Beyond a Joke: Dead Code Elimination Can Delete Live Code

Haoxin Tu*
Singapore Management University
Singapore
haoxintu.2020@smu.edu.sg

Lingxiao Jiang, Debin Gao
Singapore Management University
Singapore
{lxjiang,dbgao}@smu.edu.sg

He Jiang
School of Software, Dalian University of Technology, China
hejiang@dlut.edu.cn

ABSTRACT

Dead Code Elimination (DCE) is a fundamental compiler optimization technique that removes dead code (e.g., unreachable or reachable but whose results are unused) in the program to produce smaller or faster executables. However, since compiler optimizations are typically aggressively performed and there are complex relationships/interplay among a vast number of compiler optimizations (including DCE), it is not known whether DCE is indeed correctly performed and will only delete dead code in practice. In this study, we open a new research problem to investigate: *can DCE happen to erroneously delete live code?* To tackle this problem, we design a new approach named XDEAD, which leverages differential testing, static binary analysis, and dynamic symbolic execution techniques, to detect miscompilation bugs caused by the erroneously deleted live code. Preliminary evaluation shows that XDEAD can identify many divergent portions indicating erroneously deleted live code and finally detect two such miscompilation bugs in LLVM compilers. Our findings call for more attention to the potential issues in existing DCE implementations and more conservative strategies when designing new DCE-related compiler optimizations.

CCS CONCEPTS

• **Software and its engineering** → **Software testing**.

KEYWORDS

Reliability, software testing, program analysis, symbolic execution

ACM Reference Format:
Haoxin Tu, Lingxiao Jiang, Debin Gao, and He Jiang. 2024. Beyond a Joke: Dead Code Elimination Can Delete Live Code. In *New Ideas and Emerging Results (ICSE-NIER'24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639476.3639763

