UBfuzz: Finding Bugs in Sanitizer Implementations

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

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

Abstract

In this paper, we propose a testing framework for validating sanitizer implementations in compilers. Our core components are (1) a program generator specifically designed for producing programs containing undefined behavior (UB), and (2) a novel test oracle for sanitizer testing. The program generator employs Shadow Statement Insertion, a general and effective approach for introducing UB into a valid seed program. The generated UB programs are subsequently utilized for differential testing of multiple sanitizer implementations. Nevertheless, discrepant sanitizer reports may stem from either compiler optimization or sanitizer bugs. To accurately determine if a discrepancy is caused by sanitizer bugs, we introduce a new test oracle called *crash-site mapping*.

We have incorporated our techniques into UBfuzz, a practical tool for testing sanitizers. Over a five-month testing period, UBfuzz successfully found 31 bugs in both GCC and LLVM sanitizers. These bugs reveal the serious false negative problems in sanitizers, where certain UBs in programs went unreported. This research paves the way for further investigation in this crucial area of study.

CCS Concepts: • **Software and its engineering → Compilers**; • **Security and privacy → Software and application security**.

Keywords: Undefined Behavior, Sanitizer, Compiler, Program Generation, Fuzzing

ACM Reference Format:
Shaohua Li and Zhendong Su. 2024. UBfuzz: Finding Bugs in Sanitizer Implementations . In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617232.3624874

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0372-0/24/04…$15.00
https://doi.org/10.1145/3617232.3624874

1 Introduction

Undefined behaviors (UB), such as buffer overflow, integer overflow, *etc.*, are often responsible for creating security weaknesses in software [22, 40]. Sanitizers are crucial in enabling the large-scale detection of security vulnerabilities caused by UB [29, 30]. Popular sanitizers include Address Sanitizer (ASan) [31] for memory access errors, Undefined Behavior Sanitizer (UBSan) [5] for various undefined behaviors, and Memory Sanitizer (MSan) [32] for uninitialized memory uses. Technically, sanitizers are integrated into compilers. When a sanitizer is enabled, various checks are inserted into a program during compilation. If a check is violated at run-time, an error is reported. Owing to their superior capability and usability, sanitizers have assisted developers in discovering numerous critical vulnerabilities. For instance, by fuzzing with sanitizers, the Google OSS-Fuzz project has reported over 20K UBs in hundreds of open-source projects [4, 6]. While substantial research and engineering efforts have been made toward devising efficient fuzzers [15, 24] and reducing sanitizer costs [11, 43, 44], the robustness and reliability of sanitizers — essential for detection effectiveness — have received little attention from both academia and industry.

Both GCC and LLVM, the two most popular C/C++ compilers, support sanitizers. Over the past five years, there were only 29 bug reports related to sanitizer correctness in the bug trackers of GCC and LLVM. Most of these reports (66%) were false positive issues, where sanitizers did not miss UBs but instead incorrectly reported correct executions as containing UB. False positive issues are indeed easy to be noticed in practice. For example, typical compiler testing work [12, 18, 42] involves generating valid programs as input, which can be trivially adapted to identify false positive issues. Conversely, false negative bugs in sanitizers typically result in a UB being missed and are thus difficult to be observed.

Figure 1 illustrates a code snippet that triggers a false negative bug in GCC ASan. Since `d` points to the starting location of `b[2]`, the dereference `*(d+k)` at line 8 will cause a stack-buffer-overflow. When we compile and run this code with GCC ASan at -O0, ASan crashes the execution and generates a report as expected (Figure 1 top right). However, at -O2, it unexpectedly misses this UB (Figure 1 bottom right). This is a false negative bug of GCC ASan. As a UB detection tool, false negative bugs in sanitizers lead to missing UBs, thereby significantly impeding their effectiveness.

```
1 struct a { int x };
2 struct a b[2];
3 struct a *c=b, *d=b;
4 int k = 0;
5 int main() {
6   *c = *b;
7   k = 2;
8   *c = *(d+k);
9   return c->x;
10 }
```

```
(command line)
$ gcc -O0 -fsanitize=address a.c
$ ./a.out
==1==ERROR: AddressSanitizer:
stack-buffer-overflow in a.c:8
$

(command line)
$ gcc -O2 -fsanitize=address a.c
$ ./a.out
$
```

Figure 1. Line 8 in `a.c` contains a stack-buffer-overflow (left). GCC ASan at -O0 successfully detects it (top right). GCC's Asan at -O2, however, overlooks it (bottom right). [1]

In this paper, we aim to detect false negative (FN) bugs in sanitizers. Despite its criticality and importance, to the best of our knowledge, there exists no work that has systematically investigated this problem. We introduce the first effective testing framework for finding FN bugs in sanitizers. At a high level, the general testing workflow is (1) generating a UB program, *i.e.*, a program exhibiting undefined behavior, and (2) compiling it with sanitizers and executing the compiled binary. If no sanitizer report on a UB program is produced, a potential sanitizer bug is detected. Two main challenges exist, which we will discuss next.

Challenge 1: *UB program generation.*
To detect FN bugs, abundant and diverse UB programs should be available. The automated generation of valid programs for compiler testing has been extensively researched. Tools like Csmith [42] can generate a wide variety of valid C programs that are free from UB. However, the generation of programs exhibiting various types of UB, which is essential for sanitizer testing, remains unexplored. For instance, to test ASan, programs with memory safety bugs such as buffer-overflow, use-after-free, use-after-scope, *etc.*, are needed. One might consider randomly mutating a valid program, for example, deleting statements or altering variable values, to introduce UBs. As we will demonstrate in our evaluation §4.3, this naive mutation-based method is ineffective in generating UB programs—most of the mutated programs do not have UB. Furthermore, the generated programs encompass only a few UB types and are unable to find any FN bugs in sanitizers.

Our solution: *Shadow Statement Insertion.*
We propose a general approach for introducing UB into a valid program. Given a program and a target UB such as buffer overflow, our approach first applies static and dynamic analysis to learn the program's runtime state and identify a specific program location where the target UB can be introduced. Subsequently, we insert a new statement into the program such that the chosen program location triggers a UB. We term our approach *shadow statement insertion*. For instance, the original program of Figure 1 does not have line 7 and is thus free of UB. To introduce a buffer overflow, our tool analyzes the code and identifies that the pointer `d` points to the stack buffer `b` of size 8 bytes. It then inserts `k=2` to overflow the buffer access at line 8. As will be detailed in §3, following the same framework, our design can be generalized to other UB types.

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=105714

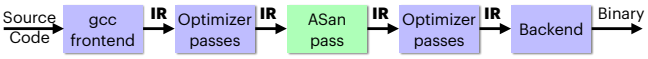

Figure 2. The high level compilation pipeline of ASan in GCC/gcc.

Challenge 2: *Compiler optimization significantly complicates sanitizer testing.*
Given an input UB program, a natural approach is to examine whether a compiler's sanitizer such as `gcc -O2 -fsanitize=address` can detect the UB. If no report is produced, one might assume that "a sanitizer FN bug is discovered". However, this is not true due to compiler optimizations.

Sanitizers are implemented as passes in compilers' pipeline. Figure 2 illustrates the high-level pipeline in GCC compilation with ASan enabled. The ASan pass collaborates with other optimizer passes to compile a program. Previous research [10, 16, 41] has shown that compiler optimizers always presume that the input program does not contain UB, resulting in the elimination of certain UBs by optimizer passes. Figure 3 provides an example where both `d[1]=1` at line 4 and `*b` at line 5 trigger stack-buffer-overflow UB. Nonetheless, if we compile it with ASan at -O2 (`gcc -O2 -fsanitize=address`), no UB report will be produced. *Different from the previous example in Figure 1, this is not a sanitizer bug*. The reason is that early optimization passes at GCC -O2 optimize away all the UB code, as depicted on the right side of Figure 3. Since there is no UB present in the input IR to the ASan pass, ASan cannot uncover the UB in the source code. As there is no UB in the final compiled binary, ASan is not considered buggy in this example.

