Don't Look UB: Exposing Sanitizer-Eliding Compiler Optimizations

RAPHAEL ISEMANN, Vrije Universiteit Amsterdam, The Netherlands
CRISTIANO GIUFFRIDA, Vrije Universiteit Amsterdam, The Netherlands
HERBERT BOS, Vrije Universiteit Amsterdam, The Netherlands
ERIK VAN DER KOUWE, Vrije Universiteit Amsterdam, The Netherlands
KLAUS VON GLEISSENTHALL, Vrije Universiteit Amsterdam, The Netherlands

Sanitizers are widely used compiler features that detect undefined behavior and resulting vulnerabilities by injecting runtime checks into programs. For better performance, sanitizers are often used in conjunction with optimization passes. But doing so combines two compiler features with conflicting objectives. While sanitizers want to expose undefined behavior, optimizers often exploit these same properties for performance. In this paper, we show that this clash can have serious consequences: optimizations can remove sanitizer failures, thereby hiding the presence of bugs or even introducing new ones.

We present LookUB, a differential-testing based framework for finding optimizer transformations that elide sanitizer failures. We used our method to find 17 such *sanitizer-eliding optimizations* in Clang. Next, we used static analysis and fuzzing to search for bugs in open-source projects that were previously hidden due to sanitizer-eliding optimizations. This led us to discover 19 new bugs in Linux Containers, libmpeg2, NTFS-3G, and WINE. Finally, we present an effective mitigation strategy based on a custom Clang optimizer with an overhead increase of 4%.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Sanitizers, Fuzzing, Optimizations

ACM Reference Format:
Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. 2023. Don't Look UB: Exposing Sanitizer-Eliding Compiler Optimizations. *Proc. ACM Program. Lang.* 7, PLDI, Article 143 (June 2023), 21 pages. https://doi.org/10.1145/3591257

1 INTRODUCTION

Languages like C, C++, and Rust specify that programs must not contain logic classified as 'undefined behavior'. If a program contains such behavior, the specification no longer mandates how an implementation has to interpret the program, and the program is rendered de facto meaningless [Wang et al. 2012, 2013; Yang et al. 2011]. Yet, by default, the compiler is not required to diagnose this condition. This combination of unpredictable results and lack of detection has made undefined behavior the premier source of software vulnerabilities such as buffer overflows, null-pointer dereferences, or use-after-free.

Authors' addresses: Raphael Isemann, Vrije Universiteit Amsterdam, Amsterdam, 1081 HV, The Netherlands, r.isemann@vu.nl; Cristiano Giuffrida, Vrije Universiteit Amsterdam, Amsterdam, 1081 HV, The Netherlands, giuffrida@cs.vu.nl; Herbert Bos, Vrije Universiteit Amsterdam, Amsterdam, 1081 HV, The Netherlands, herbertb@cs.vu.nl; Erik van der Kouwe, Vrije Universiteit Amsterdam, Amsterdam, 1081 HV, The Netherlands, vdkouwe@cs.vu.nl; Klaus von Gleissenthall, Vrije Universiteit Amsterdam, Amsterdam, 1081 HV, The Netherlands, k.freiherrvongleissenthal@vu.nl.

2475-1421/2023/6-ART143
https://doi.org/10.1145/3591257

One promising solution for detecting undefined behavior has emerged in the form of sanitizers [Song et al. 2019], a compiler feature found in all major toolchains such as Clang, GCC, and Rust. Sanitizers inject runtime checks during compilation; if a bug is detected during the execution, the program is aborted and the bug is reported to the user. Because sanitizers have access to both the original source program and the exact runtime values, they produce user-friendly error reports, typically with no false positives. As such, sanitizers are widely used in combination with unit testing or fuzzing to find security-sensitive bugs [Serebryany 2017].

In practice, the applicability of sanitizers is limited by the overhead of the injected checks [Wagner et al. 2015; Zhang et al. 2021, 2022]. In automated testing, the overhead significantly slows down bug discovery [Jeon et al. 2020], while in production, it may make sanitization impractical altogether. To reduce the negative performance impact of sanitizers, modern compiler toolchains such as Clang, GCC, and Rust thus rely on standard optimization passes.

But doing so combines two compiler features with conflicting objectives. While sanitizers want to expose undefined behavior, optimizers often exploit these same properties for performance. This clash can lead an otherwise-valid compiler optimization to cause an essential (i.e., failing) sanitizer check to be omitted from the binary—either because the check itself or, more commonly, the code exhibiting undefined behavior is optimized away—we call this a *sanitizer-eliding optimization*. Sanitizer-eliding optimizations are particularly interesting from a security perspective, as the removed sanitizer checks can hide security-sensitive bugs from sanitizer-based bug finding tools such as fuzzers. Worse, since such optimizations tend to be particularly program- and compiler-sensitive, seemingly harmless changes to the program, compiler, or optimization settings (e.g., `-O2` instead of `-O1`) may unexpectedly re-introduce previously masked vulnerabilities in production binaries.

In this paper, we analyze the problem of sanitizer-eliding optimizations and show that they affect even the most popular sanitizers (ASan, MSan, and UBSan) in both GCC and Clang and may lead to a variety of undiscovered vulnerabilities such as memory leaks, uninitialized reads, and memory corruption. To do so, we have developed LooKUB, a framework to find such optimizations in modern compilers. LooKUB relies on a mutational fuzzing pipeline to automatically generate programs susceptible to undefined behavior and on a differential testing oracle to pinpoint sanitizer-eliding optimizations.

We have applied LooKUB to optimizers and sanitizers available in Clang and GCC. We show LooKUB can find 17 sanitizer-eliding optimizations in Clang and 5 in GCC. We then examine the different classes of issues and evaluate their practical bug-hiding impact by means of static analysis and fuzzing. Overall, we show our LooKUB-guided bug finding methodology is able to uncover 20 new bugs in popular programs (Ubuntu software packages and oss-fuzz). Finally, after identifying the problematic optimizations, we evaluate optimization configurations that do not exhibit the elisions and still optimize the sanitized code, resulting in an overhead increase of 4%.

Contributions. Summarizing, we make the following contributions:

- We present *LooKUB*, an open-source fuzzing method to automatically find sanitizer-eliding optimizations in compilers.
- An experimental evaluation of our method on the optimizers and sanitizers of the Clang and GCC toolchains.
- An analysis of how and where the bugs hidden by sanitizer-eliding optimizations can be found in real world software.
- A Clang-based compiler that preserves sanitizer checks while incurring a low performance overhead.

2 A MOTIVATING EXAMPLE

Listing 1 shows a C program providing an authentication service that compares a provided password against a stored secret password. The program also logs any failed login attempts to the `stderr` output stream. Aside from some general security issues, the program contains the following memory bugs:

(1) In lines 3 and 4 the program tries to clear `buf` but overflows the buffer by one byte due to the length check being `<=64` instead of `<64`.
(2) In line 5 the arguments to `memcpy` are reversed which assigns the `out` parameter to the pointer `buf` holding the allocated memory. This leaves `err_buf` in `main` uninitialized and leads to several uses of uninitialized values in the rest of the program (the first of which is on line 12).
(3) Line 12 has a reversed allocation failure check that generates an error when the memory allocation was successful (instead of failed). This causes our program to never actually free the allocated memory as all `free` calls are now only reachable via a failed `malloc` call.

```
1  static void make_buffer(char **out) {
2    char *buf = (char*)malloc(64);
3    for (unsigned i = 0; i<=64; ++i)
4      buf[i] = '\0';
5    memcpy(&buf, out, sizeof(char*));
6  }
7
8  int main(int argc, char **argv) {
9    if (argc <= 1) return 0;
10   char *err_buf;
11   make_buffer(&err_buf);
12   if (err_buf != 0) {
13     printf("Failed to allocate log!");
14     return 1;
15   }
16
17   if (strcmp(argv[1], "secret123")) {
18     strcpy(err_buf, "NO AUTH");
19     printf("Access denied!");
20     fprintf(stderr, "%s", err_buf);
21     free(err_buf);
22     return 1;
23   }
24
25   printf("Access granted!");
26   free(err_buf);
```

Listing 1. Motivational example.

