk	m32, m64
gcc-60405	Crash	Fixed	3242	5	Csmith	4.9-trunk	m32, m64
gcc-60461	Link Error	Fixed	3242	37	Csmith	4.9-trunk	m32, m64
gcc-61184	Miscomp.	Fixed	2821	13	Csmith + Split	5.0-trunk	m32, m64
gcc-61278	Crash	Fixed	1446	30	Csmith + Split	5.0-trunk	m64
gcc-61602	Crash	Fixed	6659	7	Orion	5.0-trunk	m32, m64
gcc-61786	Miscomp.	Fixed	1823	26	Csmith	5.0-trunk	m32, m64
gcc-61969	Miscomp.	Fixed	1860	261	Csmith + Split	4.8, 4.9, 5.0-trunk	m32
gcc-62209	Crash	Confirmed	1495	23	Csmith + Split	4.8, 4.9, 5.0-trunk	m32, m64
gcc-62238	Crash	Fixed	4276	27	Csmith + Split	4.9, 5.0-trunk	m64
gcc-64684	Miscomp.	Fixed	1745	13	Orion+Split	5.0-trunk	m32, m64
llvm-18984	Miscomp.	Confirmed	2266	27	Csmith	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19026	Miscomp.	Confirmed	2729	10	Csmith	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19062	Miscomp.	New	3243	12	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19072	Miscomp.	New	1691	36	Csmith + Split	3.2, 3.3, 3.4, 3.5-trunk	m64
llvm-19073	Miscomp.	New	1703	29	Csmith + Split	3.2, 3.3, 3.4, 3.5-trunk	m64
llvm-19078	Miscomp.	New	3450	44	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19079	Link Error	New	3638	52	Csmith + Split	3.2, 3.3, 3.4, 3.5-trunk	m64
llvm-19093	Miscomp.	New	4815	25	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19109	Miscomp.	New	8232	24	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19111	Miscomp.	New	20556	26	Csmith + Split	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19132	Miscomp.	New	16626	23	Csmith + Split	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19138	Miscomp.	New	5122	12	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19146	Miscomp.	New	5369	19	Orion	3.4, 3.5-trunk	m32, m64
llvm-19184	Link Error	New	4310	47	Orion	3.5-trunk	m32, m64
llvm-19201	Miscomp.	New	9821	17	Orion	3.4, 3.5-trunk	m64
llvm-19202	Miscomp.	New	3043	10	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32
llvm-19219	Miscomp.	New	2169	23	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19225	Miscomp.	New	2241	21	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19830	Link error	Confirmed	1291	15	Orion	3.5-trunk	m32, m64
llvm-19885	Miscomp.	Confirmed	9748	24	Orion	3.2, 3.3, 3.4	m32, m64
llvm-19889	Miscomp.	Confirmed	8098	14	Orion	3.2, 3.3, 3.4, 3.5-trunk	m32, m64
llvm-19891	Miscomp.	Confirmed	11161	26	Orion	3.4, 3.5-trunk	m32, m64
llvm-19907	Miscomp.	Confirmed	5356	25	Orion	3.5-trunk	m32, m64
llvm-20172	Miscomp.	Confirmed	4683	59	Orion	3.5-trunk	m32, m64
llvm-20237	Miscomp.	Confirmed	3502	44	Orion	3.4, 3.5-trunk	m64

days on average. Second, some of these bugs were marked as critical. In fact, the GCC developers marked a third of our reported bugs as P1, the most severe, release-blocking type of bugs. Finally, both Csmith and Orion have encountered cases where compiler bugs triggered by real-world programs were actually linked to their reported bugs derived from random programs. We expect this to also hold for Proteus as we continue finding and reporting LTO bugs.

Bug Type We classify bugs into two main categories: (1) bugs that manifest when we compile programs, and (2) bugs that manifest when we execute the compiled programs. A compile-time bug can be a *compiler crashing bug* (*e.g.*, internal compiler errors), *complier hang bug* (*e.g.*, the compiler hangs while compiling the program), or *linking error bug* (*e.g.*, the compiler cannot link object files into an executable file). 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 refer to these bugs as *miscompilation* bugs. These bugs are the most serious, because the compiled programs silently produce wrong results.