A natural follow-up question is *can we only consider unoptimized compilers such as with -O0?* The answer is no for two main reasons. First, even with -O0, some basic optimizations, such as constant folding, may still optimize away the UB code. Second, many sanitizer FN bugs only exist at higher optimization levels as demonstrated in Figure 1. Testing sanitizers only at -O0 may fail to detect many critical FN bugs.

Similarly, differential testing across different compilers is ineffective as it is impossible to determine whether a discrepant report is caused by a sanitizer FN bug or merely due to compiler optimizations. For instance, although GCC ASan at `-O0` and `-O2` produce different results in both Figure 1 and 3, the latter is caused by compiler optimizations.

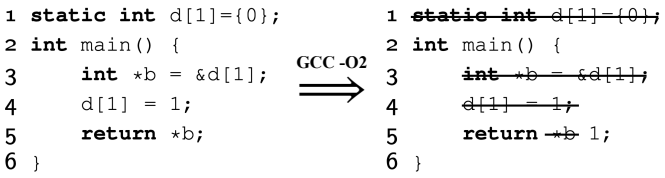

Figure 3. GCC -O2 optimizes away the UB code, thus ASan cannot discover the UB.

Our solution: *Crash-site mapping as the test oracle.* We introduce a novel test oracle, *crash-site mapping*, to accurately discern whether discrepant sanitizer reports stem from sanitizer FN bugs or compiler optimizations. Given two binaries, b_c and b_n, compiled by two compilers such as GCC ASan at `-O0` and `-O2`, executing b_c results in a crash while executing b_n exits normally. Here, the crash of b_c means that the sanitizer successfully reports the UB while the normal exit of b_n means that the sanitizer does not report the UB. Our primary approach involves using a debugger to trace the execution of both binaries. If the crash location in b_c is also executed by b_n, we can infer that the compiler does not eliminate the UB, and thus, it is highly probable that a sanitizer FN bug is present. A more detailed example will be provided in Section 2. As our evaluation will demonstrate, our crash-site mapping oracle can effectively identify discrepancies caused by compiler optimizations.

We realized our solutions in a tool named UBFuzz. It can automatically generate a substantial number of UB programs and accurately identify sanitizer FN bugs. The example we showcased in Figure 1 was found by UBFuzz. We reported this FN bug to the GCC team, who confirmed and fixed it. The root cause is that in some cases, GCC ASan would "forget" to insert checks to specific memory accesses, thus resulting in missed UB reports. During a five-month testing period, UBFuzz uncovered a total of 31 new FN bugs in ASan, UBSan, and MSan from both GCC and LLVM. We open-sourced our implementation to facilitate future research[2].

In summary, we make the following contributions:

- We introduce a general approach, *Shadow Statement Insertion*, for generating UB programs.
- We design *crash-site mapping* as an effective oracle for sanitizer testing.
- Based on the proposed UB generator and test oracle, we develop the automated tool UBFuzz for testing sanitizers.
- We report our extensive evaluation of UBFuzz, which successfully identified 31 sanitizer FN bugs.

2 Illustrative Examples

This section illustrates (1) how UBFuzz generates UB programs for a target UB, and (2) how our crash-site mapping test oracle works.

[2]https://github.com/shao-hua-li/UBGen

```
1 struct a { int x };
2 struct a b[2];
3 struct a *c = b, *d = b;
4 int k = 0;
5 int main() {
    LOG_BufRange(&b[0], sizeof(b));①
6   *c = *b;
    k = 2; ③
    LOG_BufAccess(d+k);②
7   *c = *(d+k);
8   return c->x;
9 }
```

Figure 4. Code instrumentation for UB insertion.

2.1 UB Program Generation

The uncolored (black) code in Figure 4 is the seed program. We now demonstrate how we mutate this seed program to the UB program in Figure 1, with the goal of introducing a *stack-buffer-overflow* UB. Our generator works as follows:

Step 1. Insert profiling statements for all stack buffers. Since there is only one global stack buffer `b[2]` in this seed program, a single profiling statement is inserted as indicated by ①. Next, identify all code constructs with the potential to exhibit the target UB. Given that our target UB is stack-buffer-overflow, we statically locate all memory accesses. All the pointer dereferences `*b`, `*c`, and `*(d+k)` at lines 6 and 7 are eligible. For the simplicity of presentation, we only show the profiling statement ② for `*(d+k)`.

Step 2. After the instrumentation, we compile and execute the code to obtain its runtime information including the memory range of buffer `b` and the memory address of `d+k`.

Step 3. To introduce a stack-buffer-overflow for `*(d+k)` at line 7, we mutate the value of `k` such that it overflows the pointed-to buffer. Since we have learned that the buffer `b` has a size of 8 bytes and `d` points to the starting location of `b`, we insert the shadow statement `k=2` ③ to introduce a stack-buffer-overflow at line 7. One might question whether setting `k` to an arbitrarily large enough value would also introduce a buffer overflow. However, due to the design limitation of ASan, it can only detect overflows of up to 32 bytes. Consequently, the valid range of overflowed addresses falls between 8 ~ 32 bytes beyond `b`. Our precise runtime analysis enables us to precisely mutate `d+k` such that it falls within this range.

Finally, all logging statements ①② are removed while the shadow statement ③ is kept. The resulting UB program is identical to the one presented in Figure 1.

2.2 Crash-Site Mapping as the Test Oracle

After a UB program is generated, we use at least two compilers with sanitizer enabled to compile it. We then execute the

Table 1. UB conditions and shadow statements. The first three columns describe the conditions for certain code constructs to not have the target UB. The fourth column demonstrates the location where our shadow statements will be inserted. The fifth column presents the effect of each shadow statement. The last column lists the instantiation of each shadow statement in our implementation. Here, $x, \widehat{x}, y, \widehat{y}, \widehat{c}$ are $(n+1)$-bit integers; p, q are pointers; a is an array with capacity $\textsc{ArraySize}(a)$; $lhs \xrightarrow{val} rhs$ represents the value of lhs is rhs.

UB	Code Constrcut	Sufficient condition for not having the UB	Shadow Statement $\Delta(\cdot)$	Effect of $\Delta(\cdot)$	Instantiation
Buf. Overflow (Array)	$a[x]$	$0 \le x < \textsc{ArraySize}(a)$	$\Delta(x)$; `Stmt{`$a[x]$`};`	$x \xrightarrow{val} v$ and $(v < 0 \vee v \ge \textsc{ArraySize}(a))$	$\widehat{x} = v - x$; `Stmt{`$a[x + \widehat{x}]$`};`
Buf. Overflow (Pointer)	$*p$	$p \in \textsc{BufferRange}(p)$	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} q$ and $q \notin \textsc{BufferRange}(p)$	$\widehat{c} = q - p$; `Stmt{`$*(p + \widehat{c})$`};`
Use After Free	$*p$	$\forall free(q)$, !alias(p, q)	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} q$ and q is freed	$free(p)$; `Stmt{`$*p$`};`
Use After Scope	$*p$	$\textsc{Scope}(*p) \in \textsc{Scope}(p)$	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} q$ and $\textsc{Scope}(*q)$ out of $\textsc{Scope}(p)$	$p = q$; `Stmt{`$*p$`};`
Null Ptr. Deref.	$*p$	$p \neq \textsc{Null}$	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} \textsc{Null}$	$p = 0$; `Stmt{`$*p$`};`
Integer Overflow	x op y	x op $y \in [-2^n, 2^n - 1]$	$\Delta(x, y)$; `Stmt{`x op y`};`	$x \xrightarrow{val} v_0, y \xrightarrow{val} v_1$ and v_0 op $v_1 \notin [-2^n, 2^n - 1]$	$\widehat{x} = v_0 - x, \widehat{y} = v_1 - y$; `Stmt{`$(x + \widehat{x})$ op$(y + \widehat{y})$`}`
Shift Overflow	$x \ll y$ or $x \gg y$	$0 \le y < n$	$\Delta(y)$; `Stmt{`$x \ll y$`};`	$y \xrightarrow{val} v$ and $v < 0 \vee v \ge n$	$\widehat{y} = v - y$; `Stmt{`$x \ll (y + \widehat{y})$`};`
Divide by Zero	x/y or $x\%y$	$y \neq 0$	$\Delta(y)$; `Stmt{`x/y`};`	$y \xrightarrow{val} 0$	$\widehat{y} = -y$; `Stmt{`$x/(y + \widehat{y})$`};`
Use of Uninit. Memory	`if(`x`)` or `while(`x`)`	x is uninitialized	$\Delta(x)$; `Stmt{`x`};`	$x \xrightarrow{val}$ uninit. memory	int $\widehat{x}$; `Stmt{`$x + \widehat{x}$`};`

```
...              #line,offset
andl  %r8d, %esi #10,8
movq  %rax, %rdi #10,8     movq  ptr(%rip),%rax #10,3
callq 0x10e0     #10,8     movl  $0xfff,(%rax)  #10,8
```

(a) The last three executed instructions in b_c.