Detecting the Bugs. All three bugs should be detectable by using one of Clang's supported sanitizers. Bug 1 should be detectable by AddressSanitizer, which covers generic invalid memory accesses such as buffer overflows. The first symptom of bug 2 is a conditional jump with an uninitialized value, which Clang's MemorySanitizer can detect. Bug 3 is a memory leak which can be detected by Clang's AddressSanitizer or LeakSanitizer (the latter being part of AddressSanitizer). Even though the example is artificial, the three bugs in it are real. The buffer overflow in line 3 and 4 is adapted from an overflow in the SPEC2006's `h264ref` benchmark. The uninitialized read due to a function not initializing an output variable is taken from the Linux container runtime LXC. The inverted allocation check is a bug we found in the Linux Scanner framework *SANE*.

Adding Optimizations. While we expect to find these errors by sanitizing and running the program below, the actual sanitizer reports heavily depend on the used optimizations. We can observe the following behavior when running the sanitized binary optimized on different Clang optimization levels.

(1) **No optimizations (`-O0`):** Both AddressSanitizer and MemorySanitizer detect the buffer overflow and uninitialized read respectively. The memory leak caused by bug 3 is reported if the previously found bugs are manually fixed.

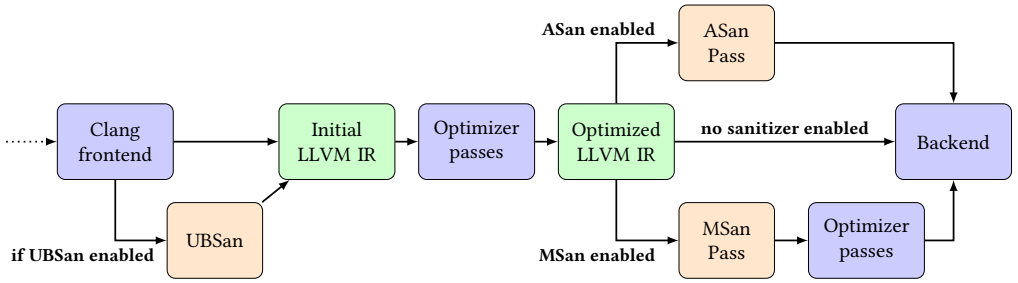

Fig. 1. Code generation and sanitization pipeline of Clang/LLVM.

(2) **Normal optimizations (-O1):** Both sanitizers fail to detect any of the three bugs. Furthermore, the compiled program now unconditionally takes the execution path that prints the error message `"Failed to allocate log!"`. Even with bug 1 and 2 manually fixed, the memory leak of bug 3 is not reported.
(3) **Full optimizations (-O2):** The sanitizers still do not report any of the three bugs. However, the memory bugs cause the optimizer to remove the authentication logic and the final program now allows logging in with any provided password.

This example demonstrates that in their current form, sanitizers and compiler optimizations do not reliably work together. This situation also forces the users of sanitizers to make the difficult choice between reliable detection of bugs and reasonable performance of the sanitized program. Ideally, compilers should be able to produce sanitized binaries that are both fast and capable of reliably detecting bugs. To this end, we develop tools that can automatically analyze the interplay between sanitizers and optimizers, and verify their soundness.

3 BACKGROUND

Sanitizers detect bugs by injecting checks into a program during compilation. When the injected checks detect a potential issue, they generate a detailed error report. Sanitizers typically transform code during compilation, calling functions in a separate runtime library for more complex functionality outside the fast path, such as error reporting logic. The runtime library also interposes some functions in the C standard library to implement some sanitizer checks.

Different sanitizers target different kinds of bugs, or offer alternative implementations for specific hardware or target applications. In this paper, we focus on the following three widely used sanitizers:

- **AddressSanitizer (ASan)** detects buffer overflows (heap and stack), use-after-free, use-after-scope bugs and memory leaks. It also detects invalid arguments passed to some C standard library functions such as `memcpy` [Serebryany et al. 2012].
- **UndefinedBehaviorSanitizer (UBSan)** consists of various checks for different kinds of generic undefined behavior such as signed integer overflows, dereferencing null pointers.
- **MemorySanitizer (MSan)** detects uninitialized memory uses [Stepanov and Serebryany 2015].

All three sanitizers are available in the LLVM-based Clang compiler, which supports sanitizing C and C++, as well as some derived languages. The GCC toolchain implements its own version of ASan and UBSan with compatible features, but does not currently support MSan.

Sanitization Pipeline. Figure 1 illustrates the sanitization pipeline in the Clang compiler. The frontend translates the C or C++ source code to LLVM's intermediate representation (LLVM IR). If UBSan is enabled, it intercepts the generation of the initial LLVM IR and injects its checking logic.

Afterwards the generated IR is passed to LLVM's optimizer passes. The specific passes that optimize the program depend on the optimization level and compilation flags. While modern compilers offer many optimization levels, we only discuss the effects of the most commonly used levels O0, O1, and O2.

- **`O0`:** This optimization level disables all optimization passes. While this severely impacts runtime performance, it also reduces compilation time and makes the final program easier to debug.
- **`O1`:** This enables most optimizations in Clang or GCC. This optimization level aims to be a middle ground between runtime performance, debuggability, and compilation time.
- **`O2`:** Nearly all optimizations are enabled at this level, further improving performance but making programs harder to debug.

Both optimization passes and sanitizers transform the LLVM IR, the optimizations being applied before the ASan or MSan sanitization passes. The MSan pass further optimizes the IR after sanitization by rerunning several standard optimization passes. Finally, the optimized IR is passed to the backend, which transforms LLVM IR into CPU-executable machine code. The sanitizers influence the remaining compilation process only in minor ways. E.g., they link the sanitizer runtime libraries into the program.

4 LOOKUB: AUTOMATICALLY FINDING SANITIZER-ELIDING OPTIMIZATIONS

This section describes LookUB, our method for detecting sanitizer-eliding optimizations.

Failure Preservation. Our method is built on the idea that compilers should preserve sanitizer failures in the presence of optimization passes. That is, the compiler must maintain the following invariant: If the execution of an unoptimized closed program reports a sanitizer failure, then the execution of the optimized version of the same program must also report a sanitizer failure. We say that a compiler which upholds this invariant is *failure preserving*. If failure preservation is violated, we can conclude that some part of the optimization process removed an essential sanitizer check. We use failure preservation as the test oracle in our randomized testing approach. By creating random programs and verifying whether they violate failure preservation, LookUB can automatically create test cases that demonstrate sanitizer-eliding optimizations in a given compiler. These test cases can then be used by developers to identify the triggered sanitizer-eliding optimization. LookUB consists of four components (see Figure 2):

- **Scheduler**: maintains a list of programs and in each iteration picks a program P that is then sent to the program mutator.
- **Mutator**: randomly mutates a program P and into P'. P' is then passed on to the test oracle.
- **Test oracle**: verifies whether program P' violates failure preservation. If a violation is detected, the program is saved as an interesting test case demonstrating a sanitizer-eliding optimization. If no violation is detected, the program is sent to the fitness function.
- **Fitness function**: takes a non-violating program P' and estimates the likelihood that a program will trigger a sanitizer-eliding optimization. Program P' and the calculated score are then sent back to the scheduler.