1 INTRODUCTION

Dead Code Elimination (DCE) is a compiler transformation that removes unreachable code or reachable ones whose results are not used[1] [1]. In compiler theory, DCE is treated as a promising technique for producing a smaller or faster executable, saving a greater amount of time or resources when compiling the source code program or running the compiled executables [21, 28]. In compiler implementations, DCE is consistently applied from slight optimizations (e.g. with `-O0`) to aggressive optimizations (e.g. with `-O3`) [1, 28]. Benefiting from such advantages of DCE, a vast number of compilers, such as GCC/LLVM, JavaScript Minifiers, .NET, and OpenJDK, are using DCE by design. Moreover, many studies [17, 18, 26, 28] rely on DCE to detect critical bugs in compilers.

```
1  int idx = 0;
2  int a = 0;
3  void __attribute__((noinline)) marker_2() { ++idx; }
4  static void c() { marker_2(); }
5  void d(int j) { for (;;) ; } // infinite loop
6  void e() { for(int i = 0; i < 100; m++) d(i);}
7  void f() {
8    e(); // live code here is erroneously deleted
9    c();
10 }
11 void g() {if (a == 0x99) f();}
12
13 int main(int argc, char* argv[]) {
14   a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
15   g();
16   printf("%d", idx);
17   return 0;
18 } /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

Figure 1: A *miscompilation* bug[2] detected by XDEAD

In particular, two main lines of research utilize DCE to construct new test programs or build test oracles to detect bugs in GCC/LLVM compilers. First, Orion [17], Athena [18], and Hermes [26] perform dead code insertion or deletion as a mutation strategy to produce a new Equivalence Modulo Inputs (EMI) variant test program. They assume that DCE will only remove dead code so either the insertion or deletion of the dead code will not change the semantics of the seed test program, thus producing EMI variants to detect crash or miscompilation bugs in compilers. Second, Theodoridis *et al.* [28] propose exploiting DCE as a test oracle to identify missed optimization opportunities, thus detecting potential performance bugs in compilers. They also simply assume that DCE will never delete live code and that only dead code is supposed to be deleted.

Research Problem. Due to the complex relationships/interplay among a plurality of aggressive compiler optimizations [28] and the sophisticated implementations of modern compilers (e.g., GCC and LLVM) [4], it is not known whether the assumption that DCE will only delete dead code is always held in practice. In this study, we aim to open an interesting problem to investigate:

Can DCE happen to erroneously delete live code?

Motivating Example. Considering the code example shown in Figure 1 (the code snippets highlighted in grey box are inserted by our approach XDEAD, and we will detail the purpose of each line in Section 2.1), the `main` function simply invokes function `g` (Line 14) which gets function call `f` executed if the condition in *if-branch* in function `g` is satisfied (Line 11). Inside the function `f`, the function `e` is invoked (Line 8) which calls the *infinite loop* in the function `d` in

*Also affiliated with the School of Software, Dalian University of Technology.
[1]For clarification, we refer to such code fragments as *dead* code, while the code is not only reachable but whose results are used is as *live* code (follow the definitions in [26]).

[2]For the sake of simplicity, we revised the code from a real bug [23] reported by XDEAD.

ICSE-NIER'24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0500-7/24/04…$15.00
https://doi.org/10.1145/3639476.3639763

the *for-loop* (Line 6). If a compiler correctly compiles the code, the executable should encounter the *infinite loop* in function d, leading to endless execution. However, the LLVM 11.0.1 downward versions with "-O1" and above miscompile the code and get the wrong result [14] caused by erroneously deleted live code in Line 8. The root cause [9] is that the buggy LLVM compilers erroneously treat calls that may not return as being dead, and then DCE is erroneously performed to remove the function e, causing the miscompilation bug in the LLVM compiler.

Although a rich collection of studies [2, 5, 16–18, 26, 30] is devoted to constructing test programs for compiler testing and they can still possibly generate such a test program in theory, they may fail to detect this bug in practice mainly due to the randomness issue during program generation. First, existing approaches (e.g., Csmith [33] or YARPGen [20]) generally perform fixed value assignments during program generation to construct test programs, which means that users may not be able to change the value of a selected variable (e.g., the variable a in Figure 1). Second, even though users can add extra implementations to support user-provided test inputs, it may still be challenging for them to generate a desirable input (e.g., 0x99) that could execute a specific portion of the executable. Thus, instead of randomly assigning fixed values, a new approach that can automatically manipulate the program inputs so that it can go through a specific portion of the executable is needed.

Our Solution. In this paper, we propose a new approach named XDEAD to tackle the program and detect compiler bugs caused by erroneously deleted live code. More concretely, given a program P, XDEAD inserts a set of markers M into the basic blocks of its source code. When P is compiled under two compilation settings to two binaries, say b1 and b2, a miscompilation bug occurs if (1) a marker m only exits in one binary (say b1) by comparing the existence of the marker m between the assembly code of b1 and b2; (2) there exists an execution path that goes through the marker *m* in b1. Our core insight is that if a divergent and revealing marker exists in only one binary while missing in another, there must exist erroneously deleted live code, either the binary contains the marker (e.g., the one shown in Figure 1) or the binary without the marker. To realize it, XDEAD utilizes differential testing and static binary analysis to find the differentiated markers (i.e., divergent markers between two binaries) and adopts divergent marker-targeted symbolic execution to automatically generate desirable divergence-revealing inputs that could execute the marker.

Taking the example shown in Figure 1 again, given a seed test program generated by Csmith [33] (the code without highlighted), XDEAD instruments the seed program by inserting the highlighted code. Then, by compiling it with two compiler versions and the same compilation options, assuming users are unaware of which compiler is buggy, to two binaries (i.e., "clang-11 -O3 -std=c99" to b1 and "clang-12 -O3 -std=c99" to b2) and comparing their corresponding control flow graphs (CFGs) to identify the divergent markers in the CFGs, XDEAD finds out that the marker_2 only exists in b1 while missing in b2. Next, with the target address of the marker_2, XDEAD utilizes binary symbolic execution (i.e., ANGR [24]) to symbolically execute the instrumented test program by symbolizing the global variable a. Finally, ANGR gives the desirable input (i.e., the value of 0x99) to reveal the execution of the marker_2, which will trigger the miscompilation bug by executing "./b1 0x99".

Preliminary Results. We run XDEAD over two versions of two mature compilers (i.e., GCC and LLVM) to evaluate the effectiveness of XDEAD. The results show that XDEAD is capable of identifying many divergent markers in different running scenarios and successfully detecting two miscompilation bugs caused by live code deleted erroneously in LLVM compilers.

Contributions. This paper makes the following contributions:

- We open a new research problem and propose a new solution XDEAD, which combines differential testing, static binary analysis, and dynamic symbolic execution, to tackle the problem.
- We open source [6] and evaluate the effectiveness of XDEAD, and the preliminary results demonstrate that XDEAD can detect important compiler bugs caused by erroneously deleted live code.

2 APPROACH

Overview. Figure 2 shows the overview design of XDEAD, where several techniques are utilized to construct new test programs that could trigger compiler bugs caused by erroneously deleted live code. XDEAD first instruments the seed program test.c in ①. After using different compilers with the same complication options to compile the instrumented program test-inst.c, XDEAD differentiates the markers from two CFGs of the compiled binaries in ②. Finally, XDEAD leverages dynamic symbolic execution to perform targeted symbolic execution (i.e., the function call address of a divergent marker) over the potentially miscompiled binary (e.g., b1) to produce divergence-revealing test inputs in ③.

2.1 Test Program Instrumentation

The purpose of instrumenting the source code of the seed program is to help identify potential divergent portions (i.e., indicators of live code that DCE might delete) in binaries. Specifically, two substeps, i.e., markers injection and program input modification, are automatically performed in this process.

Markers Injection. Identifying possible erroneously deleted live code in a binary could be challenging. This is because the two binaries, either from the same compiler with two different optimization options or from the same optimization with different compilers, can have huge differences due to different register allocation and basic block re-ordering [13]. To facilitate the identification process, we inject optimization markers into the source code and keep them in the compiled binaries to identify potential divergent markers. The optimization markers can be implemented in many forms such as function calls, compiler builtins, inline assembly, or writes to global variables [28]. In this study, we opt for the markers using a combination of different ways. More concretely, the following function marker_*i* is the function marker injected at the beginning of each block during instrumentation,