(b) The executed instructions that are from line 10 in b_n.

Figure 5. Partial executed instructions in b_c and b_n. The comment shows the corresponding line and offset in the source code.

compiled binaries to examine whether there is a discrepant report. If one of the binaries crashes while the other does not, we need to determine if the discrepancy is caused by compiler optimizations. We refer to the crashing binary as b_c and the non-crashing binary as b_n. For the code snippet in Figure 4, b_c is compiled by GCC ASan at -O0, while b_n is from -O2. Our crash-site mapping works as follows:

Step 1. *Analyze b_c:* We utilize a debugger[3] to track the execution of b_c and obtain the last executed site, *i.e.*, the crash-site. Figure 5a shows the last three executed instructions of b_c, the last of which indicates the crash-site is at (line 10, offset 8). This means that executing the instruction compiled from (line 10, offset 8) in the source code results in a crash, *i.e.*, a sanitizer report.

Step 2. *Analyze b_n:* Once again, we use the debugger to track the execution of b_n. Since we have learned the crash-site in Step 1, we only need to monitor if the crash-site is also executed in b_n. Figure 5b shows a part of the execution in b_n. We can observe that (line 10, offset 8) is executed as well.

[3]In our implementation, we use LLDB and its python API to automate our analysis. More details in the evaluation section.

```
int a[5]; int x=1;          int a[5]; int x=1;
a[x] = 1;          ⟹       x = 5; //Δ(x);
                            a[x] = 1;
```

Figure 6. The expression $x = 5$ is inserted as the shadow statement to introduce a buffer overflow in $a[x]$.

Step 3. *Mapping:* Since the crash-site from b_c is also executed in b_n, we classify this program as triggering a sanitizer FN bug. Otherwise, the discrepancy would be classified as being caused by compiler optimizations.

For the program shown in Figure 3, the crash site from GCC ASan-O0 is not present in GCC ASan-O2, which indicates that the inconsistent sanitizer reports are caused by compiler optimizations. Our evaluation in Section 4.4 will demonstrate that crash-site mapping can accurately identify discrepancies resulting from compiler optimizations.

3 Approach

We first analyze sufficient conditions for a valid program to be free from UB, which motivates the design of our shadow statement insertion method. Then, we introduce the proposed UB program generation approach. Finally, we present the crash-site mapping as the test oracle for sanitizer testing.

3.1 UB Conditions and Shadow Statement

Code constructs that are free of UB. To generate UB programs, we need to first understand how UB is triggered. The first three columns in Table 1 list the conditions for certain code constructs to *not have* the UB, as specified in the C standard [2]. For instance, the first shown UB is buffer-overflow. For an array access $a[x]$ to be free from this UB, the index x should be positive and less than the array size. Another example is signed integer overflow. As long as the calculation of x `op` y falls within the range of $[-2^n, 2^n - 1]$, it is free from this UB. We can conclude that for a code construct that has the adventure of a UB, as long as the given condition is met, it is free from the UB.

Code constructs that have UB. Since we now understand the conditions for having UBs in certain code constructs, to introduce a UB, we can simply find a way to break the condition. In this paper, we utilize *shadow statements* to achieve this purpose. For a valid program $\mathcal{P}$ that contains a code construct *expr*, we introduce a UB by placing a shadow statement $\Delta(expr)$ before the code construct as follows:

```
Δ(expr);
Stmt{expr};
```

The shadow statement $\Delta(expr)$ is designed to change the evaluation value of *expr* such that when executing `Stmt{`*expr*`}`, the UB condition is triggered. The fourth and fifth columns in Table 1 list the shadow statements and their effects. For example, to introduce a buffer overflow to $a[x]$, the inserted shadow statement $\Delta(x)$ changes the value of x to v, which is out of the range of array a. Figure 6 illustrates a concrete example where the shadow statement $x = 5$ is inserted before the array access.

There are two key questions. The first is how to understand the target effect of `Stmt{`*expr*`}`. In the above example, we need to know the concrete range of array a, which our generator uses dynamic analysis to obtain. The second is how to instantiate the shadow statement. Once we know the target effect of `Stmt{`*expr*`}`, there are plenty of ways to instantiate it. In the above example, we can also choose $x = x + 4$ or $x = x * 4 + 1$ as the shadow statement, which results in the same effect as $x = 5$. The last column in Table 1 lists the instantiations we used in our implementation. Details will be discussed next.

Algorithm 1: UB program generation

```
1 procedure Generator(Program 𝒫, Input ℐ, UBType 𝒰):
      // find all matched expr to a given UB
2     E ← GetMatchedExpr(𝒫, 𝒰)
3     prof̂ ← Profile(𝒫, ℐ, 𝒰, E)        // profiling
4     P_UB ← [ ]
5     foreach expr ∈ E do
          // synthesize a shadow statement
6         Δ(expr) ← SynShadowStmt(expr, prof̂, 𝒰)
          // insert the shadow statement
7         𝒫' ← Insert(𝒫, Δ(expr) )
          // append the new UB program
8         P_UB.append(𝒫')
9     return P_UB
```

3.2 UB Program Generator

Algorithm 1 shows the general process of generating UB programs. Given a seed program $\mathcal{P}$ and an associated input $\mathcal{I}$, our goal is to generate UB programs that contain the target UB type $\mathcal{U}$ on the input $\mathcal{I}$. Our generator works as follows:

Step 1. *Expression Matching* (line 2): find all expressions in $\mathcal{P}$ that have the target code constructs for the given UB. For example, given buffer overflow, according to Table 1, this procedure will find all array accesses *i.e.*, $a[x]$, and pointer dereferences *i.e.*, $*p$.

Step 2. *Program Profiling* (line 3): instrument and run $\mathcal{P}$ on the input $\mathcal{I}$ to collect an execution profile that contains the required runtime information such as the allocated buffers and pointer addresses.

Step 3. *Shadow statement synthesis and insertion* (lines 6-7): for each target *expr*, query the execution profile to synthesize a shadow statement, and insert it into the seed program to obtain a UB program.

As indicated by the algorithm, our generator has the following features:

- *Target UB type needs to be specified when being invoked.* For every invocation, our generator will generate a set of UB programs that all have the same target UB type.
- *Only one UB in every generated program.* Lines 6-8 in Algorithm 1 show that for every matched expression, the generator generates a UB program with shadow statement insertion. Consequently, for each generated program, there is a single UB.
- *Multiple UB programs for one invocation.* The for loop (line 5) signifies that a UB program is generated for each of the matched expressions. Ultimately, the generator returns a set of UB programs, all containing the same UB type.

Next, we detail each of the above steps.

3.2.1 Expression Matching — GetMatchedExpr(·)

Given a seed program and a target UB, we statically scan the program to find all expressions that match the code constructs as specified in Table 1. For example, suppose our target UB is signed integer overflow, we will find all expressions that have the form of x op y, where op is an arithmetic operator such as +, −, and ∗. After scanning, all matched expressions will be saved into E, each item in which contains the matched expression and its location in $\mathcal{P}$.

3.2.2 Program Profiling — Profile(·)

An execution profile provides runtime information about the program, which is essential for our shadow statement synthesis. We define the execution profile $\widehat{prof}$ as follows.

