



Dealing With C's Original Sin

Chris Hathhorn and Grigore Roşu

IN THE VERY early days of C, the compiler written by Dennis Ritchie and supplied with the UNIX operating system entirely defined the language. As the number of users and C implementations grew, however, so too did the need for a language standard—a contract between users and implementers about what should and should not count as C. This effort began in 1983 with the formation of a committee tasked with producing “an unambiguous and machine-independent definition of the language C” and led to the ANSI C Standard in 1989.¹ In retrospect, it was not until this date, 17 years after the first compiler, when C’s most notorious language feature slithered into the world: *undefined behavior*.

Trustworthy Programmers

The 1989 standard (and every one that has followed, with nearly unchanged wording) defines this term as “behavior ... for which the Standard imposes no requirements” (Sec. 1.6).² At the time, many understood undefined behavior to be a necessary



TRAFFIC ANALYZER

part of language design. “[Undefined behavior] is absolutely essential for standardization purposes,” wrote Tony Hoare in 1969, “since otherwise the language will be impossible to implement efficiently on differing hardware designs.”³ The 1989 standard committee justifies it as in the spirit of C, which it summarizes with phrases such as “trust the programmer” and “make it fast, even if it is not guaranteed to be portable.”⁴

Since C89, an appendix of each C standard has attempted to enumerate the individual cases of undefined behavior in the language. In C89, that appendix counted 95 instances of undefined behavior (Appendix A.6.2).² By C18, this number had crept up to 205 (Appendix J).⁵ And this list

likely leaves out many cases, as anything not explicitly described as “undefined” in the normative body of the standard might still be undefined simply by “omission of any explicit definition of behavior” (Sec. 4.3).⁵ Even attempting to characterize the extent to which undefined behavior permeates the standard requires careful reading between the lines.

And yet it accounts for many serious programmer errors. There are the big ones, of course, which are generally well understood: invalid pointer arithmetic, division by zero, and dereferencing invalid or null pointers. The ignominious reputation of C programs for poor security and exploitability is entirely attributable to undefined behavior. But many other undefined behaviors appear to be the work of a malicious language lawyer, serving no other purpose than to trip up otherwise well-intentioned, standards-conforming programs. For example, a nonempty source file ending in anything other than a new-line character results in undefined behavior (Sec. 5.1.1.2).⁵ Other sources of undefined behavior are a complicated minefield

not intended to make portability easier but to make specific optimizations more tractable (such as the strict aliasing rules; see the section “Strict Aliasing Violation”).

Optimization at Any Cost

Undefined behavior is very much a language design choice. The standard could severely limit the scope of allowed optimizations in its presence or require compilers to produce warnings or errors in many of the more trivial cases, often with little impact on program performance or the complexity of C implementations. We believe most users and implementers of C have since come around to the idea that much undefined behavior, or at least the extent to which it has proliferated in C and been exploited by optimizing compilers, is a mistake. If null references were a “billion dollar mistake” according to Tony Hoare,⁶ the liberal use of undefined behavior by C standards might exceed that amount by a few orders of magnitude.

As compilers become more sophisticated, they increasingly optimize under the assumption that programs never encounter undefined behavior, to a surprising and sometimes dangerous effect when a program actually does contain it. Programs that worked for a decade might start crashing in unexpected ways or spilling sensitive information after the compiler is upgraded. Undefined behavior so permeates the C standard, after all, that we would be unlikely to find a C program of nontrivial size without it.

C programmers often object that, on every platform they will ever target, integers are represented with two’s-complement arithmetic, which wraps on overflow, and creating invalid pointers will not crash the machine. Conversely, dereferencing a null pointer will cause a segmentation

fault, and division by zero will cause an exception. But in C, these are all behaviors of a particular version of a particular compiler. Compilers will decide to change such behavior in unexpected ways when it becomes convenient to do so, regardless of how one might expect the underlying hardware to behave in each case. Programming in standards-conforming C means programming for the strange abstract machine described by the standard, not for any particular compiler or hardware implementation.

Some of the confusion around undefined behavior perhaps comes from two further categories defined by the standard: *unspecified behavior* and *implementation-defined behavior*. In the former case, the standard gives compilers a choice among a set of possible values or behaviors (e.g., the order of evaluation of function arguments), with no requirement that the compiler make the same choice in every instance. Implementation-defined behavior (e.g., the size of `int`) differs from unspecified behavior only in the requirement that each compiler must document its choice. These other categories of behavior can be confused for undefined behavior, but they are radically different: the standard constrains the behavior of compilers to a small set of alternatives in these cases. With undefined behavior, the standard imposes no constraints at all. Whereas relying on implementation-defined behavior might cause an issue when porting a program to a new hardware platform, relying on undefined behavior is nearly always a bug, and often a serious bug. It might cause issues when upgrading the compiler, recompiling with different flags, or accepting input from a malicious user.

