Finding Unstable Code via Compiler-driven Differential Testing

Shaohua Li
shaohua.li@inf.ethz.ch
ETH Zurich
Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

Abstract

Unstable code refers to code that has inconsistent or unstable runtime semantics due to undefined behavior (UB) in the program. Compilers exploit UB by assuming that UB never occurs, which allows them to generate efficient but potentially semantically inconsistent binaries. Practitioners have put great research and engineering effort into designing dynamic tools such as sanitizers for frequently occurring UBs. However, it remains a big challenge how to detect UBs that are beyond the reach of current techniques.

In this paper, we introduce compiler-driven differential testing (CompDiff), a simple yet effective approach for finding unstable code in C/C++ programs. CompDiff relies on the fact that when compiling unstable code, different compiler implementations may produce semantically inconsistent binaries. Our main approach is to examine the outputs of different binaries on the same input. Discrepancies in outputs may signify the existence of unstable code. To detect unstable code in real-world programs, we also integrate CompDiff into AFL++, the most widely-used and actively-maintained general-purpose fuzzer.

Despite its simplicity, CompDiff is effective in practice: on the Juliet benchmark programs, CompDiff uniquely detected 1,293 bugs compared to sanitizers; on 23 popular open-source C/C++ projects, CompDiff-AFL++ uncovered 78 new bugs, 54 of which have been fixed by developers and 36 cannot be detected by sanitizers. Our evaluation also reveals the fact that CompDiff is not designed to replace current UB detectors but to complement them.

CCS Concepts

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

Keywords

compiler, optimization, unstable code, undefined behavior, sanitizer

ACM Reference Format:
Shaohua Li and Zhendong Su. 2018. Finding Unstable Code via Compiler-driven Differential Testing. In *Proceedings of 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

ASPLOS'23, March 25–29, 2023, Vancouver, Canada

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

```
1  /* dump a chunk of buffer*/
2  int dump_data (int offset, int len) {
3      char *data = /* buffer head */;
4      int size = /* size of buffer*/;
5      if (offset + len > size ||
6          offset < 0 || len < 0) {
7          return -1;
8      }
9      if (offset + len < offset) {
10         return -1;
11     }
12     /* dump from data+offset
13        to data+offset+len */
14     dump(data+offset, len);
15     return 0;
16 }
```

Listing 1: The second if guard in line 9 got optimized away by clang because it would only be evaluated to true when a signed integer overflow happened.

1 Introduction

Some programming languages such as C/C++ designate a set of code constructs as having undefined behavior (UB) to simplify the compiler implementation. For example, the C17 standard [9] lists 211 circumstances for which it invokes undefined behavior. According to the standard, permissible undefined behavior results in "ignoring the situation completely with unpredictable results". Compilers can assume that undefined behavior will never occur in the program which allows many optimization opportunities. A consequence of such an assumption is for code that contains undefined behavior, different compilers implementations may generate semantically different binaries. Previous study [47, 48] has shown that undefined behaviors may cause optimization-unstable code, code that could be unexpectedly discarded by compiler optimizations. In this paper, we refer to code that has inconsistent semantics across compiler implementations due to undefined behavior as *unstable code*.

Example of unstable code. Listing 1 shows an example of unstable code, where the `if` guard in line 9 tries to handle possible integer overflow. But `offset+len` can never be less than `offset` unless undefined behavior, *i.e.*, signed integer overflow, occurs. According to the standard, compilers can do arbitrary optimizations with the assumption that undefined behavior never occurs. The consequence is that an optimizing compiler (*e.g.*, `clang-O2`) optimizes away the second `if` branch (lines 9-11) while a less optimizing compiler (*e.g.*, `clang-O0`) keeps it. That means the compiled binaries have different semantics. For example, if we call

`dump_data(INT_MAX-100, 101)`, the optimized binary will dump the buffer starting from `data+INT_MAX-100` and return 0, while the unoptimized binary will dump nothing and return -1. On one hand, this issue leads to a security hole in the optimized binary as a large range of illegal memory data could be dumped. On the other hand, it breaks the functional correctness of the code as its binaries compiled by different compilers may produce divergent outputs.

Our approach. From the above example, we can observe that unstable code leads to different execution semantics across compilations once undefined behavior has been triggered. This conforms to the standard that compilers can in principle do arbitrary optimizations on such erroneous code. In this paper, we make use of this fact and propose a simple, straightforward, yet effective approach for finding unstable code. Our approach involves three steps. First, compile the target program with different compiler implementations to get a set of binaries. We consider different compilers and optimizations as different compiler implementations. For example, `gcc-O0`, `gcc-O2`, and `clang-O2` are three different compiler implementations. Second, run these binaries on the same set of test inputs and collect their outputs. Finally, compare outputs produced by different binaries on the same input and report discrepancies. For a program with deterministic output, *i.e.*, repeated executions of the program on the same input always yield the same output, output discrepancy over the same input implies the presence of unstable code. We call our approach *compiler-driven differential testing* (CompDiff). For the example in Listing 1, CompDiff can successfully detect the issue because of the divergent outputs on the same input. For non-deterministic or multi-threaded programs, they may have non-deterministic internal execution traces. But as long as they have deterministic output, they can be analyzed with CompDiff. Note that, CompDiff assumes compiler implementations are bug-free. Compiler bugs, which are rare for mature compilers [30], may indeed cause divergent outputs and thus be caught by CompDiff. As will be shown in our evaluation, compiler bugs are rarely encountered in real-world software. Once it happens, developers are willing to diagnose and report it.

CompDiff's design also covers bugs that are not due to undefined behavior. As long as a bug results in output discrepancy across compiler implementations, it will be detected by CompDiff. Our evaluation will show that CompDiff indeed finds real bugs that are not undefined behavior. We consider this as an additional benefit of CompDiff.

Existing approaches. Industry and academics have proposed a plethora of static and dynamic tools for findings frequently occurring undefined behaviors such as buffer overflow, integer overflow, division by zero, *etc.* Static tools [12, 31, 48] analyze source code without executing it to detect certain types of errors. They typically build upon heuristics and suffer from both false positives and false negatives. Dynamic tools [6, 33], on the contrary, perform analysis of concrete executions and normally incur no false positives. Sanitizers such as AddressSanitizer (ASan) [42], UndefinedBehaviorSanitizer (UBSan) [15], and MemorySanitizer (MSan) [43] supported by compilers are widely used in practice. They insert checks into necessary program locations to detect undefined behaviors at run-time. For the example in Listing 1, UBSan would insert a check around each `offset+len` to verify whether or not its value exceeds `INT_MAX`. Thanks to their strong bug detection ability, sanitizers have become de facto state-of-the-art for discovering UBs, especially in fuzzing.

Table 1: Scopes of sanitizers and CompDiff.

Approach	Scope
ASan	Memory errors (*e.g.* buffer-overflow)
UBSan	Miscellaneous UBs (*e.g.* division-by-zero)
MSan	Use of uninitialized memories.
CompDiff	A diverse range of UBs.

Sanitizers cover many frequently occurring UBs. Each sanitizer is designed for certain classes of UBs. Table 1 lists scopes of three widely used sanitizers and our CompDiff. On one hand, CompDiff covers a broader range of UB even than the combination of these sanitizers. The reason is that our design stems from unstable code, a common consequence of UB. Sanitizers contrarily design customized checks for each kind of UB. Since not all UBs have applicable checks, some UBs cannot be detected by sanitizers. We will show in Section 2 three examples where sanitizers fail to detect them. On the other hand, sanitizers have high bug coverage for UBs that they specialize in. CompDiff, however, may miss many of them. We argue that CompDiff is not to replace sanitizers but to complement them by covering extra UBs.