```
void __attribute__((noinline)) marker_i() { ++idx; };
```

where *i* refers to the sequential order of the block statements, and each marker_*i* is used uniquely once for each location in the program. The noinline attribute specified for the function is used to avoid the removal of function calls in the binary when higher optimizations are enabled: This could keep the existence of a function call and thus guides the symbolic execution (see more details later in Section 2.3). The body of the marker function includes the

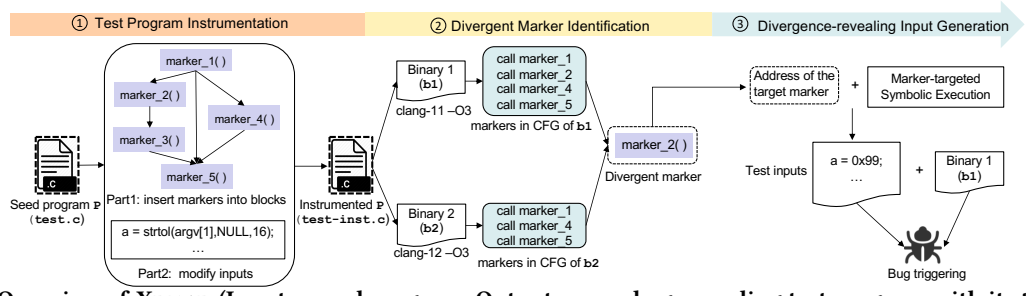

Figure 2: Overview of XDEAD (Input: a seed program; Output: a new bug-revealing test program with its test inputs)

incremented global variable `idx`, which is used to indicate the *live* code and the test oracle without affecting the functionality of the original code: If a binary executes a marker function, but another executable does not, this indicates a miscompilation bug, as the global value `idx` is not the same after execution. This follows the assumption in Csmith [33] and YARPGen [20], where they compute a checksum of the global variables and compare the checksum after binary execution: any differences expose a miscompilation bug. For the inserting locations, variable/function definitions are injected in the global scope and the printing function is injected after invoking the primary function call, e.g., the function `g` inside `main` function in Figure 1. Therefore, the code snippets in Lines 1, 3, 4, and 16 (Line 14 will be injected in the later step) in Figure 1 are injected after the marker injection step.

Program Input Modification. To increase the possibility of executing divergent portions, one possible solution is to let users provide concrete values for their own needs. Unfortunately, it is impractical to try out every possible (e.g., using random fuzzing [25]) value. To enable the automatic generation of divergence-revealing test inputs, we modify the program inputs by inserting the following code snippets inside the `main` function before the executing of the primary function call (e.g., the function call `g` in Figure 1):