Definition 1 (Execution Profile)**.** *Given a program $\mathcal{P}$, an input $\mathcal{I}$, and the target expression list E, the execution profile $\widehat{prof}$ records the following information during running $\mathcal{P}$ with $\mathcal{I}$: (1) all the values of expressions in E observed, and (2) all the allocated and freed stack and heap memory address ranges.*

To facilitate easy access to the execution profile, let e denote *expr*, we define the following queries to obtain concrete information from $\widehat{prof}$:

- $Q_{liv}(\widehat{prof}, e)$: return `true` if e is in the live region; otherwise, return `false`. This information is inferred by checking if e has a value in $\widehat{prof}$. If it does, then it is located in the live region; otherwise $\widehat{prof}$ is unable to obtain its value.
- $Q_{val}(\widehat{prof}, e)$: return the value of e.
- $Q_{mem}(\widehat{prof}, e)$: e is a pointer or an array. Return the memory range that e points to. If the memory has already been freed, return `false`.
- $Q_{scp}(\widehat{prof}, e)$: return the scope of e. We extend $\widehat{prof}$ with scope information obtained from Clang's LibTooling.

In our implementation, given a new seed program, we first obtain its execution profile $\widehat{prof}$ and then synthesize UB programs. Thus, the profiling overhead for all UB types is identical. This is an implementation choice because when testing sanitizers, UBFuzz by default generates all the supported UB programs for one seed program.

3.2.3 Shadow Statement Synthesis and Insertion — SynShadowStmt(·) & Insert(·)

For a target UB, the synthesized shadow statement should have the effect as shown in the fifth column in Table 1. In theory, there are numerous ways to instantiate a shadow statement. For instance, as previously illustrated in Figure 6, expressions like $x = 5$, $x = x + 4$, $x = x * 4 + 1$, and many others, all satisfy the requirement. In our implementation, for each shadow statement, we choose the simplest instantiation to minimize changes to the seed program. The last column in Table 1 lists the instantiations. Details are as follows:

- *Buffer overflow (array)*: We introduce an auxilary variable $\widehat{x}$ to the original expression to obtain $a[x + \hat{x}]$. $\Delta(expr)$ is $\widehat{x} = v - x$. The value of v is obtained by calculating the memory size from $Q_{mem}(\widehat{prof}, a)$; the value of x is obtained via $Q_{val}(\widehat{prof}, x)$. This instantiation does not change any other program semantics except for our target expression.
- *Buffer overflow (pointer)*: Similarly, we first introduce an auxilary variable $\widehat{x}$ to obtain $*(p + \widehat{x})$. $\Delta(expr)$ is $\widehat{x} = q - p$, where values of q and p are obtained via $Q_{mem}(\widehat{prof}, p)$ and $Q_{val}(\widehat{prof}, p)$, respectively.
- *Use after free*: $\Delta(expr)$ is $free(p)$.
- *Use after scope*: $\Delta(expr)$ is $p = q$, where $Q_{scp}(\widehat{prof}, *q)$ is not within the scope of $Q_{scp}(\widehat{prof}, p)$.
- *Null pointer dereference*: $\Delta(expr)$ is $p = (void*)0$.
- *Integer overflow*: We first introduce auxiliary variables $\widehat{x}$ and $\widehat{y}$ to the original expression to obtain $(x + \widehat{x})$ op $(y + \widehat{y})$. Then the shadow statement is set to $\widehat{x} = v_0 - x, \widehat{y} = v_1 - y$. The values of x and y are obtained via $Q_{val}(\widehat{prof}, x)$ and $Q_{val}(\widehat{prof}, y)$. To find the proper values v_0 and v_1, we adopt Monte Carlo to sample from $[-2^n, 2^n - 1]$ such that $(x + \widehat{x})$ op $(y + \widehat{y})$ exceeds the range of an $(n + 1)$-bit integer.
- *Shift overflow*: We first introduce an auxiliary variable $\widehat{y}$ to obtain $x \ll (y + \widehat{y})$ or $x \gg (y + \widehat{y})$, and then set the shadow statement to $\widehat{y} = v - y$. The value of y

is obtained via $Q_{val}(\widehat{prof}, y)$ and v is a random value satisfying $v < 0 \vee \geq n$.

- *Divide by zero*: We first introduce an auxiliary variable $\widehat{y}$ to obtain $x/(y+\widehat{y})$, and then set the shadow statement to $\widehat{y} = -y$. The value of y is obtained via $Q_{val}(\widehat{prof}, y)$.
- *Use of uninitialized memory*: We first introduce an auxiliary variable $\widehat{x}$ to obtain $x + \widehat{x}$, and then set $\Delta(expr)$ to `int` $\widehat{x}$. Since $\widehat{x}$ is uninitialized, $x + \widehat{x}$ becomes uninitialized as well.

Some of the above operations, such as *Buffer overflow (pointer)*, need to know the precise pointer information to accurately synthesize shadow statements. Such pointer information can be obtained via $Q_{mem}(\widehat{prof}, e)$, which achieves this goal by logging all allocated pointers' addresses and used pointers (see the example in § 2.1 **Step 1**). Thus, we do not need any separate pointer analysis.

3.2.4 Discussions

As the evaluation will demonstrate, our generator can effectively generate interesting UB programs for sanitizer testing. Despite its effectiveness, it does also come with certain limitations. First, our UB program generator relies on seed programs. If seed programs are not expressive enough, UBFuzz cannot generate useful UBs. Fortunately, seed program generators like Csmith have proven effective in exercising rich language features [12, 16]. Second, the list of UB in UBFuzz is non-exhaustive. The C17 standard [2] lists 219 UB types. Not all UBs are supported by sanitizers. We selected UBs that are (1) supported by at least one sanitizer and (2) included in the CWE list [34] which enumerates all common weaknesses in C by the MITRE community. Our supported UBs cover all UBs studied by the related work [10, 41]. Generally, each UB comes with a root cause and can be represented in a generic pattern, as demonstrated in our approach. For instance, using pointer subtraction to determine size is UB if two pointers point to different objects [27]. Realizing this UB in UBFuzz would require knowledge of the address ranges of each object and pointer, which can be easily obtained through dynamic profiling. We chose not to realize this UB because none of the existing sanitizers support its detection.

For each seed program, as a new UB is introduced into it, its semantics is consequently altered. We clarify that preserving the seed program's semantics is not necessary in our application scenario because we only require the resulting program to contain the desired UB. Second, all UB programs have invalid, often nondeterministic semantics because (1) their semantics rely on how the compiler deals with UBs, and (2) the compiler has full freedom in handling code with UBs. Nevertheless, UBFuzz still preserves the runtime semantics of a seed program up to the mutation site.

Algorithm 2: Crash-Site Mapping

```
1  procedure IsBug(Binary b_c, Binary b_n):
2      S_c ← GetExecutedSites(b_c)
3      S_n ← GetExecutedSites(b_n)
4      if S_c[−1] ∈ S_n then
5          return True
6      else
7          return False
8  procedure GetExecutedSites(Binary b):
9      S ← [ ]
10     debugger.Init(b)
11     while debugger.IsAlive() do
12         l ← debugger.curr_line
13         o ← debugger.curr_offset
14         S.append((l, o))
15         debugger.NextInstruction()
16     return S
```

3.3 Crash-site Mapping as the Test Oracle

With the generated UB programs, we employ differential testing across multiple compilers to find sanitizer FN bugs. Without loss of generality, assume that we have two compilers $\mathcal{C}_c$ and $\mathcal{C}_n$ with the same sanitizer enabled, *e.g.*, GCC ASan at -O1 and LLVM ASan at -O1. The corresponding compiled binaries are b_c and b_n. Suppose that executing b_c results in a crash while b_n exits normally. Here, the crash in b_c means that the sanitizer in $\mathcal{C}_c$ *successfully* reports the UB; the normal exits of b_n means that the sanitizer in $\mathcal{C}_n$ *does not* report any UB. As analyzed in Section 1, the discrepancy can arise from a sanitizer FN bug or merely compiler optimizations. Our *crash-site mapping* can identify the true cause of the discrepancy. Before introducing our approach, we formally define *crash site*.

Definition 2 (Crash Site)**.** *A binary b_i is compiled from program $\mathcal{P}$ and running b_i results in a crash. We denote the last executed instruction as $\widehat{inst}$. If $\widehat{inst}$ corresponds to the line l and offset o in $\mathcal{P}$, then the crash site of b_i is (l, o).*

Our key insight is that if the crash site in b_c is also executed by b_n, the compiler $\mathcal{C}_n$ does not optimize away the UB-triggering expression in $\mathcal{P}$, thus the discrepancy is caused by a sanitizer FN bug in $\mathcal{C}_n$. Algorithm 2 details our approach.