To improve CompDiff's practicality and detect unstable code in real-world software, we integrate CompDiff into AFL++ [17], the most widely-used general-purpose fuzzer. Our evaluation shows that, on the Juliet benchmark tests, CompDiff uniquely identified 1,293 bugs that sanitizers failed to. On 23 popular open-source C/C++ projects, CompDiff-AFL++ discovered 78 new bugs, 66 of which were confirmed, and 54 were fixed by the developers. Of these new bugs, 36 were not detected by sanitizers. Our evaluation also confirms the strong detection ability of sanitizers on certain classes of bugs. Our CompDiff-AFL++ tool is available at https://github.com/shao-hua-li/compdiff. In summary, our contributions are as follows:

- We propose CompDiff, a simple, straightforward, yet effective approach for finding unstable code.
- We integrate CompDiff into the popular fuzzer AFL++.
- We evaluate CompDiff on both benchmark and real-world programs. The results show that CompDiff significantly complements sanitizers.

2 Illustrative Examples

This section illustrates three real-world examples, which we use to demonstrate how CompDiff enables the discovery of unstable code and why sanitizers fail to detect them.

Example 1: Invalid pointer comparison. Listing 2 shows a piece of unstable code found by CompDiff in Binutils. The pointer comparison in line 5 is UB as `look_for` and `saved_start` are pointing

```
1  int display_debug_frames (...) {
2      char *saved_start=/*point to object A*/;
3      char *look_for =/*point to object B*/;
4      ...
5      if (look_for <= saved_start) {...}
6      else {...}
7  }
```

Listing 2: A pointer comparison UB found by CompDiff in binutils/dwarf.c. The if check in line 5 is evaluated differently (either true or false) across compiler implementations (https://sourceware.org/bugzilla/show_bug.cgi?id=27836).

```
1  char * GET_LINKADDR_STRING(int8_t *p) {
2      static char buffer[BUF_SIZE];
3      /* write *p to buffer */
4      ...
5      return buffer;
6  }
7  void arp_print(...) {
8      ...
9      ND_PRINT("who-is %s tell %s",
10         GET_LINKADDR_STRING(p1),
11         GET_LINKADDR_STRING(p2));
12     ...
13 }
```

Listing 3: An example of unstable code simplified from tcpdump/print-arp.c. The two calls to `GET_LINKADDR_STRING` are arguments to the function `ND_PRINT` (https://github.com/the-tcpdump-group/tcpdump/issues/919).

to different objects. The standard [9] (§6.5.8) describes that such undefined behavior happens when *"Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators."* None of the sanitizers can detect this kind of UB because it remains unknown how to design a proper check for it. CompDiff can easily detect this issue because the `if` guard will be evaluated differently across compiler implementations and thus divergent outputs will be observed.

Example 2: Evaluation order of subexpressions with conflict side effects. Listing 3 shows an example of unstable code in the well-known network packet analyzer Tcpdump [21]. The function `ND_PRINT` (line 9) dumps formatted network information. In this code snippet, developers try to dump fields `p1` (line 10) and `p2` (line 11) by calling the function `GET_LINKADDR_STRING` twice. These two calls are also arguments to the function `ND_PRINT`. According to the standard, compilers can evaluate function arguments in any order. However, *"If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings."* [9] (§6.5.0). This example matches the definition of this undefined behavior and becomes unstable. First, the function `GET_LINKADDR_STRING`

```
1  std::ostream& CanonMakerNote::print0x000c(
2      std::ostream& os, const Value& value) {
3
4      std::istringstream is(value.toString());
5      uint32_t l;
6      is >> l;
7      return os << std::hex
8                << ((l & 0xffff0000) >> 16);
9  }
```

Listing 4: A use of uninitialized variable in `exiv2` where `l` stays uninitialized even after line 6 when `is` is an empty string (https://github.com/Exiv2/exiv2/issues/1717).

uses a static char array `buffer` to store the resulting string. The memory region pointed to by `buffer` will be shared across function calls. Since there are two calls to this function, the result of the first call, which is stored in `buffer`, will be overwritten by the second call. Thus in the dumped string, the two fields `who-is` and `tell` will always be the same. Second, since the language specification poses no restriction on the evaluation order of function arguments, different compilers may evaluate these two `GET_LINKADDR_STRING` calls in a different order. If we compile Tcpdump with `gcc` and `clang` separately, the obtained two binaries will evaluate the arguments of `ND_PRINT` in reverse order, leading to inconsistent dump strings. Specifically, `clang` evaluates the arguments from the first to the last, *i.e.*, `p2` will be dumped to both `who-is` and `tell`; while `gcc` evaluates the arguments from the last to the first, *i.e.*, `p1` will be dumped to both attributes.

To discover this issue, we need to have at least two compiled tcpdumps from `gcc` and `clang`, respectively, and tests that can reach the unstable program location. We identified this issue with our CompDiff-AFL++ tool, where CompDiff was configured to compile a target with multiple compiler implementations from both `gcc` and `clang`. The back-end AFL++ generated tests that reached the target location.

All sanitizers currently do not support the detection of this type of issue. Extending sanitizers to support such detection requires the design of a new checker that could examine whether or not multiple subexpressions have side effects on conflict memory regions. It remains unknown how to implement such a checker.

Example 3: Uninitialized memory usage. Listing 4 shows a piece of unstable code due to the use of an uninitialized variable. The developers might think that although the variable `l` is uninitialized, its initial random value should be overwritten in line 6 with the content in `is`. However, in a corner case where `is` is an empty string the variable `l` will remain unchanged. The uninitialized value will then be used for the rest of the execution, in this case printing out to ostream. As the value of the uninitialized variable is indeterminate [9] (§6.7.9) and depends on run-time memory layout, different compilers and optimizations may allocate different values to it.

MemorySanitizer supports the detection of uninitialized memory usage, where uninitialized values have to be used to determine code branches, *e.g.*, an if guard relies on an uninitialized value. To avoid

false positives, it does not support cases such as the one shown in the example.

CompDiff-AFL++ can detect this issue because 1) the back-end AFL++ can generate tests that cause the variable `is` to be empty and thus `l` to be different across binaries; 2) CompDiff captures divergent outputs.

Limitations. Although CompDiff can cover extra bugs than sanitizers, it cannot detect as many issues as sanitizers for certain kinds of UBs. The reasons are twofold. First, some UBs do not lead to unstable code. Although compilers in theory could generate arbitrary binaries for code having UB, they in practice may not exploit the UB or generate semantically equivalent binaries. Second, Even if a program contains unstable code, the erroneous behavior may not propagate to the final outputs.

3 Finding Unstable Code

This section details our approach to finding unstable code. Section 3.1 formalizes unstable code and presents our proposed compiler-driven differential testing for the detection of unstable code. Section 3.2 demonstrates the implementation details of integrating CompDiff into AFL++.

3.1 Compiler-driven Differential Testing

Both C [9] and C++ [10] standards pose no requirements on how a compiler behaves on code fragments that invoke undefined behavior. A compiler implementation *w.r.t.* the programming language specification can thus in principle do arbitrary transformations and optimizations on such erroneous code fragments. Intuitively, for a program with deterministic output, *i.e.*, repeated executions of the program on the same input always yield the same output, if the binaries compiled by any two legal and correct compiler implementations produce different outputs on some input, there has erroneous code fragment in the program. We call such erroneous code fragments leading to divergent outputs across compilations *unstable code*.

To formalize unstable code, let $\mathcal{C}_a$ and $\mathcal{C}_b$ be *any* two legal compiler implementations. For a program $\mathcal{P}$ with deterministic output, $\mathcal{C}_a$ and $\mathcal{C}_b$ compile program $\mathcal{P}$ to binaries $\mathcal{B}_a$ and $\mathcal{B}_b$, respectively. With these notations, we introduce unstable code from a dynamic testing perspective as follows:

Definition 1 (Unstable Code). *$\mathcal{P}$ has unstable code if there exists an input such that executing $\mathcal{B}_a$ and $\mathcal{B}_b$ on the input produces different outputs.*

This definition shows that enumerating inputs on binaries compiled by all possible compiler implementations may help us to find unstable code. Based on this intuition, we propose compiler-driven differential testing (CompDiff) for detecting unstable code in real-world programs. For a program $\mathcal{P}$, the general workflow of CompDiff is as follows:

1) Find a set of legal compiler implementations $\mathcal{C}_i, i \in [1, 2, \cdots, k]$.
2) Compile $\mathcal{P}$ with each $\mathcal{C}_i$ to obtain binaries $\mathcal{B}_i$, $i \in [1, 2, \cdots, k]$.
3) Find an input set $\mathcal{I}$ for $\mathcal{P}$.
4) For each input $t \in \mathcal{I}$, run each $\mathcal{B}_i$ on it and obtain outputs o_i. If there exists $i, j \in [1, \cdots, k]$ and $i \neq j$ such that $o_i \neq o_j$, report t as bug-triggering input.