4.1 Scheduler

The scheduler maintains a priority list of programs (or *seeds*). In each iteration, it selects a program based on the score assigned by the fitness function, and passes it to the mutator. Each mutated program is assigned the same constant *energy* for simplicity, that is each program is mutated for a predetermined number of times. Initially the priority list can either consist of empty programs (i.e., with an empty `main` function) or randomly generated programs.

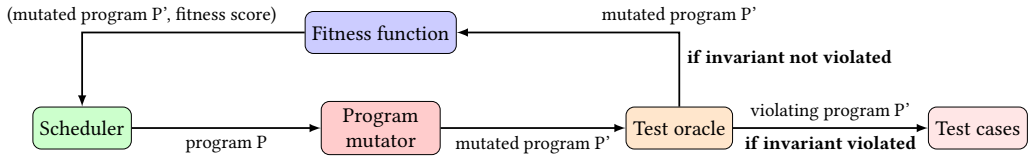

Fig. 2. Overview of LookUB.

4.2 Mutator

The mutator applies random changes to the program P provided by the scheduler. For example, it might add a new function to P or delete a statement. The only restriction to the kind of changes the mutator can perform is that the new program P' must be deterministic; we discuss this restriction in Section 4.3. This component can be implemented by existing mutators like MUSIC [Phan et al. 2018] or MILU [Jia and Harman 2008]. In our implementation, we decided to write our own code mutator, as explained in Section 5.1.

4.3 Test Oracle

The test oracle detects whether a program produces a sanitizer error that is removed by the optimizer. It first compiles the program with and without optimizations. It then executes both programs and searches their error output for sanitizer reports. If the unoptimized program produces a sanitizer report and the optimized version does not, our test oracle classifies the input program as a test case triggering a sanitizer eliding optimization. Note that our oracle does not check whether the exact same set of sanitizer reports are emitted with and without optimizations. This prevents false positives in cases where the compiler reorders sanitizer checks, as sanitizers typically only report the first violation and then abort the program.

Program Restrictions. To reliably identify invariant violations, the programs under test need to be deterministic. Our mutator therefore avoids calling non-deterministic functions like `time()`. However, the program may contain non-determinism, as long as the non-determinism is due to undefined behavior reported by a sanitizer. Should the program contain non-determinism despite our precautions, our classification strategy might lead to false positives. We explain the different causes of false positives and the techniques we use to avoid them in Section 8.

4.4 Fitness Function

The fitness function assigns a score to each mutated program. The score is used by the scheduler to select the next program for mutation. The fitness function is optional and without it LookUB runs effectively as a generational program generator. Its main purpose is to make it possible to guide the program generation towards specific sanitizer issues (see Section 5.3). The fitness function could also be used to guide the general compiler fuzzing process. However, previous work has shown that commonly used fitness functions such as the code coverage of the tested toolchain did not help speed up bug discovery [Di Luna et al. 2021].

5 IMPLEMENTATION

This section provides an overview of our implementation of LookUB. We needed 5000 lines of C++ code to implement the scheduler and code mutator. We implemented two test oracles and fitness functions for GCC and Clang in an additional 300 lines of Python code.

5.1 Mutator

We decided against using an existing code mutator, as existing mutators often only provide simple mutations or do not support features such as mutating C++ or code with compiler extensions. Instead, we created our own code mutator supporting a subset of C++. The mutator can create and mutate code that uses various C library functions, C++ heap allocation operators, `if` and `while` statements, language constructs for C++ exceptions (e.g., `throw`, `catch` and `noexcept`), and several compiler extensions. It also supports all integer and pointers types as well as related assignment, arithmetic, logical, and comparison operators. Additionally, it can create and use cv-qualified types, array types and function pointer types.

Our mutator operates on an abstract representation of the program that records functions, types, and expressions described as annotated abstract syntax trees. Our framework converts this internal representation to source code before it passes a test case to the compiler under test.

Mutation Process. The mutation process consists of three phases. First, the mutator picks a random subtree of the abstract syntax tree. Second, our mutator generates a new piece of code that fits the type and syntax restrictions of the original subtree. Third, the original subtree is replaced with the generated code.

The code generation in our mutator creates AST nodes matching a set of constraints. The generator randomly follows the production rules of a subset of the C++ grammar. The constraints limit the possible production rules the generator can follow. This ensures that the created code follows the C++ syntax and typing rules. E.g., a node might be required to be an expression of the type `int` and an `lvalue`. Given these constraints, the generator will not produce literals (as those are never an `lvalue`) or address-of operations (as they always return a pointer type). Each production rule also propagates constraints. For example, the production rule for the indirection/dereference operator would require that its child node evaluates to the type `long*` if the indirection operator itself is constrained to evaluate to `long`.

Type and Syntax Checking. Our mutator ensures that the majority of the created programs pass the type and syntax checks of the targeted compiler. This is partially ensured by the constraints inside the code generator. Additionally, manual checks within the generator itself prevent the creation of ill-formed programs. For example, before a function is deleted, the mutator checks that no call to this function exists. About 95% of the programs produced by our mutator can be successfully compiled by a standard C++ compiler. The remaining 5% of programs are ill-formed because of mutation checks that we have not yet implemented.

5.2 Scheduler

We implemented the scheduler as a priority list that stores candidate programs in the internal representation format used by the code mutator. The initial list consists of a single empty program. We limit the number of times each program can be passed to the code mutator. Once a program has been mutated often enough, we discard it from the priority queue.

5.3 Test Oracles and Fitness Function

We implement the test oracle and fitness function as standalone Python scripts which are invoked by the scheduler. Each test oracle executes a compiled test case only once with an empty input.

Clang Test Oracle. Our Clang test oracle uses the ASan, UBSan, and MSan sanitizers when compiling mutated programs. Because Clang does not allow enabling all sanitizers at the same time, every test input is separately compiled and executed with each of the tested sanitizers. The test oracle uses the sum of all sanitizer failures in each optimization level to detect violations of

failure preservation. That means the test oracle only considers a test case interesting if no sanitizer reports an issue in the optimized version. We did not test other sanitizers such as ThreadSanitizer or DataFlowSanitizer, as our generated test inputs do not use threads or data labels.

GCC Test Oracle. The GCC test oracle uses the ASan and UBSan sanitizers, but MSan is not available on GCC. To avoid false positives due to uninitialized values influencing compilation, the test oracle first runs every test input with Valgrind [Nethercote and Seward 2007] and ensures that the execution does not use uninitialized values.

Targeted Search Mode. The fitness function's purpose is to allow targeting specific sanitizer checks in the search process. This is implemented by searching the output of a program for a specified sanitizer error message (e.g. `"AddressSanitizer: heap-use-after-free on address"`). The fitness function assigns a higher score to programs that contain the specified string.

This targeted search mode allows LookUB to test compilers even if they container trivially discoverable sanitizer-eliding optimizations. For example, both GCC and Clang frequently remove sanitizer checks for memory leaks. This would end up blocking LookUB from finding other more interesting optimizations as the removed leak checks would mark many programs as interesting in the early mutation process. By specifying other non-leak sanitizer error messages with out targeted search mode, LookUB can circumvent these fuzz blockers and continue the testing process.

6 SANITIZER-ELIDING OPTIMIZATIONS IN CLANG AND GCC

This section evaluates LookUB by detecting sanitizer-eliding optimizations in Clang and GCC compiler. We use Clang/LLVM version 14.0 and GCC version 12.1 on a Linux system with glibc 2.35.