We first obtain the executed sites of both b_c and b_n, *i.e.*, all the executed (line, offset) in $\mathcal{P}$ (line 2-3). If the last executed site in b_c, *i.e.*, the crash site, is also present in b_n's executed sites, return true (line 4-5). Otherwise, return false (line 7). To obtain all executed sites in a binary, we utilize a debugger to track the execution. The procedure GetExecutedSites() provides the necessary steps. Note that when

Table 2. UB types supported by each sanitizer.

UB	Sanitizer
Buf. Overflow(Array)	ASan, UBSan
Buf. Overflow(Pointer)	ASan
Use After Free	ASan
Use After Scope	ASan
Null Ptr. Deref.	UBSan
Integer Overflow	UBSan
Shift Overflow	UBSan
Divide by Zero	UBSan
Use of Uninit. Memory	MSan

the debugger reaches an instruction, the *debugger.curr_line* and *debugger.curr_offset* return the line and offset in the source program that the instruction corresponds to. The effectiveness of crash-site mapping depends on its accuracy in identifying discrepancies caused by compiler optimizations. Our evaluation in Section 4.4 will show that it can achieve near-perfect accuracy.

4 Empirical Evaluation

Our evaluation is based on the following research questions:

RQ1 Bug-finding: Is UBfuzz effective in finding FN bugs in sanitizers?

RQ2 UB generator: How effective is our UB program generator in constructing interesting UB programs?

RQ3 Crash-site mapping: How accurate is the crash-site mapping test oracle in identifying discrepancies caused by compiler optimizations?

RQ4 Code coverage: Can UBfuzz improve code coverage?

4.1 Implementation and Evaluation Setup

Implementation. Our realization of UBfuzz consists of ~2,000 lines of C++ and ~4,400 lines of Python. We use Clang's LibTooling [35] to implement expression matching in Section 3.2.1 and program instrumentation for execution profiling in Section 3.2.2. We utilize LLDB [36] as the debugger in crash-site mapping and use its Python API to automate the analysis process. Our UBfuzz can run in a fully automated manner in testing sanitizers, including UB program generation, crash-site mapping, and debugging procedures. Once launched, our tool will automatically generate UB programs and use the crash-site mapping algorithm to find FN bugs.

Compilers and sanitizers. Sanitizers are integrated into compilers. We used UBfuzz to test the latest development versions of both GCC and LLVM, which support the most widely-used sanitizers, namely ASan, UBSan, and MSan. Note that, MSan is not yet supported by GCC. Since sanitizers are typically used with optimizations, we enabled the most frequently used optimization levels, namely -O0, -O1, -Os, -O2, and -O3, in both compilers for differential testing.

Seed programs. We use Csmith [42] — a random C program generator — to produce valid seed programs. There are three main reasons:

(1) Csmith is adopted by a lot of compiler testing work [12, 37, 39] and has become the de facto default program generator in testing C compilers;

(2) Csmith can generate complex programs with rich features (*e.g.*, pointer and integer operations), thus offering UBfuzz abundant opportunities to generate diverse UB programs; and

(3) programs generated by Csmith are self-contained meaning that they do not take inputs and can be executed.

Hardware. We conducted all our evaluations on two Linux servers running Ubuntu 20.04 LTS. Both are equipped with an AMD EPYC 7742 64-Core CPU and 256GB RAM.

Testing process. Our testing process is fully automated and runs continuously. We first use Csmith to generate a well-formed seed program. Then, for each of the supported UB, we apply UBfuzz to generate UB programs from the seed. For each of the UB programs, as we know their UB type, we use compilers with the corresponding sanitizer enabled to compile and run it. Table 2 lists the supported sanitizers for each UB. Once a discrepancy is found, we apply crash-site mapping to decide if it is a sanitizer FN bug. If so, we use C-Reduce to reduce the UB program and report the reduced program to the respective bug tracker. During a period of five months, we sporadically tested the sanitizers. UBfuzz generated around 130 million UB programs. Note that, since our work focuses on in-house testing, we assume no adversary is present. Thus, successful sanitization always results in a crash.

4.2 RQ1: Bug Finding

Table 3 summarizes the sanitizer bugs we discovered during our testing period. Overall, we reported 31 bugs. The developers have confirmed 20 of them as previously unknown, real bugs. This highlights the significant bug-finding capability of UBfuzz. Of all these bugs, 6 of them have been fixed and all the fixed bugs are in GCC. The relatively high number of unfixed bugs could be attributed to the fact that many of the reported bugs are introduced since the launch of sanitizers and affect *all stable compiler versions*. Our later analysis will show this fact. We also experienced that the LLVM developers were less responsive than GCC and mostly only labeled our reports as sanitizer bugs without further diagnosis. We are strict in marking a bug as confirmed — only if the developers have clearly diagnosed it and responded to us. This causes although UBfuzz found nearly the same number of bugs in GCC and LLVM, most confirmed and all fixed bugs are found in GCC.

Figure 7 shows the number of bugs triggered by each kind of UB. Since both ASan and UBSan support the detection of

Table 3. Status of the reported bugs in GCC and LLVM.

Status	GCC		LLVM			Total
	ASan	UBSan	ASan	UBSan	MSan	
Reported	9	7	6	8	1	31
Confirmed	8	7	2	2	1	20
Fixed	3	3	0	0	0	6
Invalid	1	0	0	0	0	1

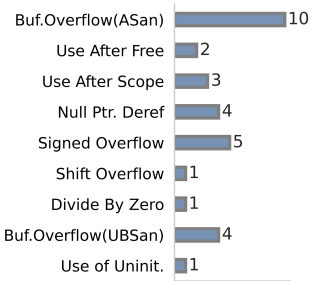

Figure 7. Number of bugs triggered by each kind of UB.

```
1 int a, b;
2 int main() {
3  int *s = &a;
4  for(b=0;b<=3;b++){
5   int i = *s;
6   s = &i;
7  }
8  *s = b;
9 }
```

Figure 8. A use-after-scope UB at line 8.

buffer overflow, we split the found bugs into BufOverflow (ASan) and BufOverflow (UBSan). We can observe that buffer overflow programs triggered the most number of bugs in ASan. Notably, UBfuzz detected bugs in all UB types, which highlights its strong bug detection capability and the importance of extensively testing sanitizers. Of the 31 bugs, 29 are sanitizer FN bugs, meaning that sanitizers failed to detect UBs in them. Interestingly, we also found 2 bugs that are not sanitizer FN bugs but rather wrong reports, which means that sanitizers report a UB but with incorrect report information such as a wrong UB type warning.

➤ **Are there any false alarms by UBfuzz?** We encountered one false alarm report generated by UBfuzz as indicated by the "Invalid" row in Table 3. The reported program is shown in Figure 8. It contains a use-after-scope at line 8 because `s` points to an inner scope variable `i`. GCC ASan at -O3 can not detect it. Our crash-site mapping can verify that line 8 is still present at -O3. The GCC developers marked this report as invalid because GCC -O3 removes the for loop and moves out the inner code, which invalidates the use-after-scope UB. This program reveals a limitation of our crash-site mapping test oracle. Nevertheless, the significant number of reported true bugs already demonstrates its effectiveness.

➤ **How significant are the bug-finding results?** To approach this question, we have conducted a manual analysis of all reported false negative bugs based on GCC and LLVM bug trackers of sanitizers. We choose GCC-5 (released in 2015) and LLVM-5 (released in 2017) as the earliest versions because they are the first stable versions that support sanitizers. The results are shown in Figure 9. In the past decade,

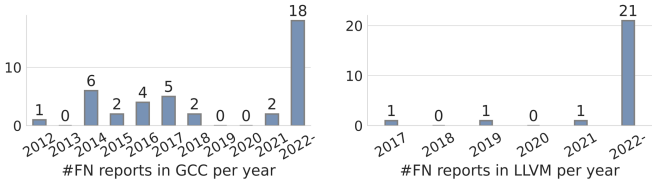

Figure 9. Number of sanitizer FN bug reports in GCC and LLVM bug trackers per year.