The above workflow shows three key factors in CompDiff, *i.e.*, compiler implementations, input set, and output examination. In the following, we will discuss each these factors in detail.

Compiler implementations. Ideally, we could find only two compiler implementations that always behave differently on unstable code. However, such ideal compilers do not exist. In practice, we should utilize a set of concrete compiler implementations. For a given target, there are typically plenty of metamorphic compiler implementations. For instance, `clang` has hundreds of optimization passes available for developers to choose from, which results in a combinatorial explosion of possible compiler implementations. Popular compilers, such as gcc [13] and clang [14], feature well-engineered optimization levels (*i.e.*, -O0, -O1, -O2, -O3, and -Os) for users to choose from. For open-source C/C++ projects, developers often use different compilers and optimization levels when developing, testing, and releasing code. Users may also freely choose their own preferences when compiling code. These compilers and optimization levels offer us a good collection of compiler implementations for CompDiff. The reason is twofold. First, each of these compiler implementations uses a different set of transformations and optimizations, which increase the possibility of exploiting unstable code across them. Second, all of these compiler implementations are widely used in real-world cases by either developers or users. As long as CompDiff discovers a discrepancy, the issue is likely affecting real users and thus persuading.

Input set. Since CompDiff is a dynamic testing approach, concrete execution is required to detect bugs. One can use a test suite provided by developers or generated with existing test generation tools. Fuzzing is one of the most popular automated testing techniques for discovering software defects due to its simplicity and effectiveness. By generating a tremendous amount of mutated inputs with feedback guidance, fuzzer is powerful in finding interesting program paths. In this paper, we use a fuzzer as the test generation tool to power CompDiff. Section 3.2 will demonstrate how we integrate CompDiff into AFL++ for finding unstable code in real-world software. Note that, CompDiff has no particular requirements on the underlying fuzzer and is generally applicable to other fuzzers.

Output examination. Conceptually, our definition of unstable code shows that finding them in a program requires verifying the semantic equivalence of its compiled binaries. Full verification is infeasible due to its complexity and undecidability [11, 38]. To practically reason about semantic equivalence, our design only concerns final outputs of binaries on a concrete input. There are two reasons behind our choice. First, it is easy and cost-efficient to obtain input/output pairs in practice. Unlike other techniques like sanitizers, such design requires no modification or instrumentation to the target program. Second, it is convincing as proof of the presence of unstable code. Since all compiler implementations are used in the real-world, any discrepancy in the final outputs indicates that the functional correctness of the program has been affected in at least one compiler implementation. Unfortunately, such a design will miss bugs when the erroneous state does not propagate to the final

Algorithm 1: CompDiff-AFL++

```
Input: Seed pool S.
1  while ¬Abort() do
2      s ← SelectSeed(S)
3      s' ← Mutate(s)
4      _ ← Execution(s', B_fuzz)
5      if s' causes failure on B_fuzz then
6          save s' to disk
7      if s' increases coverage on B_fuzz then
8          add s' to S
       /* CompDiff: Run s' on each B_i and examine
          output consistency.                    */
9      for i = 1 to k do
10         o_i ← Execution(s', B_i)
11     if ! (o_1 = o_2 = ··· = o_k) then
12         save s' to disk
```

output. One may argue that instead of only examining final outputs, we can check programs' intermediate results, *e.g.*, return values from all functions, to extend CompDiff's capability. However, it 1) may incur many false positives due to compilers' optimizations such as inlining, 2) is less persuading than final outputs in motivating developers to fix the error, and 3) requires significant and non-trivial implementation efforts in obtaining intermediate results at run-time. It is, however, worth further exploration in the future.

3.2 CompDiff-AFL++

We here demonstrate integration details of CompDiff-AFL++. Since CompDiff is orthogonal to fuzzers, we hope our demonstration can guide the future adoption of CompDiff in other fuzzers.

In CompDiff-AFL++, there are multiple binaries compiled from the target program $\mathcal{P}$. The first one is $\mathcal{B}_{fuzz}$, which is compiled by the fuzzer-configured compiler $\mathcal{C}_{fuzz}$. The compiler $\mathcal{C}_{fuzz}$ injects instrumentation code into $\mathcal{P}$ such that the fuzzer can collect coverage feedback from the resulting binary $\mathcal{B}_{fuzz}$. If sanitizers are enabled, $\mathcal{C}_{fuzz}$ also insert sanitizer checks to $\mathcal{B}_{fuzz}$. Note that, $\mathcal{B}_{fuzz}$ is compiled the same as in normal AFL++. The remaining binaries are all for CompDiff. As has been discussed in Section 3.1, we recommend the use of compilers `gcc` and `clang` with different optimization levels to form the set of compiler implementations $\mathcal{C}_i, i \in [1, \cdots, k]$. Each $\mathcal{C}_i$ compiles $\mathcal{P}$ to binary $\mathcal{B}_i$. We inject some lightweight instrumentation code into each $\mathcal{B}_i$ for efficient execution. We will discuss the instrumentation on $\mathcal{B}_i$ after introducing the algorithmic sketch of CompDiff-AFL++.

Algorithm 1 shows the high-level workflow of CompDiff-AFL++. The unhighlighted part describes AFL++'s main process:

1) (line 2) Select a seed input from the seed pool.
2) (line 3) Mutate the input with one of the available mutation operators.
3) (line 4) Execute the program on the mutated input and collect the feedback such as code coverage from the execution.
4) (line 5-8) If the new input causes a crash, save it to disk; if it increases coverage, add it to the seed pool; otherwise, drop it. Then go to step 1).

The highlighted part in Algorithm 1 shows where and how CompDiff works in AFL++. It first runs the new input on each $\mathcal{B}_i$ (lines 9-10), then cross-checks their outputs and saves the input if a divergence is found (lines 11-12). The workflow illustrates that CompDiff does not interfere with AFL++'s normal procedures but only augments it with the extra test oracle provided by CompDiff. Thus, other fuzzing enhancements to AFL++ are compatible with CompDiff-AFL++. For example, sanitizers work by instrumenting $\mathcal{B}_{fuzz}$ to expose more bugs, and thus they can be normally used in CompDiff-AFL++. Next, we will discuss key implementation details in CompDiff-AFL++.

Instrumentation on $\mathcal{B}_i$. Each compiler implementation $\mathcal{C}_i$ primarily specifies the used compiler and optimization level. For instance, in our default setting, $\mathcal{C}_1$ uses `CC=clang CXX=clang++ CFLAGS="-O0" CXXFLAGS="-O0"` and $\mathcal{C}_2$ uses `CC=clang CXX=clang++ CFLAGS="-O1" CXXFLAGS="-O1"`. To reduce the burden of launching target binaries $\mathcal{B}_i$, we also inject *forkserver* [26] instrumentations into them. Forkserver has been widely used in many fuzzers for the same purpose. At a high level, when the fuzzer needs to execute an input on $\mathcal{B}_i$, it first writes the input to shared memory, then notifies the forkserver in $\mathcal{B}_i$. After the forkserver receives the notification, it forks itself, runs the input on the forked child process, and informs the fuzzer when it is done. Interested readers can find further details in [26].

Output examination. AFL++, by default, drops all outputs emitted from the binary. To obtain output from each $\mathcal{B}_i$, we redirect output, as well as error, from each $\mathcal{B}_i$ to a file using `dup2()`. We then compare the checksum values of these files to find discrepancies. We reuse the MurmurHash3 [1] hash function supported by AFL++ for the checksum.