6.1 Clang Test setup

We tested Clang by running LookUB to generate a list of test cases that demonstrate sanitizer-eliding optimizations. We then deduplicated the test cases by resolving each exercised sanitizer-eliding optimization in a local Clang fork. Next, we repeated the testing process with our applied changes to find further issues. In total, we spent about 100,000 CPU hours testing Clang with LookUB. Our test machine was capable of running about 3 test inputs per CPU second.

6.2 Sanitizer-Eliding Optimizations in Clang

LookUB found 17 sanitizer-eliding optimizations affecting all tested sanitizers (ASan, MSan, and UBSan) in Clang 14. 14 of these bugs affected ASan, 2 bugs affected MSan and 1 bug affected all three sanitizers. In this section, we discuss the optimizations we found in more detail.

Dynamic Memory Transformations. LLVM contains several transformations that reason about dynamic memory (i.e., memory created by `malloc` or `new`). These transformations lack several checks that cause ASan checks to be elided. Consider for example Listing 4, which performs three dynamic memory allocations. The first allocation is leaked, the second one is deallocated twice (a double free bug), and the third allocation is correctly deallocated but then used twice afterwards (two generic use-after-free bugs). ASan can detect all bugs by interposing the (de-)allocation functions during program execution and instrumenting the load and store operations.

These respective checks are elided as follows. First, LLVM replaces the `printf("%d", *leak);` statement with `printf("%d", some_variable);`. This removes the only use of the allocation `leak` in our program. Because the `new` expression allocating `leak` itself has no observable behavior, LLVM removes it without checking whether there is a proper call to `delete`. The final sanitized

```c
struct Foo {
 int i;
};
int main(int argc, char **argv) {
 struct Foo f;
 int size = 5;

 // BUG: Foo is 4 (not 5) bytes,
 // so memset will overflow 'f'.
 memset(&f, 0, size);
 printf("%d", f.i);

 int *undefined;
 // Note: argc is always >= 0.
 if (argc >= 0) {
  // BUG: Random memory write.
  *undefined = 4;
 } else {
  printf("Unreachable code");
 }
}
```

Listing 2. Buffer overflow and uninitialized pointer use which are removed by the optimizer.

```c
char buffer[3] = {'a', 'b', 'c'};
// BUG: overlapping memory areas.
memcpy(buffer, buffer + 1, 2);
// BUG: overlapping memory areas.
strcpy(buffer, buffer);
// BUG: buf is not null terminated.
return strlen(buffer) != 0;
```

Listing 3. Overlapping memcpy/strcpy and buffer overflow which are removed by the optimizer.

```c
// BUG: This memory is leaked.
int *leak = new int(some_variable);
printf("%d", *leak);

char *double_free = (char*)malloc(8);
free(double_free);
// BUG: Double free bug.
free(double_free);

long *use_after_free = new long;
delete use_after_free;
// BUG: Write/Read after free.
*use_after_free = 4
return *use_after_free;
```

Listing 4. Memory leak and double free removed which are removed by the optimizer.

program lacks the single `new` call that ASan could interpose during runtime. Without this, ASan is unable to detect that there is no matching call to `delete` when the code is executed.

A similar process occurs for the `double_free` allocation. Because the allocated memory is unused, the optimizers considers removing the calls to `malloc` and `free`. While there is a safety check in the optimizer when removing dynamic allocations, the check only contains logic detecting mismatched memory management routines (CWE-762). No logic in the optimizer detects double-free bugs. The `malloc` and `free` calls are removed in the final program and ASan is unable to interpose them to perform its double-free check.

The two use-after-free bugs are removed in two sequential steps. First, the common subexpression elimination pass in the optimizer replaces `return *use_after_free;` with `return 4;`. The respective pass ignores the fact that `use_after_free` is not a valid allocation anymore. The assigment `*use_after_free = 4` is now considered a dead store and the optimizer removes it. With the use-after-free load and store operations removed, the ASan instrumentation running after the optimizer can no longer instrument it to detect the two bugs.

C Standard Library Transformations. LLVM knows the semantics of several C standard library functions and transforms calls to those functions in various ways. ASan, however, runtime interposes many of these functions to insert checks for their specific undefined behavior. Consider the example in Listing 3 that allocates a buffer and then calls several C library functions on it. The `memcpy`

call's behavior is undefined as its target and source memory areas overlap. Because the size of the `memcpy` operation is a small power of two, LLVM replaces this call with a pair of 2-byte load and store operations. These operations lack ASan's check for overlapping memory areas and hide the bug from the user. The `strcpy` call below is also undefined because of overlapping memory areas. Because source and target addresses of `strcpy` are equal, LLVM assumes the call has no effect and removes it. Finally, the `strlen` call contains a buffer overflow as the array lacks a null terminator. However, because the code only compares the length of the string to zero, LLVM replaces the call with a single check that only verifies whether the first character is a null terminator. This removes the memory accesses caused by `strlen` searching for the missing null terminator.

Removing Undefined Memory Accesses. We identified several places in LLVM that remove memory accesses to invalid regions. Consider for example the code in Listing 2. The `memset` call tries to zero out the 4-byte memory region of struct `f`. However, the `memset` call is given a byte size of 5, leading to a stack buffer overflow. LLVM can detect that the actual memory region is just 4 bytes large and will shorten the size parameter to 4 bytes. While this fixes the buffer overflow, the `memset` call no longer touches the redzone around the memory area. This causes that ASan can no longer detect it with its runtime interposed `memset` function.

The second bug in the example is the use of an uninitialized pointer to store a value, which can be detected by MSan. The use is inside the true branch of the `if (argc>=0)` statement, which is always taken as `argc` is always larger or equal 0. However, the optimizer detects that the dereferenced pointer is uninitialized, so it assumes that this branch is never taken. In our example, this causes the optimizer to unconditionally cause the execution of the (in theory) unreachable `else` part of the `if (argc>=0)` statement. The MSan check in line 17 is now never executed.

Removing Uninitialized Reads. LLVM removes reads from uninitialized memory. The values that were supposed to be read from memory are replaced by a placeholder value expressing an arbitrary bit value (usually expressed as `undef`). This in itself is not problematic, as theoretically the uses of this placeholder value could still be detected by MSan. However, several optimization passes replace this value by arbitrary picked concrete values, or remove the respective code entirely.

Consider for example the code shown in Listing 5. The `index` variable is uninitialized, and used to select one of the values in `argv`. Because the variable `index` is 4 bytes and our 64-bit system uses 8-byte pointers, the compiler zero extends the `index` before address computations. A sanitizer-eliding optimization replaces the zero-extended `undef` value with a constant zero. This erases both the uninitialized use and a potential buffer overflow that would be caught by ASan.

The `if` statement in Listing 5 demonstrates another optimization and the larger scope of this problem. The code creates an instance of struct `Foo` with only the member variable `a` initialized. The other member variable `i` is not initialized and used as the condition of the `if` statement. The optimizer first splits up struct `Foo` into two values for each member variable. Then it replaces the value of `uninit.i` with the placeholder value for uninitialized memory. Another optimization then removes the branch with the uninitialized use and instead unconditionally causes the execution of the `printf` call below. By removing the branch, the respective MSan failure is also removed.