In a sense, undefined behavior represents a blank check to an optimizing compiler. An optimizing compiler

wants nothing more, after all, than to rewrite programs into their most pristine form, namely a `nop`. This motivation perhaps provides a theoretical limit to what a compiler will do to undefined programs, but in practice, optimizations predicated on the lack of undefined behavior can result in eliding critical security checks or overwriting memory. An optimizing compiler will not, in good faith, cause a program with undefined behavior to reformat a hard disk, but the attacker exploiting the undefined behavior resulting from a buffer overrun might.

That’s Nice, but I’m Stuck With C

How should one deal with undefined behavior, given its prevalence and the ever-growing willingness of optimizing compilers to exploit it? Awareness is perhaps the most important first step. We must resign ourselves to the reality that programming in C means encountering undefined behavior in our own programs.

One benefit of the age and popularity of C is the number of analyzers available. Many of these are free and easy to use, such as Valgrind and the clang sanitizers (UBSan, ASan, MSan, and TSan). Other tools, such as static analyzers and our tool, RV-Match (see the section “Examples”), have more overhead and complexity but can catch more undefined behavior. We should use these tools, of course, whenever possible. At the same time, we must avoid hubris and a false sense of security where undefined behavior is concerned. It is, unfortunately, very difficult to catch all cases of it, and no analyzer (or combination of analyzers) is perfect. Using analyzers is no substitute for being familiar with the standard and (at least) the most important classes of undefined behavior. Good references here are the SEI

CERT C Coding Standard⁷ and John Regehr,⁸ who provides more examples and suggestions for dealing with undefined behavior.

Examples

Here we present a few examples of undefined behavior and investigate how an optimizing compiler handles them. We also demonstrate the use of our tool, RV-Match, which is available for free download.⁹ RV-Match is a C reference implementation automatically generated from an open source formal semantics of the C language.¹⁰ Unlike modern optimizing compilers, which have a goal of producing binaries that are as small and as fast as possible at the expense of compiling programs that may be undefined, RV-Match instead aims at mathematically rigorous dynamic checking of programs for strict conformance with the C standard. The RV-Match command-line interface we will demonstrate here is `kcc`, a program meant to function as a drop-in replacement for compilers such as `gcc` and `clang`.

Signed Integer Overflow

As a first example, consider this program (`overflow.c`):

```
char * safe_copy(char * src, int buf_size) {
    buf_size += 1;           //for null
                             terminator.
    if (buf_size <= 0) return NULL; //check for
                                overflow.

    char * dest = malloc(buf_size);
    strncpy(dest, src, buf_size);
    return dest;
}

int main() {
    char * foo = "foo";
    char * copy1 = safe_copy(foo, strlen(foo));
    if (copy1) puts(copy1);
    char * copy2 = safe_copy(foo, INT_MAX);
```

```
    if (copy2) puts(copy2);
}
```

Compiled with `gcc` (version 7.3) or `clang` (version 6.0) with no flags, this program appears to execute correctly. The check for overflow appears to work: when executing the program, it only prints “foo” once and terminates normally. But when enabling optimizations in `clang`, we get:

```
$ clang -O3 overflow.c
$.out
foo
Segmentation fault
```

Why is this? The short answer: integer overflow is undefined, so it is futile attempting to check for overflow (with `buf_size <= 0`) after it has already occurred (at `buf_size += 1`). The standard requires compilers to preserve nothing about an execution that eventually encounters undefined behavior. As a result, undefined behavior can sometimes appear to propagate backwards through time because compilers are not required to prove code is free of undefined behavior before reordering it.

We can see exactly where the undefined behavior occurs if we run the same program through `kcc`:

```
$ kcc overflow.c
$.out
foo
Signed integer overflow:
> in safe_copy at overflow.c:7:7
in main at overflow.c:19:7

Undefined behavior (UB-CCV1):
see C11 section 6.5:5
see C11 section J.2:1 item 36
see CERT-C section INT32-C
see MISRA-C section 8.1:3
```

But what exactly caused the `clang`-compiled program to crash in

this example? It is unlikely to have simply elided the `buf_size <= 0` check because it still serves a purpose in perfectly defined executions where the argument to `safe_copy` is negative. If we check the generated object code, we can see what the optimizer actually did to this program:

```
safe_copy:
    pushq   %r15
    pushq   %r14
    pushq   %rbx
    movq    %rdi, %r14
    testl   %esi, %esi      # test buf_size
    js      LBB0_1          # jump if buf_size is negative
    addl    $1, %esi        # add 1 to buf_size
    # [...]
```

It moved the test for overflow to before the increment, allowing the function to avoid an extra increment for the case in which the test fails.

Unsequenced Side Effects

Consider this program (`unseq.c`):

```
int main() {
    int x = 0;
    return (x = 1) + (x = 2);
}
```

When compiled with `clang`, this program returns 3 but with `gcc` we unexpectedly get 4. We can see where the problem is by running this program through `kcc`:

```
$ kcc unseq.c
$.out
Unsequenced side effect on scalar object with side
effect of same object:
> in main at unseq.c:5:7

Undefined behavior (UB-EI08):
see C11 section 6.5:2
see C11 section J.2:1 item 35
see CERT-C section EXP30-C
see MISRA-C section 8.1:3
```

The expression in the return statement invokes undefined behavior by assigning to `x` twice without an intervening sequence point. We can check the object code generated by `gcc` to see how the optimizer exploited this undefined behavior:

```
movl    $0, -4(%rbp)    # x = 0;
movl    $1, -4(%rbp)    # x = 1;
movl    $2, -4(%rbp)    # x = 2;
movl    -4(%rbp), %eax   #
addl    %eax, %eax       #
ret                     # return x + x;
```

`Gcc` simply sequenced both assignments before the addition. Optimizations like this can turn even innocuous-seeming undefined behaviors into bugs affecting the behavior of a program.