Bug-triggering inputs. We save all inputs that triggered output discrepancies into a separate directory "diffs/" for future diagnosis. Similar to crash-triggering inputs in normal fuzzing, there are many inputs that trigger the same bug. It is non-trivial to automatically identify unique discrepancies, especially in the context of differential testing. We currently rely on manual analysis of reported discrepancies to triage bug reports. Automated triage, as well as debugging, are discussed further in Section 5.

4 Evaluation

In our evaluation, we use `gcc 11.1.0` and `clang 13.0.1`, the latest stable versions at the beginning of our evaluation, as the back-end compilers in CompDiff and CompDiff-AFL++. These two compilers are selected because of their widespread adoption in open-source C/C++ projects. To thoroughly understand the capability of different optimization levels, we utilize all frequently used ones, *i.e.*, `-O0`, `-O1`, `-O2`, `-O3`, and `-Os` in both compilers. The combination

gives us 10 different compiler implementations. Our CompDiff-AFL++ is implemented atop AFL++ version 3.15a. All experiments are done in a server equipped with an AMD Ryzen Threadripper 3990X 64-Core 2.9GHz CPU and 256 GB RAM, and running Ubuntu 20.04.3 LTS.

We evaluate the effectiveness and practicality of CompDiff and CompDiff-AFL++. To understand the capability of CompDiff in finding unstable code, we evaluate it on a collection of benchmark programs from the Juliet test suite. These programs contain a diverse range of undefined behaviors and the ground truth is available. We then use 23 well-maintained open-source C/C++ projects to evaluate the bug-finding ability of CompDiff-AFL++ in real-world software. There are a plethora of static and dynamic tools for detecting common UBs. We evaluate three popular and widely-used C/C++ static analyzers, *i.e.*, Coverity [40], Cppcheck [12], and Infer [31], on the Juliet test suite. These tools implement state-of-the-art techniques for detecting various program issues and were used in previous study [25, 28]. Since CompDiff is a dynamic analysis tool, we also compare it with sanitizers, the state-of-the-art dynamic analysis tools, on both the Juliet test suite and real-world software. Other dynamic tools such as Valgrind [33] and Dr.Memory [6] do not have better detection ability than sanitizers when source code is available [16]. Thus we compare CompDiff with sanitizers only. Specifically, we compare our tool with three widely-used sanitizers, *i.e.*, AddressSanitizer (ASan), UndefinedBehaviorSanitizer (UBSan), and MemorySanitizer (MSan).

4.1 Effectiveness of CompDiff in benchmark programs

The Juliet test suite C/C++ [35] released by NIST contains a collection of test cases, which are classified based on MITRE's Common Weakness Enumeration (CWE) classification system. This test suite has been widely used to evaluate both static analysis tools [28, 31] and dynamic testing approaches [16]. Each test case can run as an independent program and contains two variants: a *bad* variant that contains a flaw and a *good* that does not. All bad variants can be used to evaluate the bug detection rate of a tool while good variants can be used to evaluate the false positive rate of a tool.

Not all CWEs are due to undefined behaviors, many of which are insecure or unsafe operations such as hard-coded password (CWE-256). In order to evaluate the effectiveness of CompDiff in finding unstable code, we manually analyzed each CWE category and selected CWEs that represent bugs due to undefined behavior. Since we only deal with programs that have deterministic output, we excluded tests that deliberately change outputs per run. We also removed tests that timed out after 5 seconds [16]. This extraction gave us 18,142 tests spanning 20 CWEs. Table 2 shows an overview of the selected CWEs.

We analyze each test with Coverity, Cppcheck, Infer, ASan, UBSan, and MSan, in addition to our tool, CompDiff. For clear presentation, we merge tests with similar causes. We use *bad* (buggy) variant of each test. We evaluate the bug detection rate and the false positive rate of each tool. Bug detection rate (or recall) is the percentage of all real bugs detected by the tool. False positive rate is the percentage of incorrect reports (or false alarms) out of all reports produced by a tool. Table 3 shows the bug detection rates (%) and false positive rates (%) of each tool. Since all sanitizers and CompDiff do not have false positive reports on the Juliet test suite, we omit their false positive columns for better presentation. The last column shows the number of bugs that can be uniquely discovered by CompDiff compared to sanitizers. We next discuss five findings on the results.

Table 2: Overview of selected CWEs.

CWE-ID	Description	#Tests
CWE-121	Stack Based Buffer Overflow	2,951
CWE-122	Heap Based Buffer Overflow	3,575
CWE-124	Buffer Underwrite	1,024
CWE-126	Buffer Overread	721
CWE-127	Buffer Underread	1,022
CWE-415	Double Free	820
CWE-416	Use After Free	394
CWE-475	Undefined Behavior for Input to API	18
CWE-588	Access Child of Non Struct. Pointer	80
CWE-590	Free Memory Not on Heap	2,280
CWE-685	Function Call With Incorrect #Args.	18
CWE-758	Undefined Behavior	523
CWE-190	Integer Overflow	1,564
CWE-191	Integer Underflow	1,169
CWE-369	Divide by Zero	437
CWE-476	NULL Pointer Dereference	306
CWE-680	Integer Overflow to Buffer Overflow	196
CWE-457	Use of Uninitialized Variable	928
CWE-665	Improper Initialization	98
CWE-469	Use of Pointer Sub. to Determine Size	18
Total		**18,142**

➤ Finding 1: Static tools have non-negligible false positive rates and relatively lower bug detection rates compared to CompDiff.

All static tools show non-negligible false positive rates. Coverity, Cppcheck, and Infer have 0% ~ 46%, 0% ~ 35%, and 0% ~ 67% false positive rates, respectively. On the contrary, CompDiff has zero false positive rate. Cppcheck can detect the same number of bugs as sanitizers and CompDiff on CWE-475 and CWE-685. Infer detects the most number of bugs on CWE-190~680 while still has 25% false positive rate. For the rest of bug categories, CompDiff has significantly higher bug detection rates than Cppcheck and Infer. For example, on CWE-588, CompDiff detect 99% of bugs while Cppcheck and Infer only detect 0% and 2% of bugs, respectively. On most of the CWEs, CompDiff detects more bugs than Coverity. For bugs from "UB", "Integer error", and "Divide by zero", Coverity has higher bug detection rates. However, Coverity has 4% to 23% false positive rates on these bugs.

Table 3: Bug detection rates (%) and false positive rates (%) on the Juliet tests. The highest bug detection rates are highlighted in green. False positive rates of static tools are shown in columns FP. Since all sanitizers and CompDiff have no false positive on the Juliet tests, we omit their FP values. The last column (#Unique) lists the number of bugs that are uniquely detected by CompDiff compared to sanitizers.

CWE-IDs	Description	Static Tools			Sanitizers				CompDiff	
		Coverity FP	Cppcheck FP	Infer FP	ASan	UBSan	MSan	**Total**	Detected	#Unique
121~127 127, 415, 416, 590	Memory error	39% 46%	13% 6%	37% 21%	94%	✗	✗	94%	63%	137
475	UB for input to API	100% 0%	100% 0%	0% 0%	100%	✗	✗	100%	100%	0
588	Bad struct. pointer	32% 21%	0% 4%	2% 2%	49%	✗	✗	49%	99%	40
685	Bad function call	100% 0%	100% 0%	0% 0%	100%	✗	✗	100%	100%	0
758	UB	100% 4%	0% 35%	0% 0%	36%	✗	✗	36%	92%	293
190, 191, 680	Integer error	22% 21%	0% 2%	49% 25%	✗	33%	✗	33%	11%	31
369	Divide by zero	54% 23%	8% 3%	3% 7%	✗	54%	✗	54%	29%	5
476	Null pointer deref.	69% 9%	29% 3%	77% 69%	✗	92%	✗	92%	93%	3
457, 665	Uinitialized memory	44% 56%	24% 15%	9% 13%	✗	✗	7%	7%	92%	882
469	UB of pointer Sub.	0% 0%	0% 0%	0% 0%	✗	✗	✗	0%	100%	18

Overall, CompDiff shows stronger bug detection ability than static analysis tools. In practice, static and dynamic tools have complementary strengths and should be used together to maximize the bug detection rate.