Modeling of Pure Functions. LLVM has a notion of *pure* functions that cannot have observable side effects. This allows the optimizer to reorder those functions or remove the calls entirely if the result is not needed. During our testing we found that this approach can remove sanitizer checks in two ways. First, due to LLVM's internal knowledge of C standard library functions, it classifies many of them as pure. However, this categorization ignores the fact that many of these functions can lead to sanitizer errors when called with incorrect parameters. This leads to failing function calls being removed when the optimizer considers their results unused. A similar issue occurs when

```c
struct Foo {
  int a = 1;
  int i;
};
int main(int argc, char **argv) {
  unsigned index;
  // BUG: Uninitialized use.
  printf("%s", argv[index]);

  struct Foo uninit;
  // BUG: 'i' is uninitialized.
  if (uninit.i)
    printf("Index not zero");
```

Listing 5. Several uses of uninitialized memory which are removed by the optimizer.

```c
static int shift = 100;

__attribute__((pure)) int ub() {
  // BUG: shift value too large.
  return 1 << shift;
}

int main(int argc, char **argv) {
  // BUG: out-of-bounds read.
  int i = memcmp("a", argv, 1000);
  i = ub();
  return i;
}
```

Listing 6. Buffer overflow and invalid shift which are removed by the optimizer.

a user marks a function as pure via the `pure` attribute (a GCC extension). In this case, all sanitizer checks are now considered undefined side effects and the optimizer may remove them.

For example, consider the two function calls in Listing 6. The call to `memcmp` could read memory out-of-bounds, as its size parameter of 1000 is much larger than the length of the literal `"a"`. Because this standard library function is modelled as pure within Clang and the result is overwritten in the next line, the optimizer deletes the call and the the underlying ASan check.

The `ub` function is marked as pure by the user and the compiler trusts this annotation. The optimizer now assumes that `ub` has no side effects such as aborting the execution. However, the function also performs a shift with a right-hand side operand that is larger than the bit width of the left-hand side type. This is undefined behavior so UBSan inserts a check that will report this error and abort the execution. Because the UBSan check is a side effect in a function assumed to be pure, the check itself is now undefined behavior. In the final program the optimizer removes the undefined UBSan check and substitutes the evaluation of `ub()` with an arbitrarily chosen value.

6.3 Root Causes of Clang's Sanitizer-Eliding Optimizations

The distribution of the sanitizer-eliding optimizations we found shows that they mostly impacted ASan and MSan checks, while UBSan checks mostly resisted being removed by an optimization. We argue that there are two reasons why those two sanitizers have the most issues.

ASan/MSan Pass Placement. First, the placement of the ASan and MSan sanitization process after Clang's optimization pipeline is not optimal. This can be seen when comparing their implementation to UBSan which sanitizes code before the optimization phase. UBSan injects many of its checks as function calls that precede each sanitized operation. These function calls to non-inlineable functions are mostly opaque operations from the perspective of the optimizer and are rarely transformed during optimizations. While this effectively prevents various kinds of optimizations from transforming code, it also is more resistant against sanitizer-eliding optimizations.

In contrast to UBSan, MSan and ASan cannot inject any similar checks until after all optimizations have run. This means that each optimization pass is now required to explicitly preserve bugs during the compilation phase. While a few optimization passes contain such code or comments indicating that such code is needed, the vast majority of them do not make any attempt at preserving bugs. A possible remedy for this would be to run the ASan and MSan sanitization process before the optimizer. We evaluate this approach in Section 10.

```
const int s = 64;
char *buf = (char *)malloc(s);
if (!buf) return 0;
for (unsigned i = 0; i <= s; ++i) {
  char *x = buf + i;
  *x = '\0';
}
return buf[0];
```

Listing 7. Program that produces only an ASan error with UBSan enabled.

```
int uninit;
if (uninit + 1)
  return 1;
```

Listing 8. Program that produces a report-free binary with both MSan and UBSan enabled.

Sanitization via Function Interposition. The second root cause we found is reliance on runtime function interposition to inject sanitizer checks into a program. For example, ASan does this to catch memory leaks or undefined calls to C standard library functions. Like the late placement of the ASan/MSan passes, this technique is vulnerable to transformations that replace calls to interposed functions with code that does not perform the same sanitizer checks. However, there is no straightforward solution for fixing this issue. The only way of resolving this is to either disable the relevant optimizations, or to reimplement parts of ASan/MSan to no longer rely on function interposition.

6.4 Results of Combining Different Sanitizers in Clang

Our test oracle compiles every test input separately with each tested sanitizer. However, Clang allows combining some sanitizers when compiling a single executable. Specifically, Clang allows combining either ASan or MSan with UBSan but rejects any sanitizer combination containing both MSan with ASan. We found that combining sanitizers can influence how reliably the checks of a single sanitizer can catch errors.

Combining ASan and UBSan. When combining ASan with UBSan in Clang we see greatly increased detection rates for several previously missed bugs. The main reason for this is that the injected UBSan checks effectively disable several kinds of optimizations that previously removed ASan sanitizer failures.

The example in Listing 7 demonstrates a buffer overflow which ASan misses with optimization level `O2` or higher. However, the bug can be reliably detected if both UBSan and ASan are enabled. The actual UBSan change that prevents the optimizer from optimizing away the buffer overflow is the pointer overflow check injected into the expression `buf + i`. What is interesting about this check is that it is redundant. No overflow can occur here with a compliant `malloc` implementation.

Combining MSan and UBSan. Clang's MemorySanitizer implementation also supports being combined with UBSan. However, with our fuzzer we found that this combination does not work reliably even in unoptimized programs. Consider, for example, the code in Listing 8. MSan is able to detect this trivial uninitialized use bug without optimizations. However, when UBSan is enabled, the injected UBSan integer overflow checks around the addition seem to break the memory tracking of MSan. We assume the root cause for these kinds of bugs is that the UBSan checks are incorrectly marked by Clang as not needing any further sanitization. This blocks MSan from injecting its checks that would detect the uninitialized variable.

Table 1. Categorization of found bugs in Ubuntu and oss-fuzz.

Bug kind	Ubuntu packages	oss-fuzz projects
Memory leak	10	0
Uninitialized read	5	1
Buffer-overflow	0	1
Overlapping `memcpy`	1	1

6.5 Sanitizer-Eliding Optimizations in GCC

We also tested the sanitizers and optimizations in the GCC compiler. Our goal was to confirm that sanitizer-eliding optimizations are a problem in other compilers and how well our approach translates. We found 5 sanitizer-eliding optimizations with LookUB in GCC after a test run using an estimated 30 000 CPU hours. The optimizations we found form a subset of the ones we found in Clang. We also found that GCC in general implements fewer ASan checks. From the 17 sanitizer-eliding optimizations in Clang, 6 affect sanitizer checks that do not appear to be implemented in GCC.

6.6 Interaction with Compiler Warnings

Some sanitizer-eliding optimizations are caused by the compiler detecting undefined behavior. This raises the questions whether compilers warn when these optimizations are applied and if these warnings can replace the sanitizer checks. We found that neither GCC nor Clang can reliably emit warnings for any of the sanitizer-eliding optimizations we found. In some cases, those warnings are just not implemented in the compiler. Another reason is the different capabilities of optimizers and the compiler warning engines. For example, while Clang's optimizer can perform inlining of functions, the built-in warning engine cannot. This causes the optimizer to detect and remove bugs with the new information gathered from inlining, which is unavailable to the warning engine. This divergence between optimizer and warning engine reaches its peak when enabling link time optimizations. While the optimizer can use this to detect bugs even across source files, neither GCC nor Clang implement support for emitting warnings during this step of the compilation phase.

7 IMPACT OF SANITIZER-ELIDING OPTIMIZATIONS

In this section, we evaluate whether sanitizer-eliding optimizations in Clang have an impact on the quality of real-world software. For this, we searched for bugs in widely-used open-source software that were hidden by sanitizer-eliding optimizations. Figure 1 summarizes our results.