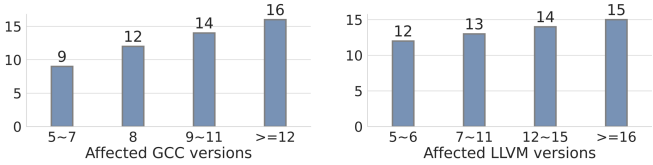

Figure 10. Stable compiler versions that are affected by the reported sanitizer FN bugs.

there were a total of 40 false negative reports on GCC's sanitizers. Of these 40 bugs, UBfuzz found 16 (40%). For LLVM, UBfuzz found 14 (58%) out of the 24 bugs. As an intermediate conclusion, UBfuzz has found a significant number of interesting bugs in both GCC's and LLVM's sanitizers. To further understand the influence of our reported bugs in different stable releases of compilers, we also ran the UB programs that accompany our bug reports on all stable compiler versions. Figure 10 presents the number of sanitizer bugs that affect each stable compiler version. It indicates that UBfuzz can find many long-standing latent bugs, further confirming the significance of our bug-finding results.

➤ **Affected optimization levels.** As shown in Figure 11, we counted the number of bugs that affect each optimization level. The result reveals that sanitizer bugs affect all optimization levels. Testing only one of the optimization levels such as -O0 would miss many bugs that only appear at other optimization levels. This demonstrates the usefulness of our crash-site mapping test oracle in identifying sanitizer bugs across optimization levels. There is no clear tendency on which optimization levels are more sensitive to sanitizer bugs. It correlates to the fact that sanitizers and compiler optimizations work independently as having been shown in Section 1.

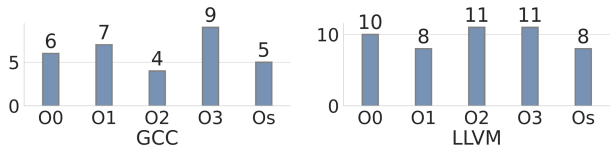

Figure 11. Affected optimization levels

Table 4. The number of generated UB programs per generator. The "No UB" column shows the number of generated programs that do not contain UB. UBfuzz having "-" on this attribute means that all of its generated programs contain UB.

Generator	UB										No UB
	Buf.Overflow (Pointer)	Use After Free	Use After Scope	Null Ptr. Deref	Integer Overflow	Shift Overflow	Divide by Zero	Buf.Overflow (Array)	Use of Uninit.	Total	
UBfuzz	4,213	3,032	461	2,082	408	287	329	2,396	664	13,872	-
MUSIC	27	0	0	1	151	487	3	26	9	704	13,296
Csmith-NoSafe	0	0	0	0	220	5,286	1,899	0	0	7,405	6,595

4.3 RQ2: Effectiveness of UB Program Generator

This section provides an in-depth understanding of the effectiveness of our UB program generator. Although there is no other UB program generator that we could compare UBfuzz against, we use the following two generators as the baseline:

- *MUSIC* [28] is a program mutator designed for mutation testing. It mutates a valid program's abstract syntax tree (AST) to generate syntactically valid mutants. By design, MUSIC may also generate UB programs as it has no guarantees regarding program semantics.
- *Csmith-NoSafe* means that one runs Csmith with its `-no-safe-math` option. To avoid UB at runtime, Csmith utilizes many safe wrappers. For example, it changes all `x/y` to `(y==0?1:x/y)` to avoid division-by-zero. We use its `-no-safe-math` option to disable all the safe wrappers, which may introduce UB in the generated programs.

For each generator, we assess the quantity of each type of UB program that the respective generator can produce. We also equip the two baseline generators with the crash-site mapping oracle to test sanitizers.

Generation quantity. We first use Csmith to randomly generate 1,000 seed programs. For each seed program, we use our generator to generate UB programs for every UB type that we support. Table 4 details the results. The column "Total" shows that out of the 1,000 seed programs, UBfuzz generates 13,872 UB programs, averaging 14 UB programs per seed. The generated programs cover all UB types that we support. Buffer overflow takes up the most generated UB programs. The reason is that the seeds from Csmith contain a large number of array and pointer operations, on which UBfuzz can generate buffer overflow programs. Relatively fewer UB programs are generated on some of the UB types such as UseAfterScope and DividebyZero. The main reason is that the code constructs required by them are more strict than others. For example, DividebyZero can only happen if operators "/" or "%" are present in the live code regions. Comparatively, NullPtrDeref requires only a pointer dereference such as "$*p$", which apparently appears more often.

For a fair comparison to UBfuzz, we apply MUSIC to randomly generate 14,000 programs from the 1,000 seeds used by UBfuzz. Then, we utilize sanitizers[4] to compile and analyze these programs to know if each of them contains UB. Table 4 shows that there are only 704 (4%) out of the 14,000 programs containing UB. The other 13,296 (95%) do not contain UB. We now use Csmith-NoSafe to generate programs. Because Csmith-NoSafe does not require a seed program, we directly use it to generate 14,000 programs. Similarly, we use sanitizers to analyze if each of the programs contains UB. From the last row in Table 4, we can find that around half (7,405) of the programs contain UB. This number is not as high as UBfuzz but already much better than MUSIC. Notably, all the UB programs are only in three types, *i.e.*, IntegerOverflow, ShiftOverflow, and DividebyZero. This is consistent with how Csmith-NoSafe work: it removes safe wrappers around numeric operations. In summary, UBfuzz can generate the most number of UB programs and cover the most types of UB. Next, we will use MUSIC and Csmith-NoSafe as the UB generator to extensively test sanitizers.

Testing sanitizers with MUSIC and Csmith-NoSafe. To understand if UB programs produced by the baseline generators can also find sanitizer FN bugs, we replace the generator component in UBfuzz with MUSIC and Csmith-NoSafe. The crash-site mapping remains unchanged to serve as the test oracle. During our testing, we let each generator generate around 1 million programs. In the end, *we did not find any sanitizer FN bugs.* The failure reason for MUSIC could be that most of the generated programs did not exercise UB. For Csmith-NoSafe, its failure is mainly due to (1) the narrow range of UB types it can generate, and (2) unlike our generator, it typically introduces multiple UB in a program, which makes it hard to discover missed sanitizer reports.

Testing sanitizers with the existing UB test suite. The Juliet test suite [26] released by NIST consists of a collection of UB programs. It is by far the most comprehensive test suite for UB detectors. To understand if UB programs from the existing test suite can find sanitizer bugs, we select all the 16,344 UB programs from the Juliet test suite that

[4]We run each program with all sanitizers. If a sanitizer reports UB on a program, we use its report to get its UB type. Note that, the programs generated by UBfuzz do not need such analysis because the design of UBfuzz allows us to know the UB type of each generated program.

are detectable by sanitizers. Instead of using a generator, we directly use all the UB programs from the test suite as the source of programs. Our results show that *none of the UB programs from the Juliet test suite can find sanitizer FN bugs*. This further confirms the necessity of a UB program generator like ours.

4.4 RQ3: Effectiveness of Crash-Site Mapping

For each generated UB program, we apply differential testing to find discrepancies across compilers. We then use our crash-site mapping to determine if a discrepancy is caused by a sanitizer FN bug or merely compiler optimizations. For the 13,872 UB programs generated from Section 4.3, we run all the sanitizers specified in the evaluation setup (Section 4.1) to select programs that cause discrepant sanitizer reports. This results in a total of 6,567 selected programs, nearly half of the generated UB programs. The substantial number of discrepancy-causing programs highlights (1) the exceptional quality of our generated UB programs, and (2) without our crash-site mapping, discerning real sanitizer bug-caused discrepancies from the 6,567 discrepancies would be practically infeasible. To evaluate the effectiveness of our crash-site mapping test oracle, we measure its *precision* and *recall*.

Precision: *Out of all selected discrepancies, how many are truly caused by sanitizer bugs?* Out of the **6,567** discrepancies, our crash-site mapping selected **58** and dropped the rest **6,505** as invalid. For each of the selected discrepancies, we manually verify if it is caused by compiler optimizations. *Our manual analysis found that all discrepancies selected by crash-site mapping are due to sanitizer bugs, which means that our crash-site mapping achieves perfect precision.* Although we have analyzed an invalid report by UBfuzz in Section 4.2, it does not appear in our quantitative evaluation. Thus, we may conclude that our crash-site mapping has a high precision.