Strict Aliasing Violation

Consider this program (`alias.c`):

```
int foo(int *p, long *q) {
    *p = 1;
    *q = 0;
    return *p + *q;
}
int main() {
    long x = 0;
    return foo((int *)&x, &x);
}
```

Compiled with `clang` and no flags, this program returns 0. If we enable optimizations (`-O3`), it returns 1. What is going on here? Let us check this program with `kcc`:

```
$kcc alias.c
$.a.out
Type of lvalue (int) not compatible with the
effective type of the object being accessed (long):
> in f at alias.c:2:7
in main at alias.c:9:7
```

Undefined behavior (UB-E1010):
see C11 section 6.5:7
see C11 section J.2:1 item 37

ABOUT THE AUTHORS



CHRIS HATHHORN is a software engineer at Runtime Verification, Inc. His research focus is on turning a formal semantics of C into a tool for catching bugs in real programs. Hathhorn received a Ph.D. in computer science from the University of Missouri. Contact him at chris.hathhorn@runtimeverification.com.



GRIGORE ROŞU is the chief executive officer of Runtime Verification, Inc. and a computer science professor at the University of Illinois at Urbana-Champaign, where he leads the Formal Systems Laboratory. Roşu received a Ph.D. in computer science from the University of California at San Diego. His research interests include programming languages; formal methods and software engineering; and, in particular, how to increase the safety, security, and dependability of computing systems. Contact him at grigore.rosu@runtimeverification.com.

see CERT-C section EXP39-C
see MISRA-C section 8.1:3


The standard generally allows compilers to assume pointers to objects of different types will not point to the same object. These are the strict aliasing rules, and violating them is undefined behavior. Regardless of whether we happen to be on a platform where `int` and `long` are both the same size and represented in the same way, accessing the same object through both `p` and `q` in this program is undefined behavior, and optimizers will exploit it. Checking the generated object code, we can see how `clang` does just that:

```
foo:
    movl    $1, (%rdi)    # *p = 1;
    movq    $0, (%rsi)    # *q = 0;
    movl    $1, %eax      #
    retq                     # return 1;
```

The assumption that `p` and `q` do not alias leads `clang` to evaluate the expression in the return statement to a constant 1.

We like the idea of a language that sacrifices everything—safety, security, perhaps even correctness—in the name of performance. We like the idea of a language that trusts its users and holds no hands. Such a language, however, should probably be a niche language, only pulled out when performance is needed above all else. It should probably not be a language still taught as a lingua franca in computer science departments across the world and still used in safety- and security-critical embedded systems.

At the same time, it is hard to imagine a C without undefined behavior. The first C standard included it, and it survives as a feature of the language today because it is easy: it is easy for the standards committees because it requires less consensus, it is easy for the implementers because it requires less error handling and simpler runtimes, and it is easy for the optimizers because it removes complicated proof obligations when transforming programs. And because of how easy

it makes bringing C to new platforms, perhaps undefined behavior is also responsible for C's rise in popularity and its persistence as a general-purpose programming language today. 

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1978.
2. *Programming Language C*, ANSI X3.159-1989, 1990.
3. T. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
4. ANSI X3J11 Committee. (1990). "Rationale for American national standard for information systems—Programming language—C." [Online]. Available: http://home.nvg.org/~skars/programming/C_Rationale.pdf
5. *Information Technology—Programming Languages—C*, ISO/IEC 9899:2018, 2018. [Online]. Available: <https://www.iso.org/standard/74528.html>
6. T. Hoare, "Null references: The billion dollar mistake," presented at QCon London. Aug. 25, 2009. [Online]. Available: <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>
7. Software Engineering Institute, "SEI CERT C Secure Coding Standard." Accessed on: June 1, 2019. [Online]. Available: <https://wiki.sei.cmu.edu/confluence/display/c>
8. P. Cuoq and J. Regehr, "Undefined behavior in 2017," Embedded in Academia, July 4, 2017. [Online]. Available: <https://blog.regehr.org/archives/1520>
9. Runtime Verification, "RV-Match." Accessed on: June 1, 2019. [Online]. Available: <https://runtimeverification.com/match/>
10. K Framework, "Semantics of C in K," GitHub. Accessed on: June 1, 2019. [Online]. Available: <https://github.com/kframework/c-semantics>



2019 IEEE Computer Society Election

Volunteer Leadership Is Vital



Vote Before
23 Sept. 2019

www.bit.ly/cs-election-19

Digital Object Identifier 10.1109/MS.2019.2930379