Table 2 classifies the bugs found by Proteus according to the above taxonomy (note that Proteus has not yet encountered any hang bugs). This result shows that the majority of LTO bugs found by Proteus are miscompilation bugs, the most important type. This is expected because we specifically target an important optimization component of modern compilers: the LTO component.

Table 2: Bug classification.

	GCC	LLVM	TOTAL
Miscompilation	6	22	28
Crash	5	0	5
Link Error	1	3	4

Affected Compiler Versions Our strategy is to test only the latest development trunks of GCC and LLVM. Once a bug is found, we also use it to validate other versions. Another strategy is to test all compiler versions in parallel. We do not implement this strategy because it is much more expensive, and developers are more interested in bugs that occur in recent versions. Nonetheless, while all of our reported bugs affect the development trunk, most of them also affect earlier stable releases. These bugs had been latent for many years.

4.3 Assorted LTO Bugs

We now sample a few of our reported bugs, three each for GCC and LLVM, to provide a glimpse on the diversity of bugs

```
/*** foo.c ***/
void foo (char c) {
  for (c = 0; c >= 0; c++) ;
}
/*** main.c ***/
extern void foo (char c);
int main () {
  foo(0); return 0;
}
/*** configuration ***/
gcc -flto -O0 -c foo.c
gcc -flto -O0 -c main.c
gcc -flto -Os foo.o main.o
```

(a) GCC trunk (v4.9 rev. 208040) miscompiles this program at -Os and above. The compiled code hangs instead of terminating. (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=60319)

```
/*** small.c ***/
int printf (const char *, ...);
int a, b, c, d = 1, e, g, h;
short f;
int fn1 (int p1, int p2) {
  return -p2 < 0 ? 0 : p2;
}
int fn2 () { return b = 0; }
void fn3 (int p) {
  int i, j = d = 0;
  i = 1 ? 1 : 0;
  f = j || j || 0 > j ? 0 : j;
  if (!(1 & i && f)) {
    if (g) {}
    for (g = 0; g != 1; g++) {
      g || fn1 (fn2 (), p);
      if (!d) {
        e = g || a;
        break;
      }
    }
    // c is updated below
    for (h = -1; c >= 0; c = h)
      if (g) break;
  }
}
int main () {
  fn3 (1);
  printf ("%d\n", c);
  return 0;
}
/*** configuration ***/
clang -flto -O0 small.c
clang -flto -O0 small.o
```

(d) LLVM trunk (v3.5 rev. 203239) miscompiles this program at -O0 and -O1. The executable prints 0 instead of expected -1. (http://llvm.org/bugs/show_bug.cgi?id=19078)

```
/*** small.c ***/
struct S {
  int f1;
  int f2;
} a[1] = { {0, 0} };
int b, c;
static unsigned short fn1(struct S);
void fn2 () {
  for (; c;) ;
  b = 0;
  fn1 (a[0]);
}
unsigned short fn1 (struct S p) {
  if (p.f1) fn2 ();
  return 0;
}
int main () {
  fn2 ();
  return 0;
}
/*** configuration ***/
gcc -flto -Os -c small.c
gcc -flto -Os small.o
```

(b) GCC trunk (v4.9 rev. 208393) fails while linking this program at -Os. (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=60461)

```
/*** small.c ***/
#include <assert.h>

int *a, **volatile b = &a, c,
 *d = &c, *e = &c, **f = &a, g;

void foo () {
  for (;;) {
    assert (a == 0);
    *f = &g;
    *b = 0;
    assert (a == 0); // fails
    *f = 0;
    if (*e) break;
  }
}
int main () {
  *d = 1;
  assert (a == 0);
  foo ();
  return 0;
}
/*** configuration ***/
clang--flto -O2 -c small.c
clang -flto -O2 small.o
```