➤ **Finding 2: CompDiff complements sanitizers by discovering many extra bugs.** CompDiff complements sanitizers' bug detection ability from three perspectives. First, for some kinds of bugs, CompDiff has a higher detection rate than the combined sanitizers. For example, on CWE-588 and CWE-758, CompDiff detects 99% and 93% of bugs while sanitizers in total only detect 49% and 36% of them. On CWE-457 and 665, although MSan in sanitizers specializes in detecting uninitialized variable uses, it only covers 7% of bugs while CompDiff identifies 92% of them. Second, even CompDiff fails to detect as many bugs as sanitizers in some classes of tests, it still discovers unique bugs that are missed by sanitizers. For example, for the memory errors shown in the first row, sanitizers detect 94% of bugs while CompDiff only achieves 63% detection rate. But there are 137 bugs that can only be covered by CompDiff. Third, CompDiff covers UBs that are not yet supported by sanitizers. None of the sanitizers discovers any bug on CWE-469, CompDiff, however, exposes them all.

➤ **Finding 3: CompDiff has the highest bug coverage compared to each individual sanitizer.** Compared to the results shown in columns "ASan", "UBSan", and "MSan", the second-to-last column shows that CompDiff can detect a diverse range of unstable code. Each sanitizer, on the contrary, only specializes in certain kinds of bugs. For example, MSan is only designed for uses of uninitialized variables and thus it cannot detect all other bugs. CompDiff's design, however, stems from the general consequence of UBs and thus has a higher overall bug coverage in principle.

➤ **Finding 4: CompDiff misses certain kinds of bugs.** Since sanitizers are designed for specific classes of bugs, they work better than CompDiff on them. For instance, UBSan reaches higher coverage than CompDiff in bugs related to integer errors and divide by zero. Because CompDiff only concerns a program' final output, an erroneous state resulting from these errors may not propagate to the output. This weakness is expected from the design choice of CompDiff. As we have been emphasizing, CompDiff is not to replace sanitizers but to complement them to cover more bugs.

➤ **Finding 5: CompDiff has no false positive.** To measure whether or not CompDiff report any false positive, we also run CompDiff on *good* variant of each test. The result shows that CompDiff has no false positive, which is also the reason why we do not use a separate table for the result. For programs with deterministic output and correct compiler implementations, it is expected that outputs from the same input are identical across different compilations.

Summary. The above findings suggest that CompDiff is effective in detecting bugs related to UBs and has the highest overall bug coverage. Our results also confirm the strong discovery rate of each sanitizer on certain kinds of bugs. Compared to the combined sanitizers, on some of the UBs, CompDiff cannot detect as many bugs as them but still discovers additional unique bugs. We position CompDiff as a complementary tool to sanitizers. One should use both techniques to maximize the bug detection rate in practice.

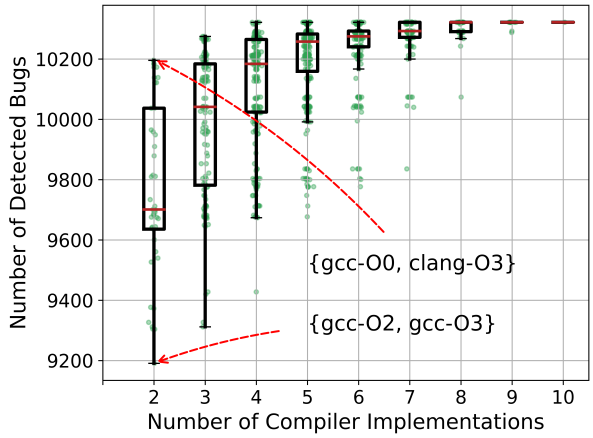

Figure 1: Number of bugs could be detected by each subset of compiler implementations.

4.2 Impact of reducing #compiler implementations

By default, there are ten compiler implementations used in CompDiff, *i.e.*, `gcc` and `clang` with their respective optimization levels -O0, -O1, -O2, -O3, and -Os. In order to understand if it is necessary to use all of them in practice, we evaluate the number of bugs that can be detected by each subset of compiler implementations. We use the same Juliet tests as the previous evaluation. For each subset, *e.g.*, {`gcc-O0`, `gcc-O1`, `clang-O2`}, we modify CompDiff's configuration so that it only checks outputs from compiler implementations in the subset. We enumerate all possible subsets with sizes ranging from 2 to 10 and each subset does not contain duplicate compiler implementations.

The results are shown in Figure 1. We organize subsets according to their sizes. The X-axis means the size of each subset, *i.e.*, the number of compiler implementations in the subset. The Y-axis shows the number of bugs detected by each subset. We can find that with the increase of #compiler implementations, more bugs can be detected overall. For subsets of the same size, their detection ability varies a lot. For instance, when the number of compiler implementations equals 2, we have 45 subsets in combination. The number of bugs detected by them has a great difference as shown in the first box in Figure 1. As annotated in the figure, the best performing subset with size=2 is {`gcc-O0`, `clang-O3`}. Intuitively, these two compiler implementations maximize compilation differences: both from different compilers, one is an unoptimizing compiler while the other is an aggressively optimizing compiler. The worst performing subset with size=2 is {`gcc-O2`, `gcc-O3`}. The reason is that they have the same basic compiler and their optimizations are relatively similar. CompDiff's default setting achieves the best performance. Some small subsets, *e.g.*, 9, 8, or even 5, could detect nearly the same number of bugs as the default full size. We argue that this is due to the limited types of bugs in the Juliet tests. Discarding any compiler implementation risks missing bugs in practice.

Although the detection capability of small subsets may not be as good as the full size, they have lower run-time costs. For instance, using {`gcc-O0`, `clang-O3`} can detect ~98% bugs compared to the full size, however, only has ~20% run-time costs. If there are resource or time constraints, our results suggest that one can equip CompDiff with a smaller subset of compilers but should at least include two instances using different compilers and unoptimizing/(aggressively) optimizing optimizations.

Table 4: Details of selected target projects.

Target	Input type	Version	Size(LoC)
tcpdump	Network packet	4.99.1	99K
wireshark	Network packet	3.4.5	4.6M
objdump	Binary file	2.36.1	74K
readelf	Binary file	2.36.1	72K
nm-new	Binary file	2.36.1	55K
sysdump	Binary file	2.36.1	10K
openssl	Binary file	3.0.0	702K
ClamAV	Binary file	0.103.3	239K
libsndfile	Audio	1.0.31	66K
libzip	Compress tool	v1.8.0	29K
brotli	Compress tool	v1.0.9	55K
php	PHP	7.4.26	1.4M
MuJS	JavaScript	1.1.3	18K
pdftotext	PDF	4.03	130K
pdftoppm	PDF	21.11.0	203K
jq	json	1.6	46K
exiv2	Exiv2 image	0.27.5	384K
libtiff	Tiff image	4.3.0	37K
ImageMagick	Image	7.1.0-23	655K
grok	JPEG 2000	9.7.0	127K
libxml2	XML	2.9.12	458K
curl	URL	7.80.0	13K
gpac	Video	2.0.0	597K

4.3 CompDiff-AFL++

In this part, we evaluate the bug detection capability of CompDiff-AFL++ in real-world software.

Target projects. Table 4 lists details of all selected target projects. All these targets are well-studied and frequently used in the fuzzing community. They cover a broad range of functionalities including network packets analyzers, binary file analyzers, multimedia file processing, programming language implementations, compression algorithms, *etc.* The sizes of these projects range from 10KLoC to 4.6MLoC, further emphasizing their diversity.

