The Intended Meaning of *Undefined Behaviour* in C Programs

M. Anton Ertl*

TU Wien

Abstract. All significant C programs contain undefined behaviour. There are conflicting positions about how to deal with that: One position is that all these programs are broken and may be compiled to arbitrary code. Another position is that tested and working programs should continue to work as intended by the programmer with future versions of the same C compiler. In that context the advocates of the first position often claim that they do not know the intended meaning of a program with undefined behaviour. This paper explores this topic in greater depth. The goal is to preserve the behaviour of existing, tested programs. It is achieved by letting the compiler define a consistent mapping of C operations to machine code; and the compiler then has to stick to this behaviour during optimizations and in future releases.

1 Introduction

The C standard does not define the behaviour of most C programs; e.g., it does not define the behaviour of any terminating C program, nor of any library function not defined in the C standard. Some of these behaviours that the C standard does not define are called *undefined behaviour*. And while the C standard makes no difference between these and other non-standard behaviours as far as program conformance is concerned[1], language lawyers and some compiler maintainers are especially fond of *undefined behaviour*.

There are 203 undefined behaviours listed in appendix J of the C11 standard, and most C programs may perform some undefined behaviour under some circumstances, some of them unintended (e.g., buffer overflow vulnerabilities), some of them intended (e.g., a check protecting against a buffer overflow that contains an address computation overflow intended to wrap around).

* Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

[1] Any working C program is a *conformant C program*, and no program with any kind of behaviour not defined in the C standard (e.g., no terminating program) is a *strictly conformant C program*.

A huge number of C programs[2] performs undefined behaviours, even some[3] maintained by the very people who advocate[4] that no C program should perform undefined behaviour.

In the good old days compilers generated code consistently. The behaviour of the generated code for the undefined-behaviour cases was a straightforward extension of the behaviour for the normal case (see Section 4).

Unfortunately, over the last two decades a significant number of compiler maintainers have taken the attitude that their compilers can assume that the programs they compile don't perform undefined behaviour, and this leads to undesirable consequences in combination with sophisticated optimization.[5] In the words of John Regehr[6]: "A sufficiently advanced compiler is indistinguishable from an adversary." In the rest of this paper, such compilers are called *adversarial compilers*.

What to do about it? Some people suggest "fixing" all programs with undefined behaviour, and propose to find them using "sanitizers", i.e., run-time checkers that report some of the undefined behaviours that occur during a given execution run.

An alternative approach is to actually compile these programs as intended. Then programmers can spend their valuable time on hunting down and fixing the remaining bugs, on increasing the performance effectively and/or on extending the functionality of the program. One claim that the advocates of adversarial compilers have made is that they don't know the intended meaning of programs with undefined behaviour, going as far as claiming that it would take psychic powers[7] to compile the code as intended by the programmers.

In this paper I take a closer look at what the intended meaning of C programs is. Section 2 looks at previous work. Section 3 explains that the dream of a totally defined C is not realistic because of architectural differences. Section 4 gives a basic, tight model of the intended meaning. Section 5 discusses whether, how and why that model can be loosened while still staying within the intended meaning of existing, tested programs, and how that affects optimizations. Section 6 looks at how this applies to C library functions, using `memcpy()` as example.

---

[2] Dan Bernstein estimates 100% https://groups.google.com/forum/message/raw?msg=boring-crypto/48qa1kWignU/b7WsAezEAwAJ.

[3] http://blog.regehr.org/archives/761

[4] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[5] E.g., OpenSSH had a `memset()` intended to erase the private key, i.e., to mitigate the effects of some vulnerability elsewhere; the compiler "optimized" this `memset()` away, because it would require undefined behaviour to read this private key; unfortunately, OpenSSH contained such a vulnerability, the undefined behaviour happened, and private keys were compromised. https://lwn.net/Articles/672465/

[6] https://blog.regehr.org/archives/970

[7] <news:h02dnSk5W8n4p57OnZ2dnUVZ_qSdnZ2d@supernews.com>

2 Previous work

Wang et al. [WCC+12] give a number of examples of the problems that adversarial compilers cause to existing programs, and provide performance numbers for the effect of optimizations based on adversarial assumptions.

Some people have reacted to adversarial compilers by trying to initiate a tighter definition of C.

E.g., John Regehr et al. started on a Proposal for a Friendly Dialect of C[8]; in their initial posting they proposed some reasonable behaviour for some undefined behaviours and wanted to continue based on finding a consensus among C experts. They eventually gave up[9], partly because finding a consensus is very tough going, and partly because hardware differences make it impractical to make C completely defined in some places. The present paper takes existing, tested programs rather than experts as the yardstick for determining what a piece of C means, and accepts that programs may be hardware-specific. The two examples of disagreements mentioned by Regehr et al. are discussed in the present paper: `memcpy()` in Section 6, and shifts in Section 3.