```
int n = 1; var = strtol(argv[n++], NULL, 16);
```

Such functions enable users to have the flexibility to control the values of variables (we do not include "`int n = 1;`" in the code example in Figure 1 as there is only one interesting variable to investigate). Therefore, Line 14 is inserted for this purpose.

Since it is challenging to obtain desirable test inputs, XDEAD leverages symbolic execution to automatically generate desirable test inputs by symbolizing the program inputs. It is worth noting that the selection of interesting variables to be controlled by users can affect the trade-off between the number of symbolized variables and the potential path explosion problem during symbolic execution. In this study, we select all integer global variables as interesting variables in XDEAD.

2.2 Divergent Marker Identification

After the instrumentation in ①, XDEAD differentially compiles the instrumented test program under two different versions of compilers with the same optimization options and language standard (e.g., "clang-11 -O3 -std=c99" and "clang-12 -O3 -std=c99" shown in Figure 2) to produce two versions of binaries. Then, XDEAD constructs the CFGs from the two binaries and retrieves the recordings of marker function call names (e.g., `marker_2`) from the CFGs, to identify divergent markers among two comparative binaries. The choice for retrieving markers from CFGs is motivated by the fact that, if the marker is in CFGs, it is possible that there exists an execution path that could execute the marker, assuming the CFGs are correctly yielded for each binary.

2.3 Divergence-revealing Input Generation

This subsection describes the divergent marker-targeted symbolic execution in ③ to automatically generate the divergence-revealing input that could execute a divergent marker identified in ②.

To accomplish this, XDEAD first leverages the address of the function call in the divergent marker (i.e., `marker_2` in Figure 1) as a target and makes the controllable variables as symbolic. Note that after symbolizing every interesting variable, XDEAD maintains the type information of each variable and adds a value range for it to avoid potential undefined behaviors such as integer overflow. Second, XDEAD utilizes the target information to guide a symbolic execution engine (i.e., ANGR [24]) to check if there is an execution path that can execute the target function. Finally, if a solution is found through the directed symbolic execution, XDEAD obtains the concrete values (e.g., `0x99`) solved by a constraint solver.

In this way, divergence-revealing inputs are automatically generated. For example, `test-inst.c` with the concrete input `0x99` triggers the miscompilation bug in Figure 1.

3 PRELIMINARY EVALUATION

3.1 Experiment Setup

Implementation. We have implemented XDEAD [6] using LLVM's `LibTooling` [8] for the instrumentation of the seed programs. We used the tool `bcov` [3] to generate CFG of a binary and Shell code scripts for automatically differentiating markers. We used ANGR [24] as our binary symbolic execution engine and implemented Python and Shell code for the construction of divergence-revealing inputs. During symbolic execution in ANGR, we set the maximum iteration number of 1,000 to loops whose condition involves a symbolic variable, to avoid infinite forking of loop and improve the efficiency.

Benchmarks. We adopt Csmith [33] to generate seed programs, as it is widely used in existing work [16–18, 26–29]. However, the seed programs are not restricted to the program generators, other real-world benchmarks such as SPEC CINT2006 Benchmarks [11] or test suites [7, 10] from GCC or LLVM can also be applicable in XDEAD. For the test subjects, we opt for recent versions of GCC (i.e., GCC-10/11) and LLVM (i.e., LLVM-11/12).

Running Settings. In both GCC and LLVM, the option "-O3" aggressively enables a large number of optimizations on the code.

Table 1: Statistics of divergent markers and test programs

Testing Scenarios	Num.Div.b1	Num.Div.b2	Num.TP	Per.TP	Ave.M
GCC-10/11 (-std=c99)	52,553	0	5,897	58.97%	8.91
GCC-10/11 (-std=c11)	49,431	0	5,758	57.58%	8.59
LLVM-11/12(-std=c99)	187	60	70	0.007%	4.12
LLVM-11/12 (-std=c11)	142	57	68	0.0068%	2.93

* **Num.Div.b1** and **Num.Div.b2** refer to the number of divergent markers in two binaries from different testing scenarios. For example, in the first scenario GCC-11/12 (-std=c99), b1 is compiled with "GCC-11 -O3 -std=c99" while b2 is compiled with "GCC-12 -O3 -std=c99". **Num.TP** represents the number of test programs that contain divergent markers. **Per.TP** refers to the percentage of test programs that exist divergent markers among 10,000 test programs. **Ave.M** counts the average number of markers calculated by (**Num.Div.b1** + **Num.Div.b2**) / **Num.TP**.

Therefore, we separate the experiments into four scenarios (*cf.* Table 1) and compare the results under "-O3" with commonly used C language standards (i.e., "c99" and "c11"). We use CReduce [22] to reduce and produce the minimal version of the bug-revealing test programs. The experiments are run on Ubuntu 18.04 with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz × 12 processors and 64GB RAM.