(e) LLVM trunk (v3.5 rev. 203564) miscompiles this program. The executable violates the second assertion in `foo`. (http://llvm.org/bugs/show_bug.cgi?id=19109)

```
/*** fn1.c ***/
extern void fn2 (void);
extern int a;
void fn1 () {
  a = -1;
  fn2 ();
  a &= 1;
}
/*** fn2.c ***/
extern int a;
void fn2 (void) { a = 0; }
/*** main.c ***/
extern void fn1 (void);
int a;
int main () {
  fn1 ();
  if (a != 0) __builtin_abort ();
  return 0;
}
/*** configuration ***/
gcc-trunk -flto -O1 -c fn1.c
gcc-trunk -flto -Os -c fn2.c
gcc-trunk -flto -O0 -c main.c
gcc-trunk -flto fn1.o fn2.o main.o
```

(c) GCC trunk (v5.0 rev. 219832) miscompiles this program. The compile code aborts instead of terminating normally. (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=64684)

```
/*** foo.c ***/
extern int a, c;
static int bar (int p1, int p2) {
  return p2 + 1;
}
void foo (void) {
  int e = 0, g = 0, b = 1;
  a = (b == 0 ? 0 : b);
  if (bar (a || 0, g) & 1)
    a = e && 0;
}
/*** main.c ***/
extern int foo (void);
int a, c;
int main () {
  for (; c < 2; c++)
    foo (); // called twice
  return 0;
}
/*** configuration ***/
clang -flto -O0 -c foo.c
clang -flto -O1 -c main.c
clang -flto -O1 foo.o main.o
```

(f) LLVM 3.4 and trunk (v3.5 rev. 203791) miscompile this program. The compiled executable hangs. (http://llvm.org/bugs/show_bug.cgi?id=19132)

Figure 4: Example test programs uncovering a diverse array of GCC and LLVM bugs.

detected by Proteus. For LLVM bugs, we only discuss their symptoms, because LLVM developers have not investigated and explained the causes of the failures.

GCC Bug #60319 The function `foo` in Figure 4a relies on the wrap of the variable `c` to a negative number to exit the loop. However, GCC miscompiles this function into an infinite loop, causing the compiled executable to hang.

During the build process, GCC compiled `foo.c` without any optimization (`-O0`). At this optimization level, GCC enables by default the flag `fno-strict-overflow`, which tells the compiler not to treat signed overflow as undefined. Consequently, GCC accepted the wrapping behavior of `c` and dumped the correctly compiled IR to the object file `foo.o`.

```
/*** small.c ***/
int e[2][2];

int main () {
  return e[0][3];
}
$ ccomp -interp small.c
Time 11: program terminated (exit code = 0)
$ clang -fsanitize=undefined -m32 -O0 small.c
small.c:4:12: warning: array index 3 is past the end \
    of the array (which contains 2 elements) \
    [-Warray-bounds]
  return e[0][3];
         ^    ~
small.c:1:1: note: array 'e' declared here
int e[2][2];
^
1 warning generated.
$ a.out
small.c:4:12: runtime error: index 3 out of bounds \
    for type 'int␣[2]'
```

(a) CompCert fails to reject a program that has an invalid multi-dimensional array access.

```
/*** small.c ***/
#include <stdio.h>
void main () {
  printf("foo\n");
}
$ ccomp -interp small.c
$ ccomp small.c
$ a.out
foo
$ clang-trunk -m32 -O0 small.c
small.c:2:1: warning: return type of 'main' is not \
    'int' [-Wmain-return-type]
void main () {
^
small.c:2:1: note: change return type to 'int'
void main () {
^~~~
int
1 warning generated.
$ a.out
foo
```

(b) CompCert fails to reject a program that has invalid main function's signature.

Figure 5: CompCert's bugs found during our reduction process.

At link time, the build script instructs GCC to optimize the program at `-Os`. However, because GCC enables the flag `fstrict-overflow` by default at this optimization level, it treated the wrap in `foo.o`'s IR as undefined behavior. Subsequently, GCC incorrectly replaced the condition `c>=0` with `1`, causing the compiled program to hang.

GCC Bug #60461 The program in Figure 4b triggers a linker error:

```
function fn2: error: undefined reference to 'a'
```

This bug happens because the linker cannot find the reference to the global variable `a[1]`, and as a result, it cannot generate the executable. The bug manifests while GCC's LTO component performs interprocedural analyses. The reference of `a` is not properly updated during the *scalar replacement of aggregates* optimization.