In his proposal for boringcc,[10] Dan Bernstein partly takes a very similar position to the one I take in the present paper (compile existing programs to behave the same way as old compilers do), and partly defines things beyond that for security reasons, e.g., initializing everything to zero, and tightly defining the behaviour of out-of-bounds accesses (the latter will have a huge performance impact). You can see the present paper as a more elaborate version of the first aspect of boringcc.

Wang et al. [WZKSL13] have written a static program analyzer that tries to find code that an adversarial compiler might optimize away against the intent of the programmer. They identify a C dialect C* that assigns well-defined semantics to code fragments that have undefined behaviour in C; they give some examples of these semantics, but do not discuss them in any depth.

In earlier work [Ert15] I focused on debunking the performance claims of the advocates of adversarial compilation, but I also shortly discussed the intended meaning of programs. Some readers were not satisfied with that discussion, so here I present a more elaborate version of these ideas.

In the Linux kernel/user space interface, the Linux maintainers take exactly the position that I take here for the source-code/compiler interface: Existing programs that worked on an earlier version have to continue working on a new version, even if it can be argued that the programs are buggy.[11]

A debate related to the one that lead to this paper is the one about VAXocentrism[12]. One significant difference is that the VAXocentrism debate was about portability of programs to other platforms, while the present discussion

[8] https://blog.regehr.org/archives/1180
[9] https://blog.regehr.org/archives/1287
[10] https://groups.google.com/forum/#!msg/boring-crypto/48qa1kWignU/o8GGp2K1DAAJ
[11] https://felipec.wordpress.com/2013/10/07/the-linux-way/
[12] http://catb.org/jargon/html/V/vaxocentrism.html

is about whether optimization or newer versions of a compiler on the same platform should compile existing, tested programs to preserve the behaviour of the old versions of these compilers.

Looking at the individual VAXocentric assumptions, it is interesting that some assumptions are now universally valid on general-purpose hardware (flat address space, byte addressing and thus a single pointer format), while advocates of adversarial compilers point to ancient or hypothetical hardware to justify adversarial compilation on modern hardware; other assumptions seem to be on the way to winning (general-purpose machines with big-endian byte order or that trap on unaligned accesses for normal loads/stores are becoming rare); on the other hand, some VAXocentric assumptions (e.g., dereferencing null pointers) are now almost universally invalid on general-purpose platforms.

3 Totally defined C is impractical

Different computer architectures differ in instruction set, but in general have a lot of hardware characteristics in common; however, there are also differences. These (present or former) differences are the source of some prominent undefined behaviours. E.g., the different signed-number representations of earlier times, and resulting differences in signed overflow behaviour led to signed overflow not being defined in C.

While hardware now has consolidated on the 2s-complement representation of signed numbers and all current hardware can perform wraparound arithmetic efficiently, there still are other operations where different hardware produces different behaviour: e.g., shifting by the data width, unaligned accesses, byte order, data sizes, or the behaviour on integer division by zero or overflow.

This is one reason why a total definition of C across all platforms is impractical. E.g., if the totally-defined C specified that shifting by the data width produces 0, the compiler would have to implement shifts more expensively on some machines; and if it specified that it produces the unshifted value, it would have to implement shifts more expensively on other machines. Allowing either result turns this into a portability problem, which is not nice, but far preferable to undefined behaviour. E.g., the rotation idiom `(x<<n)|(x>>(32-n))` produces the intended `x` for `n=0` on either kind of machine, while an adversarial compiler (Clang) compiles this to 0 if it knows that `n=0`.

For shifts the cost of emulating the other behaviour is not that big, but in other cases it can be significant, e.g., the behaviour on unaligned accesses: Simulating unaligned accesses on hardware requiring aligned accesses has significant costs (e.g., 11 instructions instead of 1 for a store on Alpha); conversely, simulating alignment traps on hardware that handles unaligned accesses transparently is not cheap, either, simply because loads and stores are frequent.

In the end, C is a language that is close to the machine, so it is appropriate if differences between machines are reflected in the behaviour of the resulting code in certain cases. Of course, if someone wants to implement a set of compilers for different machines that exhibits common behaviour rather than machine-specific

behaviour for some operations, that is also ok. Such a compiler would not be able to run some programs written for a closer-to-the-machine compiler; that's acceptable, because it's a different compiler, not a newer release of the same compiler.