Experimental setting. We used the same experimental environment as previous experiments. To comprehensively evaluate CompDiff-AFL++'s capability, we equipped CompDiff with all ten compiler implementations. The initial seeds for targets are from their official test suites. Seeds of type images and videos are expanded with Mozilla Fuzzdata [41]. After a bug was found, we reported it to developers and relied on their feedback to triage all reports. As

Table 5: Bugs detected by CompDiff-AFL++ on 23 open-source C/C++ projects.

	Unstable code due to undefined behavior					LINE	Misc.	Total
	EvalOrder	UninitMem	IntError	MemError	PointerCmp			
Reported	2	27	8	13	1	6	21	78
Confirmed	2	19	8	13	1	5	17	65
Fixed	2	15	6	12	1	5	9	52

has been discussed, automated triage is challenging in the context of differential testing, we will discuss it further in Section 5. To compare with sanitizers, for each program, we compiled it with ASan/UBSan and MSan to obtain sanitizer-enabled binaries. We then ran AFL++ on each binary and collected crashes found by sanitizers. All fuzzers had 24 hours timeout threshold and we repeated each fuzzing campaign 10 times.

Summary. Table 5 summarizes the number of bugs detected by CompDiff-AFL++. We categorize bugs based on their root causes. In total, CompDiff-AFL++ reported 78 bugs, 65 of which were confirmed by developers, and 52 were already fixed. The results demonstrate that CompDiff-AFL++ is effective and useful for identifying unstable code. Interestingly, as shown in the "LINE" and "Misc." columns, CompDiff-AFL++ detected not only undefined behaviors but also other real bugs due to various issues. We next analyze these bugs in detail. We guide our analysis with six consecutive research questions.

➤ RQ1: What kinds of unstable code does CompDiff-AFL++ find?

As shown in the "Unstable code due to undefined behavior" column in Table 5, we classify the found unstable code by their root causes into five categories. Note that, unstable code that can be covered CompDiff-AFL++ is in principle not limited to these categories. For example, we did not find any bug related to CWE-469, on which CompDiff has shown its strong detection ability in Table 3. We anticipate CompDiff-AFL++ can detect a broader range of unstable code when evaluating on more software. We next show our analysis of each category.

EvalOrder. When the evaluation order of subexpressions has conflict side effects, the result becomes unstable. An example has been shown in Listing 3 in Section 2. These two bugs are all found in Tcpdump. After we reported these issues, the tcpdump developers quickly fixed it. They also manually diagnosed that the other 7 locations potentially had the same issue and fixed them as well. The developer commented that their fixes are just for *"less disruptive"* to the current code base and *"We should consider the cleaner long-term mechanism for 5.0 or later to get rid of static buffers"*.

UnitMem. Bugs due to uninitialized memories are classified into this category. We have shown an example in Listing 4 in Section 2. UnitMem bugs appear the most. The reason is that values of uninitialized variables depend on run-time memory layout, which often changes per binary. Some bugs in this category could be detected by MSan as well. We will compare our CompDiff with MSan on these bugs in **RQ3**.

IntError. Integer overflow/underflow can cause unstable code. The example in Section 1 shows a case where a code fragment is discarded due to integer overflow. Sometimes integer overflow may lead to inconsistent results across compiler implementations. For instance, the following code

```
1  int a, b;
2  long x, y;
3  ...
4  x = y + a * b;
```

contains a potential signed integer overflow in line 4 when `a*b` exceeds the range of `int`. In most cases, `a*b` is first calculated and the result is then represented and stored to an `int`. However, some compiler implementations such as `clang-O1` first cast both `a` and `b` into `long` and then does the calculation. These two routines store different results to `x` when `a*b` overflows and thus becomes unstable.

MemError. Bugs in this category are caused by memory-related errors such as buffer-overflow and use after free. On one hand, compilers can assume these errors never occur and transform code having MemError arbitrarily at compile time. On the other hand, these errors cause corrupted memory states and thus unstable program states. These bugs can be detected by ASan in principle. We will compare our CompDiff with ASan on these bugs in **RQ3**.

PointerCmp. Comparing pointers pointing to different objects/unions/structs is undefined. The comparison result can be either true or false depending on the compiler's choice. CompDiff-AFL++ detected one such bug in readelf. We have discussed this bug in Listing 2 in Section 2.

LINE. The C17 standard specifies multiple permissible behaviors for __LINE__ macro [9] (§6.10.4). The interpretation of __LINE__ is implementation-defined behavior, , meaning that different compiler implementations may have divergent interpretation results. CompDiff-AFL++ detected such inconsistencies in `readelf`, `ImageMagick`, `Wireshark`, `libtiff`, and `php`. For instance, for the following php code with the bug in line 3,

```
1  <?php
2  $a = 0;
3  var_dump($b::class);
4  ?>
```

some php interpreters compiled from different compilers incorrectly label line 2 instead of 3 as buggy.

Miscellaneous. Surprisingly, CompDiff-AFL++ detected 21 real bugs that are not due to undefined behavior. Their root causes are miscellaneous. One interesting category is compiler bugs/issues, where compilers instead of application programs are blamed for the divergent output. We will further discuss this kind of bugs in the next RQ. CompDiff detected many other program-specific issues including, but not limited to, bad random value (libtiff), printing pointer address instead of value (objdump), and even unknown reasons (wireshark). All of these issues lead to divergent outputs on the same input, and thus none of them are false positives. These bugs demonstrate CompDiff's strong ability in exposing unstable issues.

➤ RQ2: Did CompDiff-AFL++ detect any compiler bug or issue?

In our CompDiff's design, we assume compilers are bug-free. In practice, compiler bugs rarely affect real-world programs' integrity [30]. According to our unstable code definition, compiler bugs or issues can indeed cause divergent program outputs and thus be caught by CompDiff. On the 23 real-world programs, we found 3 compiler bugs due to miscompilation and 4 compiler issues due to floating point imprecision. Next, we discuss them in detail.

Compiler miscompilation. CompDiff reported two compiler miscompilations in `gcc` and one in `clang`, all of which were found during fuzzing `MuJS`. Unlike other bugs introduced by program developers, compilers are blamed for these bugs. After we reported these bugs, the `MuJS` quickly confirmed these issues and proposed solutions to temporarily avoid compiler miscompilations in their code. Although compiler issues have been detected, we do not anticipate CompDiff can be used to intensively find compiler miscompilations. Nevertheless, these issues are real bugs that affect program correctness. CompDiff helps identify them and guarantees program's integrity across compilations.

Floating point imprecision. Different compilers or optimizations may use different strategies to calculate floating point data. For example, `clang-O3` sometimes transforms `pow()` to the more efficient `exp2()` libcall and has different decimal results from others. Rounding strategies in different compiler implementations may also cause divergent results. Strictly speaking, this is not a program bug, but a compiler issue. We reported four such cases and developers confirmed three of them. Only `brotli`'s developers committed to fixing the reported bug, which was because floating-point imprecision affected the internal state of a compression algorithm, making the compressed file to be different across compiler implementations. We do not consider any of these bugs as false positives because they indeed lead to inconsistent results across compilations. Compared to other bugs, floating point imprecision is relatively less serious in most cases.

➤ RQ3: How many bugs found by CompDiff-AFL++ can be covered by sanitizers?

Some of the detected bugs can in principle be captured by sanitizers. Specifically, *MemError* by ASan, *IntError* by UBSan, and *UninitMem* by MSan. For each bug detected by CompDiff, we check if it can be discovered by sanitizers. Recall that AFL++ with ASan, UBSan, and MSan were run on each target 10 times, we collected all reports produced by them. For each bug reported by CompDiff, we performed manual analysis to identify whether or not sanitizers' reports cover it. Table 6 shows the summarized result. Out of 78 bugs detected by CompDiff, 42 of them can also be covered by sanitizers. The left 36, however, cannot. Specifically, for *MemError* and *IntError*, ASan and UBSan successfully detected all of them. MSan, however, only exposed 21 out of 27 *UninitMem* bugs. In fact, we also observed that many bugs detected by sanitizers during fuzzing cannot be detected by CompDiff, either. The observation is aligned with our findings in the Juliet tests. As we have been emphasizing, CompDiff is not to replace sanitizers but to complement them in exposing more bugs in practice. The unique 36 bugs detected by CompDiff support our claim.