3.2 Preliminary Experiment Results

The Distribution of Divergent Markers. We generate 10,000 test programs using Csmith [33] and record the number of divergent markers and the number of test programs that contain divergent markers. Table 1 lists the results of divergent markers identified by Xdead. From the column **Ave.M** of the table, we can see that in all scenarios Xdead could detect divergent markers between two binaries. Specifically, GCC and LLVM behave differently in terms of the number of divergent markers and the number of test programs that contain the divergent markers. This is reasonable because LLVM performs more aggressive DCE and a recent study [28] shows LLVM eliminates more dead blocks than GCC among different optimization versions, which indicates that the number of divergent markers that exist in LLVM compilers is smaller than those in GCC compilers. Furthermore, no divergent markers were found with the higher GCC version (i.e., GCC-11) compared with the lower version (i.e., GCC-10) while LLVM compilers behave differently. This result suggests that different versions of GCC compilers perform very similar strategies while doing DCE.

Practical Bug Detection Capability of Xdead. We continuously run Xdead for 90 hours, which follows the same testing duration [16] in detecting compiler bugs in previous work, for each scenario and check how many compiler bugs are detected. As a result, Xdead successfully detects two miscompilation bugs in LLVM compilers. One bug is triggered under LLVM-11/12(-std=c99) with `-O3`, which has been shown as Figure 1 and discussed in Section 1. Another bug [15] is detected under the scenarios of LLVM-11/12 (-std=c11) with `-O3`, where LLVM-11 deletes live code again, causing the miscompilation bug. The root cause of this bug is that LLVM-11 encounters some issues in handling `goto` loops surrounded by complex control/data flows, leading to the deletion of live code.

Comparison with Existing Tools. We also run the same settings (i.e., running 90 hours with the same compilation options) using two notable program generators, i.e., Csmith [33] and YARPGen [20]. As a result, they failed to detect any miscompilation bugs. This is reasonable as (1) modern compilers are already resilient to the test programs generated by them [12, 29] and (2) the low possibility of generating the desirable test program that could execute the divergent portions. In contrast, Xdead could construct new programs that existing program generators are hard to generate, thus contributing to a larger possibility of detecting unique bugs.

Discussion. Xdead is *sound*, which means that Xdead will not produce false positives, under the assumption that the tools, including bcov and Angr, used in the implementation are sound. Xdead is not *complete* as it may miss some opportunities when the deletion of live code is happening but Xdead can not catch it. This is because the function call divergence may be too coarse-grained and not every bug can be detected using this way. There are two threats to validity. One threat lies in the implementation of Xdead, including CFG generation and symbolic execution tools. Another threat comes from the evaluation of Xdead: we only evaluate Xdead over test programs generated by Csmith [33] while other test programs (e.g., from test-suite [7, 10]) are not tested yet.

4 RELATED WORK

Test Program Construction for Compiler Testing. Mainstream test program construction approaches either apply generation-based approaches [20, 33] or adopt mutation-based approaches [2, 17, 18, 26] to construct test programs. All of the above approaches use random strategies to construct test programs, which can be time-consuming and ineffective in detecting certain specific compiler bugs, such as the one caused by erroneously deleted live code. Unlike existing test program construction approaches, which find compiler bugs by luck, the novelty of this study lies in that Xdead designs a new way to construct desired test programs, which could be a new direction for compiler testing.

Finding Semantic Differences in Binaries. Many techniques have been proposed to find the semantic differences between two binary programs when the source code is not available [13, 19, 31, 32]. They typically combine static binary analysis with control-flow graph comparison and combine with symbolic execution for partial instructions in the binary to reduce false positives. Different from existing approaches, Xdead could be a new solution to find the semantic differences between binaries, by first finding the possible semantically different portions and then leveraging dynamic symbolic execution to explore the whole binary instead of partial instructions in the binary for further reducing false alarms.

5 CONCLUSION AND FUTURE PLANS