So the intended behaviour may be hardware-dependent and therefore non-portable. While portability would be nice, this is and always has been an issue that the programmer had to solve, and is outside the scope of the present paper: If a programmer has written a program that only works on, say, AMD64 machines, it should continue to work on AMD64 machines with the next version of the compiler; if it has not worked on ARM before, it is unlikely to work on ARM in the future without changes.

Even portable programs often contain hardware-specific code, often protected by an appropriate `#ifdef`.

4 Basic model of the intended meaning

In the basic model the C compiler maps each C operation consistently to a machine instruction or a sequence of machine instructions *of its choice* that satisfies the requirements of the C standard. E.g., on AMD64, `*p` (where `p` points to a 64-bit value) might be translated to `mov (%rax), %rax`.

This basic mapping is the specification for the behaviour, including non-standard cases. The compiler writer can use the considerable freedom that the C standard provides to choose a behaviour amenable for his goals (performance, safety, compatibility with other compilers, etc.), but once the behaviour has been set, the compiler maintainer must preserve it. The compiler is free to produce more efficient code in optimization modes, and/or in later releases, but has to preserve the behaviour of the basic mapping.

As an example, the AMD64 `mov` instruction above loads the 64-bit value starting at `p` even if `p` is not 8-byte aligned, so on optimization that behaviour has to be preserved. Auto-vectorization is a valid optimization, but has to preserve this behaviour; one way to achieve this is to use the `movdqu`/`vmovdqu` instruction (which does not pose alignment requirements) when autovectorizing a `mov` rather than `movdqa`/`vmovdqa` (which traps when the accessed address is not 16/32-byte aligned).

Even if using `movdqa`/`vmovdqa` was faster (it isn't[13]), using it would be wrong, because it does not preserve behaviour. That does not mean that there should be no way to make use of `movdqa`/`vmovdqa`, only that that must not be achieved by optimizing instructions that do not pose alignment requirements.

5 Loosening the basic model

As explained below, the basic model is a little too tight for a language like C. This means that we have to leave the heights of absolute equivalence, and have to find

[13] http://pzemtsov.github.io/2016/11/06/bug-story-alignment-on-x86.html
http://www.complang.tuwien.ac.at/anton/autovectors/

out whether existing, tested programs rely on certain properties. Fortunately, we have a large corpus of programs for testing that. In some cases, we can also find by reasoning that programs are unlikely to rely on certain properties.

5.1 Out-of-bounds memory accesses

If we want to be absolutely equivalent in the case of out-of-bounds memory accesses, no change is possible (no optimization, and no other change to the executable code). That's because we can use out-of-bounds memory accesses to read the executable machine code, and any change to the executable code can, in theory, change the behaviour.

While there are some programs that access the executable machine code [RS96,PV08], no program relies on the exact contents of specific memory locations. The reason is that such a program would be unmaintainable even without compiler changes: Any change to the source code of such a program, even inserting a piece of code that is not executed, would have a good chance to change its behaviour. Because programs do not rely on that, the compiler can change the machine code, e.g., for optimization.

For data memory, I have seen code that accesses neighbour parameters of a function using address arithmetic.[14] So there are cases where code makes assumptions about the layout of data in memory. These assumptions are probably for data that tend to have the same layout across different compiler versions anyway as long as the definitions of these data and any definitions between them are not changed (e.g., automatic variables in memory or global variables). So a compiler does not have to be extra careful in order not to break these assumptions, the usual amount of care is sufficient.

In addition to programs that rely on a particular result from an out-of-bounds access, there are also programs that perform such an access, but where the actual result is not relied on, because it is not used, the out-of-bounds part is masked away, or because it cancels itself out. Moreover, there are cases where an out-of-bounds address (beyond the one-past-the-end exception) is computed, but not accessed. In such cases an adversarial compiler maintainer feels entitled to compile the program into anything, and such cases have happened (e.g., "optimizing" a finite loop into an infinite loop, or "optimizing" a bounds check away), and of course, a benign compiler will compile the program as intended.

5.2 Uninitialized data

Access to uninitialized data is another issue where absolute equivalence with the basic model would make important optimizations impossible. Consider a variable v at the end of its life (e.g., at the end of a function). Unless the compiler can prove that the location of the variable is not read later as a result of reading uninitialized data (say, reading the uninitialized variable w living in the same

[14] And, to my surprise, this code worked even on RISCs of the day (1990s), despite register-based calling conventions.

location in a different function), v would have to stay in the same location in future compiler versions or other optimization levels; or at least the final value of v would have to be stored in this location, and the initial value of w would have to be fetched from this location.