GCC Bug #64684 The program in Figure 4c should terminate normally. It is because at the `if` statement in `main`, the value of `a` should be `0`. However, GCC miscompiles the program using the build configuration in Figure 4c: the compiled program aborts. The value of `a` in this case is `1`.

This happened because during its analyses, GCC gathered incorrect information about the global variable `a`. Specifically, GCC mistakenly concluded that `fn2` did not modify `a`. Consequently, while compiling `fn1`, GCC ignored the call to `fn2`. The value of `a` was `1`, which was the result of `-1 & 1` (this should have been `0 & 1`).

LLVM Bug #19078 The program in Figure 4d is expected to print -1 (the value of the global variable `c`), but prints 0 instead. Variable `c` is assigned `-1` in the second `for` loop of function `fn3`. However, when LTO is enabled, LLVM miscompiles it into the following code, which prints 0:

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

LLVM Bug #19109 In Figure 4e, the compiled code by LLVM fails the second assertion `assert(a == 0)` in function `foo`. This should not happen because `b` points to `a`, and the assignment `*b = 0;` sets `a` to 0 just before the assertion.

The reason is that, when LTO is enabled, LLVM incorrectly moves the statement `*f = &g` down right before the second assertion. Since `f` also points to `a`, this assignment changes the value of `a`. The assertion is thus no longer valid.

LLVM Bug #19132 Figure 4f shows a bug in LLVM's LTO component. The `main` function invokes `foo` twice in the `for` loop and terminates. However, when compiling with LTO, LLVM incorrectly replaces the body of `main` with an infinite loop. The miscompiled code is as follows:

```
int main() {  while(1); }
```

4.4 Discussion

Proteus found a few hundred inconsistencies during our testing period. We managed to reduce a good fraction of them, and reported 37 bugs.[2] However, we are yet to reduce many other interesting ones because the current reduction tools do not work very well for these programs. In general, these programs have very deeply nested constructs, and neither C-Reduce nor Berkeley Delta is able to simplify such constructs. For example, Proteus found a link error bug in GCC 4.7, in which the function calls recurse deeply on their arguments (*i.e.*, the function call argument is the call result of the same function, whose argument is also the call result of that function, and so on). As another example, Proteus found many LLVM bugs, in which array member accesses are deeply nested. We are developing new reduction strategies that exploit programs' syntactic and semantic structures to reduce these programs.

Because Proteus's reduction phase used CompCert's C interpreter heavily, we were able to exercise it thoroughly. During our testing, we found two interesting CompCert bugs, shown in Figure 5. Note that although both Csmith and Orion have spent considerable efforts to stress-test CompCert, they have yet to find a bug in its optimizer.

In the first bug, CompCert mistakenly allows an invalid access to a multi-dimensional array (Figure 5a). The code

[2]Many inconsistencies are duplicate, thus we only report the representatives.

should have undefined behavior as the array access `e[0][3]` is clearly out-of-bound. The warnings from Clang and its UB sanitizer, shown in the same figure, also indicate that the code has undefined behavior. The author of CompCert acknowledged the problem. The reason is that CompCert's C semantics checks that memory accesses are within the bounds of the top-level object being accessed (`e` in this case), but not of its sub-objects being accessed (`e[0]` in this case).

In the second bug, CompCert's C interpreter fails to reject the program that has invalid `main` function's signature (Figure 5b). This program is undefined as the C standard only allows two signatures: `int main(void)` and `int main(int, char **)` (and its equivalent forms). CompCert's author confirmed this bug: the interpreter silently rejects the invalid program without providing any error messages.

5. RELATED WORK

Compiler testing and verification has been a very active research area, primarily due to the critical impact of compilers on every computer system. This section surveys representative, closely related work in this area.

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 the semantic preservation. CompCert [9, 10] is the most notable example of verified compilers. CompCert is mostly written in 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 [26]. 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 [11].