Recall: *Out of all sanitizer bug-caused discrepancies, how many are selected?* This measures if our crash-site mapping will miss interesting discrepancies. Ideally, we should analyze all the dropped discrepancies to verify if any of them are due to sanitizer bugs. However, this requires a manual analysis of 7,966 discrepancies. To reduce the cost, we randomly sampled 200 dropped discrepancies by the crash-site mapping and then manually analyzed each of them. Perhaps surprisingly, after our analysis, we found that none of the dropped discrepancies were caused by sanitizer bugs. In other words, *our crash-site mapping achieves 100% recall* on these samples. Since our evaluation is on sampled data, it is not complete, but it does suggest that crash-site mapping has a high recall.

Soundness of Crash-Site Mapping: As defined in Definition 2, the *crash site* is associated with the source location of the last executed instruction. The soundness of *crash-site mapping* largely depends on a reliable mapping between instructions and source locations. In our implementation, we enable -g option for all compilations, which enriches the produced binaries with debugging meta-data. These meta-data can then be utilized by a debugger to obtain the source location, *i.e., (line number, offset)*, of each instruction. Although compiler optimizations can remove instructions with their meta-data, it will not cause the soundness problem in *crash-site mapping* because this resides in the scope discussed in Challenge 2 in §1. Unfortunately, a recent study [37] has confirmed that bugs in compilers may lead to incorrect debugging meta-data. Buggy meta-data can theoretically cause unsound or incorrect *crash-site mapping* results. For instance, *crash-site mapping* can incorrectly flag the existence of an eliminated crash-site, and thus generate false positive reports. During our extensive testing period, we did not observe any false positive reports though. We believe such compiler bugs to be rare in our testing scenario. Handling buggy debugging meta-data is an orthogonal research program and we assume always correct meta-data in this work.

Table 5. Line coverage (LC), function coverage (FC), and branch coverage (BC) of GCC and LLVM.

	GCC			LLVM		
	LC	FC	BC	LC	FC	BC
Seeds	63.1%	65.5%	49.4%	30.4%	38.2%	23.3%
MUSIC	63.1%	65.5%	49.4%	30.5%	38.2%	23.4%
Csmith-NoSafe	63.6%	65.5%	50.1%	32.5%	40.2%	24.8%
UBfuzz	63.7%	65.5%	50.8%	31.8%	39.3%	24.3%

Table 6. Bug category according to root cause analysis.

Category	GCC	LLVM
No Sanitizer Check	2	2
Incorrect Sanitizer Optimization	5	3
Wrong Red-Zone Buffer	1	1
Incorrect Sanitizer Check	2	7
Incorrect Expression Folding/Shorten	4	1
Incorrect Operation Handling	0	1
Wrong Line Information	2	0

4.5 RQ4: Code Coverage

We utilized Gcov and only instrumented sanitizer-related files to collect coverage in both GCC and LLVM. We used the generated programs from Section 4.3 to profile coverage. Table 5 summarizes our results. In all cases, compared to the seed programs, all generators lead to a moderate coverage improvement, with UBfuzz and Csmith-NoSafe showing the largest increase on GCC and LLVM, respectively.

```c
1 int g, *ptr = &g;
2 int **p_ptr = &ptr;
3 int main() {
4   int buf[3]={1,2,3};
5   *ptr = 1;
6   *p_ptr =&buf[3];
7   *ptr = 0xfff;
8 }
```

(a) GCC ASan at -O1 missed the buffer overflow access `*ptr` at line 7. [7]

```c
1 int a, c;
2 short b;
3 long d;
4 int main() {
5     a = (short)(d == c |
6         b > 9) / 0;
7     return a;
8 }
```

(b) GCC's UBSan at all levels missed the division-by-zero at line 5. [9]

```c
1 void b() {
2   int c[1];
3   c;
4 }
5 int main() {
6   int d[1]={1};
7   int *e = d;
8   a = 0;
9   for(;a<=5;++a){
10    int f[1]={};
11    e = f;
12    a||(b(), 1);
13  }
14  return *e;
15 }
```

(c) GCC's ASan missed the use after scope at line 14, where the pointer `e` points to an inner scope variable `f` defined at line 10. [8]

```c
1 volatile int a[5];
2 void b(int x) {
3     if(x)
4       a[5] = 7;
5 }
6 int main(){ b(1); }
```

(d) LLVM's ASan missed the buffer overflow at line 4. [19]

```c
1 int main() {
2     int *a = 0;
3     int b[3]={1, 1, 1};
4     ++b[2];
5     ++(*a);
6 }
```

(e) LLVM's UBSan missed the null pointer dereference at line 5. [20]

```c
1 int main() {
2     unsigned char a;
3     if (a-1)
4         __builtin_printf("boom!\n");
5     return 1;
6 }
```

(f) LLVM's MSan missed the use of uninitialized memory at line 3. [21]

Figure 12. Sample UB programs that trigger sanitizer FN bugs.

4.6 Case Study

In order to understand the reason why sanitizers make mistakes, we categorize all bugs according to their root causes. The categorization is based on both our manual analysis and developers' feedback. Table 6 shows the result. Both GCC and LLVM make some common mistakes. For example, their sanitizer implementations may conduct "Incorrect Sanitizer Optimization" causing valid sanitizer checks to be removed. We discuss a selection of representative bugs in each bug category.

Figure 12a: (*No Sanitizer Check*) This program contains an overflowed memory access at line 7. Since `p_ptr` initially points to pointer `ptr` at line 2, `ptr` will point to the overflowed address `&buf[3]` after line 6. Therefore, a stack-buffer-overflow occurs at line 7, and then the value `0xfff` is written to `buf[3]`. However, due to a sanitizer instrumentation bug, GCC ASan at -O2 fails to insert the check for the validity of `*ptr` at line 7, and thus cannot report it. This bug affects GCC trunk and has been fixed.

Figure 12b: (*Incorrect Expression Folding/Shorten*) This program reveals a long latent bug in GCC UBSan, which fails to report the DivisionbyZero UB at line 6. The root cause is that UBSan only cares about integer operands, but not booleans. However, although `(d==c|b>9)` is boolean, it gets widened to `short`. GCC UBSan incorrectly handles this case and thus misses the UB. This bug exists since the introduction of UBSan in GCC.

Figure 12c: (*Incorrect Sanitizer Optimization*) This program contains a UseAfterScope UB at line 14, where `e` points to an inner scope variable `f`. GCC ASan fails to report this bug at -O3. In fact, GCC ASan initially indeed inserts a scope check for `f` at line 10, but another sanitizer analysis module removes this check when exiting the loop.

Figure 12d: (*Wrong Red-Zone Buffer*) This program has an overflowed array access at line 4, where the array `a` is of length 5. LLVM ASan incorrectly marks the overflow access as within the scope of array padding while in fact, it is not. This bug reveals a fundamental problem with ASan handling of global arrays. It affects all LLVM versions at all optimization levels.

Figure 12e: (*Incorrect Sanitizer Check*) This program contains a NullPointerDereference UB at line 5, where the pointer `a` is NULL and the program tries to increment it. LLVM UBSan does not report this bug because the null pointer check is not placed before the increment operation. The developer believes that the `++` operator misleads UBSan's internal logic because if we replace `++(*a)` with `*a += 1`, UBSan would work again.

Figure 12f: (*Incorrect Operation Handling*) The `if` branch in this program can be taken differently depending on the value of uninitialized variable `a`. LLVM MSan incorrectly handles the subtraction and thinks that the value of `(a-1)` is fully determined. The LLVM developers have confirmed this bug and are working on a fix.

4.7 Discussion on Approach Generality

Despite sanitizers are the most popular UB detectors, there are many other dynamic and static UB detection tools. Dynamic tools such as Dr. Memory [1] and Valgrind [25] can detect memory errors including buffer overflows, use of uninitialized memory, improper free, *etc.* Static tools such as CppCheck [3] and Infer [23] can detect null pointer dereferences, integer overflows, *etc.* In principle, our approach can also be used to test these detectors. We currently focus on sanitizers because they have a wider real-world impact, especially in the area of fuzzing. Our evaluation results on testing sanitizers have already confirmed the significant UB program generation and bug-finding capability of our tool. Extending our testing scope to other detectors would be an interesting application of our approach and help solidify these additional tools.