7.1 Finding Hidden Bugs via Static Analysis

For some sanitizer-eliding optimizations found in Clang, we can detect the bugs they are hiding via static analysis. We created an LLVM-based static analyzer designed to catch only bugs hidden by sanitizer-eliding optimizations. We based our design on the known approach of injecting bug detection logic into the optimization process [Wu et al. 2020]. Our static analyzer first detects when a problematic optimization is applied and then checks whether the optimization is about to remove an actual bug that would be caught by an enabled sanitizer.

Detecting Optimizations. To accurately detect when an optimization is applied, we implemented the static analyzer inside LLVM's optimization pipeline. For each relevant optimization, we embedded a check that is executed directly before the optimization actually transforms the code. For several types of bugs, we implemented a custom check that analyzes the IR and reports an error if a potential bug has been detected. For this experiment we selected three types of bugs:

(1) **Removed memory leaks**: This check intercepts the optimizer when it is about to remove a heap allocation function such as `malloc`, `calloc` or `new`. The bug detection logic consists of analyzing all uses of the to-be-removed pointer for any deallocation functions. If no deallocation function is called on the allocated pointer, then we assume this memory is leaked.
(2) **Removed uninitialized reads**: This check intercepts the optimizer when it is about to replace a value loaded from memory with an `undef` value (modeling unspecified values in LLVM). Our check analyses the uses of the loaded value and determines if it is directly or indirectly used in a conditional jump.
(3) **Removed overlapping `memcpy` bugs**: This check detects calls to `memcpy` with a size of 1, 2, 4 or 8 when they are transformed to a load and store (thereby removing overlapping memcpy checks). The bug detection logic then uses LLVM's alias analysis to check whether the copied memory areas overlap.

Scanning Setup. We implemented our static analyzer in a custom Clang compiler version that automatically analyzes all source code it compiles. We then repurposed the Ubuntu build system to rebuild and scan all 24329 Ubuntu 20.04 software packages that contained C or C++ source files. Each package is built with Clang's `-O3` optimization level and with link time optimizations, if the package's build system enabled them.

Reports. Our static analyzer generated 2700 bug reports distributed across 208 packages. Each report contained the suspected bug that is being hidden, the function the bug was found in, and a dump of the current LLVM IR within the optimizer. We semi-automatically analyzed these reports to filter out various kinds of false positives and unreportable bugs.

False Positives. Most of the false positive reports for removed memory leaks were caused by harmless memory leaks in programs. Examples for harmless memory leaks are single dynamic allocation in a program's `main` function or directly before a call to `exit` or `abort`. False positives of uninitialized uses were usually caused by dead code that was not yet optimized out.

Bug Reporting. Several bug reports were not actionable for various reasons, and we could not report bugs for them. For example, we found several bugs in packages where the developers were unreachable. We also did not report bugs in packages where the bugs seem intentional (e.g., `groff` appears to not deallocate some of its allocated buffers in an attempt to simplify its memory management code). We did not include these bugs in the final bug count.

Results. We found 16 new bugs in various open source projects including SANE, WINE, NTFS-3G, httperf, and X11. Of these 16 bugs, 10 were memory leaks, 5 uses of uninitialized values, and 1 a call to `memcpy` with overlapping source/target areas. 11 of these bugs have been fixed at the time of writing or are scheduled to be fixed for the next release of the respective project.

7.2 Searching Elided Bugs via Fuzzing

In addition to our static approach, we used fuzzing to find previously hidden bugs in programs with sanitizer-elided optimizations disabled. Unlike static analysis, this approach works reliably for all our identified sanitizer-eliding optimizations.

Fuzzed Software Set. We reuse the fuzzing setup of the oss-fuzz project [Serebryany 2017] to fuzz a large set of projects with minimal manual effort. We selected the 433 C or C++ projects in oss-fuzz as the target of our fuzzing evaluation. All 433 projects support being fuzzed with Clang's ASan and 169 of them support being fuzzed by Clang's MSan.

```
1 int divisor = 1;
2 jmp_buf env_buffer;
3 if (setjmp(env_buffer) != 0)
4   // BUG: Potential division by 0.
5   return 2 / divisor;
6 divisor = 0;
7 // Jumps back to line 3.
8 longjmp(env_buffer, 1);
```

Listing 9. Program with unsanitized but optimized behavior.

```
1 char *z = calloc(1, 4);
2 char *y = calloc(1, 2);
3 char *x = calloc(1, 4);
4 int res = *(x - 29) + *z;
5 free(x);
6 free(y);
7 free(z);
8 return res;
```

Listing 10. Program with ASan miss due to redzone alignment.

Fuzzing Setup. We disabled all optimizations to prevent sanitizer-eliding optimizations from affecting the results. We built all 433 selected projects and fuzzed them with either ASan or MSan enabled. We spent about 10,000 CPU hours fuzzing these projects, resulting in 424 crashing test inputs. To filter out crashes that were not hidden by optimizations, we reran every crashing input on a fuzz target built using the default set of optimizations. If an input triggers a sanitizer error in the unoptimized project but the optimized project does not report an error, we concluded it is a bug that was hidden by a sanitizer-eliding optimization.

Results. We found and reported 3 previously unknown bugs in the analyzed oss-fuzz projects.

(1) In the Linux container runtime LXC, we found an uninitialized value being used when calculating the supported process capabilities of the host Linux kernel. The bug was optimized out because this value was only used within an if-statement which was removed by a sanitizer-eliding optimization. Based on the project's git history, we estimate that this bug was introduced 8 months before we discovered it.
(2) In the image processing library leptonica, we found a heap buffer overflow. This bug was optimized out as the out-of-bounds read access was moved behind a bounds check. We estimate this bug to be introduced 2 years before we discovered it.
(3) In the widely used MPEG-2 video stream library `libmpeg2`, we found a call to `memcpy` with overlapping memory areas. This bug was optimized out due to the `memcpy` call moving 8 bytes of data which the optimizer lowered to a load and store instruction. According to the oss-fuzz issue tracker, this bug was the first new oss-fuzz bug found in `libmpeg2` in two years. The bug appears to be present in the initial commit in the version control system which was 7 years before we discovered it. The same issue was concurrently discovered and reported to Google by another security researcher.

8 THREATS TO VALIDITY

In this section, we discuss possible sources of false positives and how they can be avoided.

Unavoidable Sources of Nondeterminism. Our test oracle requires programs to be deterministic. However, there are some rare sources of non-determinism even in programs that do not call any non-deterministic external functions. For example, a randomly generated program could generate a pointer that points to the code or data segment of the running program. As the compiler might generate different segment contents on different optimization levels, the program execution can take a different execution path depending on the read values.

Generic Optimizer Bugs. Sanitizer-eliding optimizations are not optimizer bugs. They only change the semantics of programs that contain bugs or undefined behavior but are always valid when applied to well-defined programs. Distinguishing a sanitizer-eliding optimization from incorrect optimization is difficult as it requires verifying that there is no well-defined program where the same optimization would perform an invalid transformation. While test cases in this category are false positives for our purposes, we consider detecting generic optimizer bugs in general to be useful.

Probabilistic sanitizer checks. Some sanitizers are susceptible to false negatives which can be induced by valid optimizations. For example, AddressSanitizer uses redzones between allocations to track buffer overflows and out-of-bound memory accesses that do not touch these redzones are not detected. Given that both heap and stack allocations might be removed or rearranged by optimizations, it is possible that the allocations in the optimized binary are arranged in a way that causes an AddressSanitizer false negative.