Verified compilers have clear advantages over traditional compilers due to their semantic preserving property. However, there is much work to be done before we have a production-quality verified compiler. CompCert, for example, currently supports fewer language constructs and optimization techniques than GCC and LLVM. As a result, verified compilers are mainly suitable for safety-critical domains which may be more willing to trade language expressiveness and performance for increased correctness guarantees.

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 [20]. Pnueli *et al.* did an early work on translation validation to validate the non-optimizing compilation from SIGNAL programs to C programs [18]. Subsequently, Necula [16] extends it to handle optimizing transformations and validates four optimizations in GCC 2.7. Tristan *et al.* adapt the work on equality saturation [21] to validate intraprocedural optimizations in LLVM [23].

However, translation validation has not yet been successful in practice due to the following reasons. First, current techniques only focus on intraprocedural optimizations, as it is challenging to validate interprocedural optimizations. Second, changes in an optimizer may require appropriate changes in the validator attached to it. Lastly, it is possible for the validator to produce wrong validation results, because itself is unverified and thus may contain bugs.

Compiler Testing In practice, 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 which is maintained along with its development. There are also some commercial test suites available for checking compiler conformance and correctness such as PlumHall [17], SuperTest [1].

The alternative is to use random testing to complement these manually written test suites. Recent work by Nagai *et al.* [14, 15] generates random arithmetic expressions to find bugs in C compilers' arithmetic optimizations. They have found several bugs for 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, 19, 25]. Based on the idea of differential testing [12], Csmith randomly generates C programs and checks for deviant behavior across compilers or compiler versions. Csmith is very successful: it found a few hundred GCC and LLVM bugs over the last several years and contributed significantly to improving the quality of these compilers. Le *et al.* introduce equivalent modulo input, a technique that allows creation of many variants from a single program that are semantically equivalent under some input to stress test compilers [8]. Their tool, Orion, has also found hundreds of bugs in GCC and LLVM. The majority of their bugs are miscompilation bugs.

Proteus is complementary. Its target is the LTO component of a compiler, which has not been considered in previous techniques. Several new challenges arising from testing this component have been discussed in previous sections. Despite its simplicity, Proteus can find LTO bugs at a considerable rate in the most widely-used production compilers.

6. CONCLUSION

We have presented Proteus—a differential testing technique to stress-test link-time optimizers—and the first extensive effort to validate the LTO components of production C compilers. Our evaluation on GCC and LLVM has led to 37 bug reports in 11 months, clearly demonstrating the importance of and Proteus's utility in detecting LTO bugs. Our work complements existing compiler validation techniques such as Csmith and Orion. For future work, we plan to use Proteus to continuously stress-test GCC and LLVM, and extend it to support test programs derived from real-world projects. The key challenge is how to effectively reduce such large test cases, typically consisting of many source files.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for valuable feedback on an earlier draft of this paper. Our special thanks go to the GCC developers for their impressive promptness in analyzing and fixing our reported bugs. Our evaluation also benefited significantly from Berkeley Delta [13], and University of Utah's Csmith [25] and C-Reduce [19] tools.

This research was supported in part by National Science Foundation (NSF) Grants 1117603, 1319187, and 1349528. 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.

[2] B. Anckaert, F. Vandeputte, B. Bus, B. Sutter, and K. Bosschere. Link-time optimization of ia64 binaries. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 284–291. Springer Berlin Heidelberg, 2004.

[3] A. Balestrat. CCG: A random C code generator. 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] B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6(1), Feb. 2007.

[6] GCC Wiki. Finding miscompilations on large testcases. http://gcc.gnu.org/wiki/Analysing_Large_Testcases/.

[7] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–88, 2012.

[8] 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.

[9] 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. ACM Press, 2006.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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, London, UK, UK, 1998. Springer-Verlag.

[19] 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.

[20] H. Samet. *Automatically proving the correctness of translations involving optimized code.* Phd thesis, Stanford University, May 1975.

[21] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, 2009.

[22] The Clang Team. Clang 3.4 documentation: Libtooling. http://clang.llvm.org/docs/LibTooling.html.

[23] 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.

[24] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 17–27. ACM Press, Jan. 2008.

[25] 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.

[26] 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.