Fortunately, existing tested programs avoid relying on this equivalence, because it tends to break on maintenance even while keeping the compiler version the same: if the maintenance programmer inserts or deletes a variable or field, the location of v, w, or both can easily change and thus break the program. In practice, compilers have long performed register allocation, copy propagation and other optimizations that can change the value of uninitialized variables, including in compilers that are generally considered benign compilers and have been named as models for boringcc, such as early versions of gcc.

Another way to deal with this issue is to actually initialize data to, say, 0, in the basic model; then rearranging the variables does not change the behaviour of accessing uninitialized variables. This is relatively cheap for automatic scalar variables, because the initialization by the program would turn most of these auto-initializations into dead code, which would be optimized away and therefore cost nothing. For bigger compound data the balance is different: possibly higher initialization cost,[15] and lower register allocation return, so one might consider performing the initialization only for scalars; however, introducing such a difference complicates the programming model and is not programmer-friendly.

5.3 Optimizations

A benign compiler like, say, early gcc, can be viewed as following this looser model. In addition to the optimizations performed by early gcc, many other optimizations are possible in this model that are not covered by early gcc; e.g., auto-vectorization, the pride of recent gcc releases, is compatible with this model.[16] In other words, the compiler does not need to be adversarial in order to perform optimizations.

I have not discussed multi-threading here. It complicates matters, but can probably be resolved with similar reasoning, especially because things like race conditions are so unpredictable that programmers cannot rely on much anyway. However, I do not have enough knowledge in that area to present such a reasoning.

6 Library functions

Many programs do not rely on the implementation internals of library functions, but some do, so it's a good idea to check any potential change in library functions against the corpus of existing programs.

[15] This needs further investigation; loading a cache line from main memory can be much more expensive than zeroing it.

[16] However, I think that it is better to let the programmer express vector operations directly rather than letting the programmer write scalar code, and hoping that he writes it in a way that the compiler can auto-vectorize.

A prominent example is `memcpy()`. If the source and destination ranges do not overlap, the order in which the bytes are loaded and stored does not make a difference, but if they overlap, it does. So an optimization that is in line with the basic model is to check for overlap: If there is overlap, perform something absolutely equivalent to the original implementation; if not, and use the most efficient order.

Looking at the discussion surrounding a change in the `memcpy()` implementation in glibc[17], we can determine something more about what existing, tested programs relied on at the time. Some programs were affected by the difference between the versions, so they called `memcpy()` with overlapping source and destination. Old versions of glibc behave like `memmove()` when the destination address is lower than the source address, and exhibit varying behaviour (that did not lead to such bug reports) when the destination address is higher than the source address; using `memmove()` instead of `memcpy()` did not break the programs, either. But the change that led to the discussion made the programs behave differently than intended, so the programs obviously had overlapping source and destination ranges.

The programs obviously relied on being able to copy from higher to lower addresses even if the ranges overlapped, but did not use `memcpy()` for copying overlapping ranges from lower to higher addresses. So in theory there was the option to make use of this to have a more efficient `memcpy()` than `memmove()` while preserving the intended behaviour. In practice, the performance difference is close to zero, and the use of `memmove()` in glibc 2.24 is a good solution.

7 Conclusion

While the position for adversarial compilers is relatively simple and clear, the intended meaning of programs with undefined behaviour has been called into question. This paper defines a benign position based on preserving the behaviour of existing, tested programs on the platforms that they were tested on. This position does not solve portability problems, but these were problems that C programmers had to solve by themselves already in the good old days of benign compilers.

In the basic model the compiler just maps each C operation to a machine instruction or sequence of machine instructions, consistently, and then preserves that behaviour in optimization or future versions. This basic model is too tight for any optimization, but fortunately, we can loosen this model a little without breaking existing, tested programs. So the results of out-of-bounds accesses and of reading uninitialized data may vary to some extent between compilations, which enables all optimizations available in early (benign) gcc, and also other optimizations, e.g., auto-vectorization.

[17] https://bugzilla.redhat.com/show_bug.cgi?id=638477
https://sourceware.org/bugzilla/show_bug.cgi?id=12518

References

Ert15. M. Anton Ertl. What every compiler writer should know about programmers. In Jens Knoop and M. Anton Ertl, editors, *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'13)*, pages 112–133, 2015. 2

PV08. Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction (CC'08)*, pages 163–177. Springer LNCS 4959, 2008. 5.1

RS96. Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996. 5.1

WCC+12. Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Asia-Pacific Workshop on Systems (APSYS'12)*, 2012. 2

WZKSL13. Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 260–275. ACM, 2013. 2