Table 6: Of all the bugs detected by CompDiff, the number of bugs that can also be discovered by sanitizers.

	Sanitizers				CompDiff
	ASan	UBSan	MSan	**Total**	
MemError	13	-	-	13	13
IntError	-	8	-	8	8
UninitMem	-	-	21	21	27
Remaining bugs	-	-	-	0	30
Total				42	78

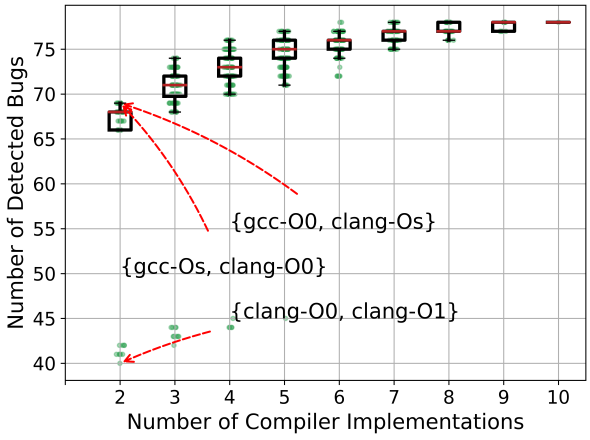

Figure 2: Number of bugs could be detected by each subset of compiler implementations by CompDiff-AFL++.

➤ RQ4: Impact of #compiler implementations in CompDiff-AFL++.

We have analyzed on the Juliet tests the impact of different subsets of compiler implementations in CompDiff. To examine if a consistent tendency can be observed in CompDiff-AFL++, we also evaluate each subset of compiler implementations on the 78 real bugs. Figure 2 presents the results. Compared to our previous

results in Figure 1, similar conclusions can be drawn: More compiler implementations bring higher bug detection rates in CompDiff-AFL++; different compilers with unoptimizing and aggressively optimizing levels can bring us the best performance while similar compiler implementations are less effective.

➤ RQ5: Handling non-deterministic or multi-threaded programs in CompDiff-AFL++.

Recall that CompDiff-AFL++ is designed to find unstable code in programs with deterministic output. Non-deterministic programs are typically concurrent or multi-threaded programs. They can have either deterministic outputs, *i.e.*, running the program on the same input always emits the same output, or inconsistent outputs, *i.e.*, outputs may change per run. For non-deterministic or multi-threaded programs, CompDiff can handle them as long as they have deterministic output. In fact, among our evaluated 23 real-world programs, 6 of them are non-deterministic or multi-threaded programs, *i.e.*, `tcpdump`, `wireshark`, `MuJS`, `ImageMagick`, `grok`, and `gpac`. In total, CompDiff-AFL++ found 23 bugs in them, which shows that CompDiff is capable of handling non-deterministic programs that have deterministic outputs.

For non-deterministic programs without deterministic output, our experience is that many of the non-determinism happen when programs deliberately include random numbers or timestamps in their outputs. Such non-determinism can be easily eliminated with a post-processing script on the program's output. For example, the output of `wireshark` includes the timestamp when it generates warnings:

```
10:44:23.405830 [Epan WARNING]
```

Different binaries are thus emitting inconsistent warning messages. To filter out these values, we used a regular expression to match and remove all these timestamps in its outputs.

➤ RQ6: False positives of CompDiff-AFL++.

Theoretically, CompDiff has no false positives since all test inputs saved by CompDiff are guaranteed to trigger discrepancies across at least two compiler implementations. However, we still need to deal with cases where some but not all binaries timeout. For efficiency reasons, AFL++ deliberately terminates a binary after a timeout threshold. Such terminations will inevitably truncate a binary's output, causing output discrepancy between terminated binary and timed-out binary. In our CompDiff-AFL++, when a generated input times out on partial binaries, we let CompDiff-AFL++ save it first and then increase its timeout threshold until it terminates. It is theoretically possible that a binary hangs forever. However, our experience is that as long as one of the compiled binaries terminates, others will terminate eventually but may have longer execution time.

As we discussed before, false positives are also possible when using CompDiff on non-deterministic programs that do not have deterministic output. We have shown in RQ5 that many of the non-determinism in the output have fixed patterns such as random number and timestamp and can thus be eliminated. We consider the general handling of programs with non-deterministic output as a limitation of CompDiff and discuss further in Section 5.

5 Discussion

Fault localization and bug report: When a program failure is found, it is beneficial to automatically and accurately localize the root cause in the source code. Fault localization techniques, in general, although extensively studied [49], are not yet practical [5, 36]. Sanitizers provide function stack traces to help developers pinpoint the root causes of crashes. Since bugs found by CompDiff do not necessarily lead to crashes, such stack trace-based approaches are not applicable. As all tests reported in CompDiff result in different outputs across binaries, it is possible to compare execution traces from different binaries to pinpoint the root cause. Aligning executions on two binaries is challenging in general. Although CompDiff has the advantage that all binaries are compiled from the same source code, compilers, especially optimizations, will result in huge differences in control flows and variable values. It would be interesting for future work to explore how to accurately and efficiently align and compare multiple execution traces.

Although we do not analyze the root cause bugs with CompDiff, our current bug reports are useful for developers to diagnose and fix bugs. In our bug reports, we include the following information: 1) *test input* that triggers the bug, 2) *two or more compiler configurations* that can be used to reproduce the bug, and 3) *the divergent outputs* on the provided test input. With our current reports, at the time of writing, 54 out of the 78 reported bugs had already been fixed by developers. Thus, there is clear evidence that our reports are useful as developers are able to use them to quickly diagnose these issues.

Limitations: CompDiff has detected many unknown bugs in real-world software. All these bugs lead to inconsistent/incorrect outputs. CompDiff incurs no false positive in programs with deterministic output. Although all UBs by definition could lead to unstable code, existing compiler implementations may not exploit all of them. Thus CompDiff cannot detect all UBs in practice. For unstable code, CompDiff cannot discover all of them for two reasons. First, not all erroneous states of unstable code propagate to final outputs. Since CompDiff concerns output only, there is no way to detect vanishing errors. Second, not all unstable code leads to inconsistent behaviors. At run-time, binaries may enter the same erroneous state, *e.g.*, choosing the same uninitialized value, and emitting identical but incorrect output. Extending CompDiff to a program's internal states would incur huge analysis overhead and might be generally infeasible and impractical. The succinct design of CompDiff guarantees its generality and scalability. Another limitation of CompDiff is in handling non-deterministic programs that have non-deterministic output. An example is a multi-threaded program where multiple threads concurrently print into the standard output while there is no requirement on the order. In such cases, divergent outputs across runs are normal and thus CompDiff will not be useful in detecting errors. However, during our testing, we did not encounter such cases and all our targeted non-deterministic programs have deterministic output.

Overhead: Our experimental results suggest that we should enable all ten compiler implementations when using CompDiff in practice. The run-time overhead is roughly 10x normal execution. However, our evaluations on the Juliet tests and real-world software also

reveal the fact that a smaller subset of compiler implementations can give us a similar bug detection rate. For example, using only `clang-O0` and `gcc-Os` in CompDiff-AFL++ can discover 69 out of total 78 bugs. The run-time overhead can be reduced from roughly 10x to 2x normal execution. When using CompDiff in practice, we suggest users enable as many compiler implementations as possible to maximize its bug-finding capability. If resources are constrained, users should enable at least different compilers with diverse optimization levels.