We open a new research problem and present a new solution Xdead to tackle the problem. The techniques of injection of marker functions, marker differentiation based on differential testing and static binary analysis, and automatic generation of divergence-revealing test inputs are designed in Xdead. The evaluation results show that Xdead can detect important miscompilation bugs caused by erroneously deleted live code. The existence of such bugs calls for more attention to existing DCE implementations and more conservative decisions when designing new DCE-related compiler optimizations.

Future Plans. We are considering the following concrete plans to turn our new idea into a full-length paper in the near future:

- Utilize more fine-grained binary analysis to identify fine-grained divergent portions (e.g., instruction or register) in binaries.
- Apply static program analysis over instrumented programs to select interesting program inputs to be symbolized.
- Conduct more experimental evaluations (e.g., cross-component comparison) to fully assess the effectiveness of Xdead.

REFERENCES

[1] Aho Alfred V, Lam Monica S, Sethi Ravi, Ullman Jeffrey D, et al. 2007. *Compilers-principles, techniques, and tools.* pearson Education.

[2] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating focused random tests using directed swarm testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 70–81.

[3] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2020. Efficient Binary-Level Coverage Analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1153–1164.

[4] Jeremy Bennett. 2024. How Much Does a Compiler Cost? Retrieved 08/01/2024 from https://www.embecosm.com/2018/02/26/how-much-does-a-compiler-cost/

[5] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 305–316.

[6] Developers. 2024. Xdead Implementation. Retrieved 01/08/2024 from https://github.com/haoxintu/Xdead

[7] GCC Developers. 2024. GCC Testsuite. Retrieved 01/08/2024 from https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite

[8] LLVM Developers. 2023. LibTooling. Retrieved 01/08/2024 from https://clang.llvm.org/docs/LibTooling.html

[9] LLVM Developers. 2024. Bug fixing commit before we reported the bug. Retrieved 01/08/2024 from https://reviews.llvm.org/D94106

[10] LLVM Developers. 2024. LLVM Testsuite. Retrieved 01/08/2024 from https://github.com/llvm/llvm-project/tree/main/clang/test

[11] SPEC Developers. 2024. SPEC CINT2006 Benchmarks. Retrieved 01/08/2024 from https://www.spec.org/cpu2006/CINT2006/

[12] Karine Even-Mendoza, Cristian Cadar, and Alastair F Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Software Engineering* 27, 6 (2022), 1–35.

[13] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security (ICICS)*. 238–255.

[14] Godbolt. 2024. Execution results on buggy and non-buggy compilers for bug 1. Retrieved 01/08/2024 from https://godbolt.org/z/z7zxexfr1

[15] Godbolt. 2024. Execution results on buggy and non-buggy compilers for bug 2. Retrieved 01/08/2024 from https://godbolt.org/z/xos1d64xo

[16] He Jiang, Zhide Zhou, Zhilei Ren, Jingxuan Zhang, and Xiaochen Li. 2022. CTOS: Compiler Testing for Optimization Sequences of LLVM. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2339–2358.

[17] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 216–226.

[18] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 386–399.

[19] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. α-diff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 667–678.

[20] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[21] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. 2018. An extensible approach for taming the challenges of JavaScript dead code elimination. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 291–401.

[22] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 335–346.

[23] Bug Report. 2024. LLVM Issue 63121. Retrieved 01/08/2024 from https://github.com/llvm/llvm-project/issues/63121

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. 138–157.

[25] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *The Network and Distributed System Security Symposium (NDSS)*, Vol. 16. 1–16.

[26] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 849–863.

[27] Yixuan Tang, He Jiang, Zhide Zhou, Xiaochen Li, Zhilei Ren, and Weiqiang Kong. 2022. Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4411–4432.

[28] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. 697–709.

[29] Haoxin Tu, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. 2022. Remgen: Remanufacturing a Random Program Generator for Compiler Testing. In *Proceedings of the 33rd International Symposium on Software Reliability Engineering (ISSRE)*. 529–540.

[30] Haoxin Tu, He Jiang, Zhide Zhou, Yixuan Tang, Zhilei Ren, Lei Qiao, and Lingxiao Jiang. 2022. Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing. *IEEE Transactions on Reliability* (2022), 343 – 357.

[31] Sami Ullah and Heekuck Oh. 2021. BinDiff NN: Learning Distributed Representation of Assembly for Robust Binary Diffing against Semantic Differences. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3442–3466.

[32] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 363–376.

[33] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 283–294.