5 Related Work

Compiler Testing. Finding compiler bugs has been extensively studied; significant research effort has been devoted to testing various compiler functionalities. Csmith [16] is the most popular program generator for C and has found hundreds of compiler crashes and correctness bugs. Csmith-generated programs are guaranteed to be free of undefined behavior. Instead of generation, Equivalent Modulo Input (EMI) [12] is proposed to mutate/transform a seed program while preserving its semantics under the same input. EMI can be implemented by deleting dead statements [12], inserting new code in dead regions [13], or synthesizing equivalent code in live regions [33]. Together with Csmith, EMI has found thousands of compiler optimization bugs. YARPGen [18] is another C/C++ program generator that aims to test scalar optimizations in compilers.

In addition to optimization correctness, other issues in compilers such as incorrect debug information [17, 39] and missed optimizations [37] have also been studied. Li *et al.* [17] construct the so-called actionable programs to validate the debug information generated for optimized code. Dfusor [39] transforms a seed program into multiple variants and then uses them to find debug information inconsistencies. Dead [37] injects markers into dead regions of a program to find missed dead code eliminations in optimizing compilers.

Sanitization. ASan and MSan use shadow memory to record and check the safety of each memory access. Runtime checks are inserted around memory accesses during the compilation of a program. Similarly, UBSan uses tailored checks for different UBs such as overflow checks for additions and null pointer checks for pointer dereferences. These checks will inevitably increase a program's runtime overhead. Many approaches have been proposed to reduce the overhead by removing redundant checks [43], optimizing checks [44], or applying checks to only a subset of the original code [14, 38]. These optimizations are meaningful in improving sanitizers' practical utility. UBfuzz can also be used to validate their implementations once they are integrated into mainstream compilers.

6 Conclusion

We have presented a novel framework for testing sanitizer implementations. We have introduced a UB program generator that generates UB programs from a seed program via shadow statement insertion. Based on this generator, we have employed differential testing across multiple compilers to test sanitizers. To filter out discrepancies caused by compiler optimizations, we have designed a new test oracle, crash-site mapping, that is capable of accurately identifying true sanitizer bugs. UBfuzz, our implementation of the testing framework, has discovered 31 bugs in ASan, UBSan, and MSan from both GCC and LLVM. Our work represents a promising, initial step toward comprehensive validations of sanitizer implementations, and highlights the importance of this problem.

7 Acknowledgments

We thank the anonymous ASPLOS reviewers for their valuable feedback. Our special thanks go to the GCC and LLVM developers for useful information and for addressing our bug reports. This work was partially supported by a Meta Security Research RFP award.

A Artifact Appendix

A.1 Abstract

The artifact contains the code and datasets we used for our experiments, as well as scripts to generate the numbers and tables of our evaluation. Specifically, it includes (a) links and bug-triggering test cases of each reported bug; (b) 1,000 Csmith seed programs used for evaluation; (c) scripts for generating UB programs with UBfuzz, MUSIC, and Csmith-NoSafe; (d) scripts for reporting coverage achieved by each apprach; and (e) detailed instruction documentation for using UBfuzz. Everything is packaged and pre-built as a docker image. A standard X86 Linux machine running docker is necessary to evaluate this artifact.

A.2 Artifact Check-List (Meta-Information)

- **Run-time environment:** Linux
- **Hardware:** X86
- **Output:** Statistics of CompDiff detection results on the Juliet testsuite and 23 real-world programs.
- **How much disk space required (approximately)?:** 40GB
- **How much time is needed to prepare workflow (approximately)?:** 10-20 minutes to download and import the docker image.
- **How much time is needed to complete experiments (approximately)?:** 20 hours

- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** Yes

A.3 Description

A.3.1 How to access

The artifact can be downloaded from the following link: https://doi.org/10.5281/zenodo.8406414

A.3.2 Hardware dependencies

A standard X86 machine.

A.3.3 Software dependencies

Docker

A.4 Installation

tar xf compdiff-asplos23-ae.tar.gz
cat compdiff-asplos23-image.tar | docker import - compdiff_ae

A.5 Experiment Workflow

1. Read the documentation.
2. Start the docker container as instructed.
3. Check bug reports.
4. Run UBfuzz, MUSIC, and Csmith-NoSafe to generate UB programs.
5. Collect coverage information for each approach.

A.6 Evaluation and Expected Results

We provide data and scripts to generate all the evaluation results in Section 4. Specifically, Tables 3, 4, and 5 are reproduced.

References

[1] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 2011 International Symposium on Code Generation and Optimization (CGO'11)*. IEEE, 213–223.

[2] 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.

[3] CppCheck developers. 2023. A Tool for Static C/C++ Code Analysis. Retrieved March 7, 2023 from http://cppcheck.sourceforge.net/

[4] Google developers. 2017. OSS-fuzz - continuous fuzzing of open source software. https://github.com/google/oss-fuzz. Accessed: March 7, 2023.

[5] LLVM developers. 2023. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: March 7, 2023.

[6] Zhen Yu Ding and Claire Le Goues. 2021. An empirical study of oss-fuzz bugs. In *Proceedings of the 2021 IEEE/ACM International Conference on Mining Software Repositories (MSR'21)*. IEEE, 131–142.

[7] GCC. 2022. Bug Report. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=106558.

[8] GCC. 2022. Bug report. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108085.

[9] GCC. 2023. Bug report. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109151.

[10] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik Van Der Kouwe, and Klaus von Gleissenthall. 2023. Don't Look UB: Exposing Sanitizer-Eliding Compiler Optimizations. In *To appear at the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'23)*. 1–21.

[11] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing.. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 249–263.

[12] 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'14)*. 216–226.

[13] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. 386–399.

[14] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. 2018. PartiSan: fast and flexible sanitization via run-time partitioning. In *Proceedings of the 21st International SymposiumResearch in Attacks, Intrusions, and Defenses (RAID'18)*. Springer, 403–422.

[15] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proceedings of the ACM on Programming Languages* 7, OOPSLA (2023), 1–27.

[16] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. 238–251.

[17] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. 1052–1065.

[18] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'20)* (2020), 1–25.

[19] LLVM. 2022. Bug report. https://github.com/llvm/llvm-project/issues/55189.

[20] LLVM. 2023. Bug report. https://github.com/llvm/llvm-project/issues/60236.

[21] LLVM. 2023. Bug report. https://github.com/llvm/llvm-project/issues/61982.

[22] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[23] Meta. 2023. A Tool to Detect Bugs in Java and C/C++/Objective-c Code. Retrieved March 7, 2023 from https://fbinfer.com/

[24] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. 351–365.

[25] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 89–100.

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

[27] NIST. 2023. CWE-469: Use of Pointer Subtraction to Determine Size. https://cwe.mitre.org/data/definitions/469.html. Accessed: March 7, 2023.

[28] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. 2018. Music: Mutation analysis tool with high configurability and extensibility. In *Proceedings of the 2018 IEEE international conference on software testing, verification and validation workshops (ICSTW'18)*. IEEE, 40–46.

[29] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *Proceedings of the 2016 IEEE Cybersecurity Development (SecDev'16)*. IEEE, 157–157.

[30] Kostya Serebryany. 2016. Sanitize, fuzz, and harden your C++ code.

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

[32] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. IEEE, 46–55.

[33] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. 849–863.

[34] CWE Content Team. 2008. CWE: Weaknesses in Software Written in C. https://cwe.mitre.org/data/definitions/658.html.

[35] The Clang Team. 2023. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

[36] The LLDB Team. 2023. The LLDB Debugger. https://lldb.llvm.org/.

[37] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. 697–709.

[38] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, 866–879.

[39] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. 146–158.

[40] 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.

[41] 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 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 260–275.

[42] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 283–294.

[43] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. 2021. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs.. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 479–494.

[44] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. 2022. Debloating Address Sanitizer. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security'22)*. 4345–4363.