Consider the example in Listing 10. Line 4 contains a buffer overflow as the pointer `x - 29` points outside the allocation created in line 3. ASan is able to catch this buffer overflow without optimizations as it touches a redzone area around the buffer `y`. However, the buffer `y` is unused and will be optimized out on optimization level 01. This causes the memory layout in the optimized binary to shift so that buffer `z` is now in the memory area that is accessed by the out-of-bounds write. Because this buffer is not a redzone, ASan is unable to detect this out-of-bounds write and the respective sanitizer failure appears to be optimized out. However, it is possible to filter out these false positives by, for example, rerunning test cases while randomizing the redzone size.

```
1  long *buffer;
2  size_t buffer_size;
3  long *allocate(long value) {
4    long *res = new long(value);
5  #ifdef DEBUG_PRINT
6    printf("DEBUG: alloc %p\n", res);
7  #endif
8    return res;
9  }
10 void func(long user_index,
11            long user_value) {
12   // Bounds check.
13   if (user_index < 0
14       || user_index >= buffer_size)
15     return;
16
17   time_t *t1 = allocate(time(NULL));
18   printf("TIME: %ld=", *t1);
19   delete t1;
20
21   long *t2 = allocate(0);
22   delete t1; // BUG: double-free.
23   long *index = allocate(user_index);
24   // BUG: Overwrites 'index' value.
25   *t2 = user_value;
26   printf("%ld\n", *t2);
27
28   // BUG: Buffer overflow.
29   buffer[*index] = user_value;
```

Listing 11. Bug exploitable depending on the configuration.

Difference in Resource Consumption. Compiler optimizations in most programming languages are allowed to remove memory allocations on heap and stack. Running out of available stack or heap memory is treated by all sanitizers as a bug that will be reported. It is therefore possible that an unoptimized binary consumes more memory than available and fails with a sanitizer report, while the optimized binary consumes less memory than available and reports no error. We do not consider optimizations that reduce memory consumption as sanitizer-eliding even though they violate the test invariant of our test oracle. The reason for this is that this would otherwise designate a large subset of optimizations in modern compilers as sanitizer-eliding.

Our workaround for these false positives consists of increasing or decreasing the amount of available stack and heap space when repeatedly executing binaries. Test programs that only violate our invariant for some memory limits are declared as false positives.

Unsanitized Undefined Behavior. A program can contain behavior that is not well-defined and not detectable by a sanitizer but still exploited by the optimizer. This means that the optimized version might take a different execution path which does not reproduce the sanitizer report. In our test oracle this would lead to a false positive report that a sanitizer-eliding optimization has been detected. For example, Listing 9 demonstrates undefined behavior that is removed due to an unsanitized bug. Line 6 changes the value of the local variable `divisor` and then performs a `longjmp` to a previous point in the execution, which leaves the value of `divisor` indeterminate. The value of `divisor` is afterwards used as the divisor in line 5, which is checked by Clang's and GCC's UBSan implementation to be non-zero. No current sanitizer detects that the indeterminate value of `divisor` is used. However, GCC and Clang's optimizers rely on the value of `divisor` not being allowed to change and replace the division in line 5 (and the associated sanitizer check) with a constant value of 2.

Even though we did encounter several such cases during our testing, it is possible to filter them out via adjustments to the compiler (e.g., disabling the relevant optimization) or code generator (e.g., do not generate calls to `longjmp` for the example above). Additionally, these test cases can be considered useful for the purpose of finding mismatched capabilities between optimizers and sanitizer. E.g., we found that Clang removes write accesses to constant memory buffers but offers no sanitizer that can detect this behavior. This compiler feature was hiding crashes in Busybox [Wells 2000] and the `jackd` audio daemon.

9 REACTIVATING ELIDED BUGS

The sanitizer-eliding optimizations in GCC and Clang often remove sanitizer checks alongside the underlying security issue. For example, removed memory leak checks in both compilers are caused by the respective heap allocation being removed from the program. In this section, we discuss how these hidden bugs can still impact the security of applications. The essential step in exposing an application with an elided bug is for the developer themselves to unintentionally reactivate the bug. We identify three likely scenarios how this could happen. Each real bug we found (see Section 7) can be reactivated using at least one of the described scenarios.

Example Vulnerable Program. Listing 11 shows a C++ program containing a double free bug. The two attacker-controlled `user_*` parameters are used to set the value inside `buffer` with a given `index`. The buffer access itself is guarded by a bounds check, which prevents a buffer overflow. However, the dynamic memory stored in `t1` is deallocated twice, leading to `t2` pointing to the memory of `index` on a system using glibc. This means that the attacker now controls both the address and value of an unrestricted write operation. This code is not reported as a double free bug by Clang/ASan as all three dynamic allocations and the associated double-free bugs are optimized out at `O2`. The optimized program then behaves as intended. However, the bug can be reactivated by minor changes.

Using Different Compilers. Different compilers interpret undefined behavior in various ways, and this affects sanitizer-eliding optimizations [Wang et al. 2013]. This can leave a program exploitable in a production environment, despite being sanitized and tested extensively. For example, only Clang will remove the double free bug in our example on optimization level `O1` while GCC keeps the program vulnerable on the same optimization level.

Changing the optimization level. The bug can also be reactivated by changing just the optimization level. For example, if this application was compiled using GCC using optimization level `O2`, it would not contain a double free bug. However, by switching to optimization level `O1`, the double free bug is no longer optimized out and becomes exploitable. In practice there is no clear defined optimization level that is consistently used in production or testing. E.g., oss-fuzz usually uses optimization level `O1` while the popular fuzzer AFL++ uses `O3` by default.

Activating UBSan. As discussed in Section 6.4, UBSan effectively disables most optimizations due to the checks it inserts at the start of the compilation process. While this is in general beneficial to the bug discovery process, it can be detrimental to the security of a program. The double free bug in our example is not exploitable on optimization level `O2` using Clang. However, when compiling the program with the same settings and UBSan enabled, the optimizer is no longer able to remove the double free bug and it can be exploited by an attacker.

Minor code changes. In its current state the code is highly unstable and even small changes can reactivate the bug. For example, a developer might be tempted to re-enable the `printf` call within `#ifdef DEBUG_PRINT` to better understand how memory is allocated in a production setup. However, this change disables the sanitizer-eliding optimization and reactivates the double-free bug on all optimization levels.

10 PERFORMANCE IMPACT OF FAILURE-PRESERVING SANITIZERS

Sanitizer-preserving optimizations trade off performance against detection rate. We evaluate two approaches for avoiding these optimizations in Clang to demonstrate their performance impact.

10.1 Implementation

In this section, we present the two sanitizer-preserving compilers we created. Both compilers no longer contain any of the sanitizer-eliding optimizations described in Section 6.

Early ASan/MSan Pass. The first approach is based on our suggestions from Section 6.3. For this, we created a custom Clang version in which we moved the ASan/MSan passes to the front of the optimization pipeline. This resolves the sanitizer-eliding optimizations we found for load/store instrumentation. We also disabled a small group of optimizations which remove calls to interposed C standard library functions. This required 4 lines of code to be changed in Clang.

Targeted Changes Guided by LookUB. With the second approach, we resolved each sanitizer-eliding optimization we identified via LookUB manually. Each change consists of removing a small part of an existing optimization pass or surrounding it with an additional check (e.g., by checking that there is no bug in a piece of code before it is transformed). In total, we changed about 300 LoC within this version of Clang.

10.2 Evaluation Setup

The benchmarking environment consists of a workstation with an Intel i7-8700K and 64 GB of memory. We compare our two custom Clang versions against a default Clang as the baseline.

10.3 Changes to SPEC CPU2006

SPEC CPU2006 contains several programming errors that result in sanitizer reports when processing the reference input. By default, these abort the program or incur additional performance overhead. As such, we applied two patches to our version of SPEC CPU2006 to fix the detected ASan and MSan failures. We benchmarked SPEC CPU2006 with and without our patches to ensure they did not influence the run time to a meaningful degree (i.e., no change above 1% for any benchmark).