Improvements and future work: The current design of CompDiff-AFL++ keeps the fuzzer's core logic and applies differential testing on each generated input. Its effectiveness largely depends on the quality of test inputs. AFL++ utilizes code coverage feedback from executions to guide its mutation strategies. If we incorporate the divergence observed from different binaries into the feedback, there is a potential that AFL++ could generate more unstable code-triggering test inputs. NEZHA [37] takes a similar approach by exploiting behavioral asymmetries between multiple programs to improve AFL's efficacy in generating inputs that are likely to trigger semantic bugs. Its design is for different implementations on the same specification such as OpenSSL and LibreSSL. In CompDiff, binaries are compiled from the same source code. Aligning execution paths and finding execution divergence is much easier than NEZHA's situation. Integrating execution divergence into fuzzers' feedback may enable CompDiff to find a lot more unstable code. We believe this will be an interesting future work to further enhance compiler-driven differential testing.

6 Related Work

Fuzz testing: Fuzzing is the most popular random testing technique for discovering security vulnerabilities in software [20, 32]. Due to its simplicity and effectiveness, coverage-guided greybox fuzzers such as AFL [50], LibFuzzer [45], and honggfuzz [19] are widely deployed and have been the fundamental tools for many other advanced fuzzers. To boost fuzzing performance, a lot of research efforts have been put into improving various aspects of fuzzing, such as seed scheduling [3, 4], mutation strategies [2, 29], and path explorations [23, 44]. Our proposed CompDiff could serve as a plug-and-play oracle for all these fuzzers to extend their capability from discovering security defects to finding semantic bugs.

Differential testing: Researchers have leveraged differential testing to find semantic bugs across many types of programs, such as database management systems [39], Java Virtual Machine (JVM) implementations [8], REST APIs [18], and compilers [27]. Incorporating different testing into fuzzing engines is also gaining more and more interest for discovering functional bugs. For example, HeteroFuzz [51] detects platform-dependent divergence for heterogeneous applications running on both CPU and FPGA. DifFuzz [34] discovers side-channel leakages by analyzing two executions on the same binary. DIFUZZRTL [24] finds CPU bugs by differentially comparing multiple CPU RTLs. It develops a novel register coverage for higher fuzzing efficiency and efficacy.

Each of these differential fuzzing efforts focuses on one specific domain. In contrast, CompDiff utilizes a generally applicable test oracle that concerns a totally different kind of bugs, *i.e.*, unstable code. Another notable difference is that most of them incorporate new designs to fuzzers in order to improve the possibility of discovering bugs. CompDiff does not make changes to fuzzers core logic and has been proven to be effective in finding interesting bugs. Designing new feedback for CompDiff should be an interesting research direction for future exploration.

N-version programming: The concept of N-version programming (NVP) was first introduced in 1978 by Chen and Avizienis [7]. It aims at improving fault tolerance of software by having N independent individuals or groups implementing the same specification. Recent and practical applications of NVP are to opportunistically leverage existing diverse software implementations. For example, Frost [46] executes multiple replicas with complementary thread schedules to protect a program from data race errors; Varan [22] utilizes system call level synchronisation to realize N-version execution systems; many differential testing approaches discussed early are also instances of NVP such as Chen *et al.* [8] that leverages different JVM implementations like Oracle's HotSpot and IBM's J9. CompDiff employs different compiler implementations to obtain N-version binaries of a program. We do not require any modification to a program's source code or its execution environment. In contrast to many differential testing approaches, where additional implementations of a target program are necessary, CompDiff, however, is not subject to this requirement. We argue that our employment of compiler implementations to get multiple replicas is novel and unique compared with the existing NVP schemes.

Finding undefined behavior: Undefined behavior in C/C++ covers a wide range of illegal program states. Apart from sanitizers, there are also many other popular tools such as Dr. Memory [6] and Valgrind [33] that require no compile-time instrumentation and detect errors at the binary level. Static tools such as STACK [48], Infer [31], and Cppcheck [12] cal also cover frequently occurring UBs. CompDiff is orthogonal to them as 1) as a dynamic tool, CompDiff discovers bugs that are beyond the reach of static tools, and 2) developers can always use static tools, as well as dynamic tools, in different development stages for finding more bugs.

7 Conclusion

We have introduced compiler-driven differential testing (CompDiff), a simple, straightforward, yet effective approach for finding unstable code in C/C++ programs. CompDiff concerns program input/output behaviors across metamorphic compiler implementations. The succinct design of CompDiff poses no constraint on the underlying programming language and is generally applicable.

We also integrated CompDiff into AFL++ to improve our work's practicality. Our extensive evaluation on both benchmark and real-world programs confirmed that CompDiff is effective in covering a broad range of unstable code and significantly complements existing sanitizers by finding many unique bugs. We expect our study to inspire further the community to explore the impact and detection of unstable code.

References

[1] Austin Appleby. 2016. MurmurHash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3. Accessed: March 7, 2022.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[3] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[5] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 117–128.

[6] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 213–223.

[7] Liming Chen and Algirdas Avizienis. 1978. N-version Programming: A Fault-tolerance Approach to Reliability of Software Operation. In *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS-8)*, Vol. 1. 3–9.

[8] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[9] JTC1/SC22/WG14 The C Standards Committee. 2018. ISO/IEC 9899:2018, Programming languages — C. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

[10] JTC1/SC22/WG21 The C++ Standards Committee. 2020. Standard for Programming Language C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf.

[11] Mila Dalla Preda, Roberto Giacobazzi, Arun Lakhotia, and Isabella Mastroeni. 2015. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 329–341.

[12] CppCheck developers. 2022. A Tool for Static C/C++ Code Analysis. Retrieved October 15, 2022 from http://cppcheck.sourceforge.net/

[13] GCC developers. 2022. Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed: March 7, 2022.

[14] LLVM developers. 2022. Clang - the Clang C, C++, and Objective-C compiler. https://clang.llvm.org/docs/CommandGuide/clang.html. Accessed: March 7, 2022.

[15] LLVM developers. 2022. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: March 7, 2022.

[16] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[18] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 312–323.

[19] Google. 2022. Honggfuzz. https://honggfuzz.dev/. Accessed: March 7, 2022.

[20] Google. 2022. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: March 7, 2022.

[21] The Tcpdump Group. 2022. TCPDUMP. https://github.com/the-tcpdump-group/tcpdump. Accessed: March 7, 2022.

[22] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 339–353.

[23] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.

[24] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1286–1303.

[25] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 323–333.

[26] lcamtuf. 2014. Fuzzing random programs without execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html. Accessed: March 7, 2022.

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 216–226.

[28] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22), July 18–22, 2022, Virtual, South Korea*.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[30] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[31] Meta. 2022. A Tool to Detect Bugs in Java and C/C++/Objective-c Code. Retrieved October 15, 2022 from https://fbinfer.com/

[32] Microsoft. 2022. One Fuzz: A self-hosted Fuzzing-As-A-Service platform. https://github.com/microsoft/onefuzz. Accessed: March 7, 2022.

[33] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation *(PLDI '07)*. 89–100.

[34] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. 2019. DiffFuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 176–187.

[35] NIST. 2017. Juliet Test Suite for C/C++ 1.3. https://samate.nist.gov/SARD/test-suites/112.

[36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[37] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*. IEEE, 615–632.

[38] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 142–157.

[39] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.

[40] Coverity Scan. 2022. Coverity Scan: Find and Fix Defects in Your Java, C/C++, C#, JavaScript, Ruby, or Python Open Source Project for Free. Retrieved October 15, 2022 from https://scan.coverity.com/

[41] Mozilla Security. 2021. Fuzzdata. https://github.com/MozillaSecurity/fuzzdata. Accessed: March 7, 2022.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.

[43] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 46–55.

[44] 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 *NDSS*, Vol. 16. 1–16.

[45] The LLVM team. 2022. Honggfuzz. https://llvm.org/docs/LibFuzzer.html. Accessed: March 7, 2022.

[46] Kaushik Veeraraghavan, Peter M Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and surviving data races using complementary schedules. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. 369–384.

[47] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *Proceedings of the Asia-Pacific Workshop on Systems*. 1–7.

[48] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 260–275.

[49] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[50] Michal Zalewski. 2014. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed: March 7, 2022.

[51] Qian Zhang, Jiyuan Wang, and Miryung Kim. 2021. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 242–254.