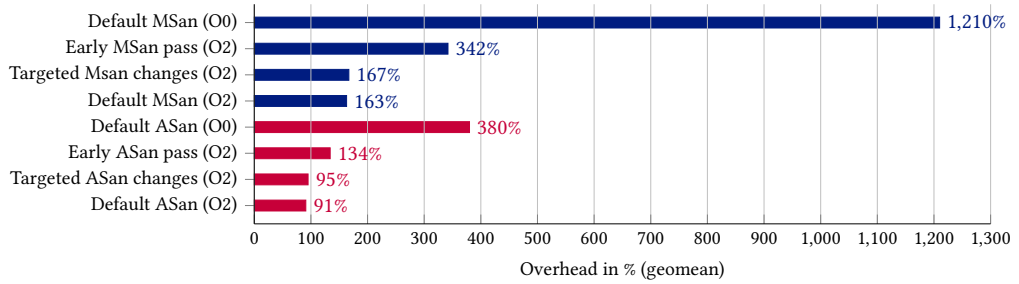

Fig. 3. SPEC CPU2006 overhead with ASan or MSan.

10.4 Benchmarking Results

Figure 3 shows the overhead of running ASan and MSan with each of the tested compilers. The baseline is the SPEC CPU2006 score without any sanitizers using the optimization level `O2`. The "Default" configuration is using an unmodified version of Clang 14 using the specified optimization level. The "Early pass" and "Targeted changes" configurations use our custom Clang versions as described in Section 10.1 using the `O2` optimization level.

No Optimizations. Disabling all optimization using the `O0` optimization level is a trivial solution for avoiding sanitizer-eliding optimizations. The large overhead in our benchmarks (up to 12.1 for MSan) shows that this approach is not viable in practice.

Early ASan/MSan Pass. As expected, the results show a large increase in overhead when moving the sanitization passes to the front of the pipeline. For instance, MSan's overhead more than doubles. When inspecting the generated code, we found that many redundant checks injected by these sanitizers were not removed by the optimizer, thereby degrading performance.

Targeted Changes. Our Clang version with a small set of targeted changes increases the overhead by only 4% when compared to default ASan. The generated code in this configuration is often similar to the one in ASan's default configuration, as most optimization passes run as expected. This shows that sanitizer-eliding optimizations do not provide major performance benefits and can be safely disabled in security testing scenarios.

Threats to Validity. Our measured overhead is that of a Clang version with a manually created sanitizer-preserving optimizer, based on the sanitizer-eliding optimizations we found. However, unknown (or future) sanitizer-eliding optimizations might require further changes, which could degrade performance. At the same time, existing changes within our tested Clang version could be too restrictive and disable optimizations when they do not actually elide a sanitizer failure.

11 RELATED WORK

Detecting undefined behavior. Wang et al [Wang et al. 2013] introduce the notion of *unstable code*; a code segment is called unstable, if a compiler may discard it under the assumption that undefined behavior cannot occur. Unstable code may be removed by the compiler and therefore may lead to a sanitizer-eliding optimization. [Wang et al. 2013] finds unstable code with a static analyzer called `STACK` which detects code fragments that can potentially be removed (or simplified) when assuming the absence of undefined behavior. Their method requires a precise semantic modeling of when undefined behavior may occur, which may be difficult, especially for advanced cases like undefined behavior linked to uninitialized memory, compiler extensions, floating point or standard

library functions. In contrast, LookUB directly relies on sanitizers to detect undefined behavior. This allows LookUB to directly test existing production compilers and their optimizers without requiring an often extensive formal specification of undefined behavior.

Compiler Testing. While previous work [Le et al. 2014; Livinskii et al. 2020; Yang et al. 2011] proposed differential testing of compilers using different optimization levels, their test oracles relied on the generated programs being free of undefined behavior. This was necessary to make the program behavior on different optimization levels comparable. LookUB's solution to comparing program behavior with undefined behavior is to detect it using sanitizers checks.

Sanitizer Testing. YARPGen [Livinskii et al. 2020] is a random program generator that generates programs free of undefined behavior. Even though it was primarily designed to test optimizations via differential testing, it can also be used as a tool for finding false positives in sanitizers. YARPGen solves the test oracle problem by verifying that the programs it generates are free of undefined behavior. Because all generated programs are well-defined, every sanitizer report that is generated when running a YARPGen program can be concluded to be a false positive. However, YARPGen cannot test sanitizers for false negatives as it is not designed to generate programs with the specific types of undefined behavior covered by the sanitizer.

Sanitizer Optimizations. While alternative optimizations approaches for sanitizers have been proposed [Wagner et al. 2015; Zhang et al. 2021], most of them do not guarantee preservation of sanitizer failures. One exception to this is `ASAN--`[Zhang et al. 2022], which implements several ASan-specific optimizations and verifies each optimization using formal techniques.

12 CONCLUSION

We presented LookUB, a differential testing based approach to systematically find sanitizer-eliding optimizations that hide bugs in programs. LookUB generates test programs using random program mutation and compiles them with sanitizers at different optimization levels. After running the binaries, any difference in the sanitizer failure reports reveals the presence of sanitizer-eliding optimizations. In our evaluation, we showed that such optimizations exist in modern production compilers such as Clang and GCC. Moreover, we demonstrated that the optimizations hide bugs in real software and described how they can be found via fuzzing or static analysis. Finally, we evaluated the performance impact of sanitizer-eliding optimizations and found that their contributions are insignificant. In conclusion, we recommend that standard optimization settings should preserve sanitizer checks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was supported by EKZ through the AVR Memo project, by Intel Corporation through the Allocamelus project, and by NWO through projects Theseus and INTERSECT.

ARTIFACT AVAILABILITY

All software artifacts are available on Zenodo [Isemann 2023]. The artifact contains a docker image with an instance of LookUB, our static analyzer and our oss-fuzz instance. It also contains instructions for each section on how to reproduce our experiments. LookUB is also available under an open source license on GitHub: https://github.com/vusec/LookUB.

REFERENCES

Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *ASPLOS*.

Raphael Isemann. 2023. *Artifact for "Don't Look UB"*. https://doi.org/10.5281/zenodo.7684001

Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. {FuZZan}: Efficient Sanitizer Metadata Design for Fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 249–263.

Yue Jia and Mark Harman. 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Testing: Academic & Industrial Conference - Practice and Research Techniques (taic part 2008)*. 94–98. https://doi.org/10.1109/TAIC-PART.2008.18

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*.

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. In *OOPSLA*. 1–25.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*. 89–100.

Duy Loc Phan, Yunho Kim, and Moonzoo Kim. 2018. Music: Mutation analysis tool with high configurability and extensibility. In *ICSTW*.

Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Security - Invited Talk.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*.

Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *IEEE S&P*. IEEE, 1275–1295.

Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*. IEEE, 46–55.

Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *IEEE S&P*. IEEE, 866–879.

Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *ApSys*. 1–7.

Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *SOSP*.

Nicholas Wells. 2000. Busybox: A swiss army knife for linux. *Linux Journal* 2000, 78es (2000), 10–es.

Zekai Wu, Wei Liu, Mingyue Liang, and Kai Song. 2020. Finding Bugs Compiler Knows but Doesn't Tell You: Dissecting Undefined Behavior Optimizations in LLVM. (2020). BlackHat Europe.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. 283–294.

Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. 2021. {SANRAZOR}: Reducing Redundant Sanitizer Checks in {C/C++} Programs. In *OSDI*.

Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. 2022. Debloating Address Sanitizer. In *Usenix Security Symposium*.

Received 2022-11-10; accepted 2023-03-31