Analysis of Speedup Gain of Undefined Behavior Optimizations in OpenBSD

1st Lucian-Ioan Popescu
CS Department, Politehnica University of Bucharest
lucian.popescu187@gmail.com

***Abstract*—The ISO C Standard added the undefined behavior notion as a mean to portability. State-of-the-art compilers such as GCC and Clang/LLVM use it to issue aggressive optimizations that break the the intention of the progammer. We argue that the performance impact of undefined behavior (UB) optimizations in operating systems, such as OpenBSD, is low. Furthermore they introduce unobservable and undocumented effects that have great impact of program robustness and security. To test our hypothesis we take the compiler implementation used in OpenBSD, i.e. Clang/LLVM, and disable all undefined behavior optimizations. Then we compare the performance of the system on multiple hardware architectures with the above mentioned optimizations turned on and off.**

***Index Terms*—compiler, optimization, undefined behavior, operating system**

I. Context and motivation

The C90 [22] standard provides a loose definition of undefined behavior. This permits compiler implementations to abuse the definition and use it as a mean to aggressive optimizations that break the intention of the programmer. Code that works with a lower level of optimization is broken when the optimization level is elevated. Furthermore code that works on previous versions of the compiler is suddenly broken in newer versions because the standard imposes no requirements on undefined behavior.

This has created serious security problems throughout the years [3], [5], [28]. A number of initiatives to solve this problem were started from different parties [7], [18], [26], [29] however the problem still persists. The primary open source developer groups have seized the unsteady definition of undefined behavior to justify dangerous silent code transformations that break the intention of the programmer.

This philosophy is very dangerous in terms of programming expressivity. The compiler has very little context of what the developer wants to accomplish with a specific piece of code. For example the compiler cannot make the distinction between a read from an uninitialized memory region that might produce unspecified results and a read from a memory mapped device that cannot be written in order to initialize it. Or it cannot distinguish between an erroneous floating pointer access to an integer variable and a smart method of computing an arithmetic function [25]. The general principle is that the developer has the responsibility to decide what the code should do, the job of the compiler is to translate the code into machine readable instructions and to apply optimizations only when there is no risk of losing developer intentionality.

The argument of the people that defend this kind of optimizations is that C code that contains undefined behavior has no meaning and the compiler is free to do various types of modifications on it. In Control Theory terms, such a system is described by a low degree of controllability and observability. Which is paradoxical in the philosophy described above where the compiler forcefully takes the responsibility, from the developer, of generating relevant code. The implications of this are that no meaningful engineering can be done in this framework where processes inside a compiler cannot be understood and analyzed.

This has created an adversarial view of the compiler. Developers are forced to modify the default behavior of the compiler with flags such as -fno-delete-null-checks, -ftrapv, -fno-strict-aliasing, -ftrapv so that they can impose strong requirements on the code generated by the compiler. This in turn creates a more complicated development environment. This also creates de-facto standard ways of working with software, e.g. GCC first introduced -ftrapv and Clang/LLVM followed the trend and adopted the same flag.

Another argument that defends the aggressive optimizations view is that code generated by these compilers runs faster on artificial benchmarks [10], [11]. This does not necessarily hold for real-life software projects that differ in complexity from the artificial benchmarks and that make use of non-trivial code constructs. Ertl [19] makes an interesting observation regarding the performance of UB optimizations. He notes that source level changes buy greater speedup factors than UB optimizations for certain classes of programs. While his research in this field is valuable, the limitation of his work is that he draws conclusions based on SPECint benchmarks.

The contribution of this work is that we analyze the speedup factors of UB optimizations for real-life software projects, such as operating systems, in particular OpenBSD. We choose OpenBSD because it is a self-contained robust and secure implementation of operating system. The system emphasizes ”portability, standardization, correctness, proactive security [...]” [1], goals that are shared with our philosophy of generating code close to the intention of the programmer. Finally, by running this experiment we provide a trade off analysis between the performance gained using UB optimizations and the risks of issuing them.

This paper is structured as follows. Section II presents a background on UB optimizations with examples, risks and incomplete solutions for solving the risk they introduce and also introduces to the notion of programmer intentionality.

Section III describes previous results in analyzing the performance gain of UB optimizations. Section IV introduces our research plan that focuses on removing UB from OpenBSD and analyzing the performance of the system. Finally, Section V summarizes our research proposal.

II. Background

This section presents UB optimizations in real-life software projects such as Linux and OpenBSD. After we presents such examples, we provide an analysis of the risks they introduce and current solutions that try to tackle the risks, but are however incomplete. Then we introduce the notion of programmer intentionality and present how it correlates with UB optimizations.

A. Undefined Behavior Optimizations

Wang et al. [28] compiled a list of UB optimizations that show the dangerous effects of using the UB definition when issuing compiler optimizations. They created case studies for the following classes of undefined behaviors: division by zero, oversized shift, signed integer overflow, out-of-bounds pointer, null pointer dereference, type-punned pointer dereference and uninitialized read. The consequences of these optimizations range from unexpected code generation [8], [15] to real-life vulnerabilities [12].

Code snippets with a high risk of triggering UB optimizations are provided in Listings 1, 2 and 3.

```
if (!msize)
  msize = 1 / msize; /* provoke a signal */
```

Listing 1. Compiler assumes that dividing a number by zero makes no sense and the whole block is deleted (lib/mpi/mpi-pow.c in the Linux kernel)

```
ifa = &in6ifa_ifpwithaddr(ifp,
  &satosin6(rt_key(rt))->sin6_addr)->ia_ifa;
if (ifa) {
  ...
}
```

Listing 2. Compiler assumes that in6ifa_ifpwithaddr returns NULL then makes ipa6 NULL and deletes the if check (sys/netinet6/nd6.c in the OpenBSD kernel)

```
static __inline int
hibe_cmp(struct hiballoc_entry *l, struct
    hiballoc_entry *r)
{
  return l < r ? -1 : (l > r);
}
```

Listing 3. Comparing pointers that do not point to the same aggregate or union is undefined behavior so the compiler is free to return anything from this function (sys/kern/subr_hibernate.c in the OpenBSD kernel)

The code shown in these examples was fixed up to this day [6], [9], [13] but the risk of existing code triggering uncatched UB optimizations still persists.

To address these issues the research community created solutions that tackle the problem from different angles. One approach was to introduce new compiler improvements that would catch undefined behaviors either at compile-time or at run-time. However such endeavours could not provide the expected results.

On one hand, generating reports for all undefined behaviors at compile-time is undecidable [21]. Moreover, generating such reports is unuseful in specific cases. Listing 4, for example, could generate reports such as:

- pointer a may originate from a non-integral or non-void pointer
- pointer a may be NULL
- variable b may be uninitialized

```
void foo(int *a, int b) {
  *a = b;
}
```

Listing 4. Code that may report false undefined behavior

This is the case because the internal representation of the compiler may not have enough context to report only the useful information about undefined behaviors and because the compiler cannot understand the intention of the programmer when issuing an UB optimization. In this context, to issue UB optimizations is paradoxical. The compiler does not have the context to find and report undefined behaviors, but it uses undefined behaviors in order to generate code transformations [24].

On the other hand, catching undefined behavior at run-time proves to be an incomplete approach. The run-time checker would need to visit all the states of the program in order to ensure that no undefined behavior is triggered. To catch all states that may contain undefined behavior we need to run the checker for as long as it requires, which may not be desirable in most cases because it may take too much time. Checkers for this task are IOC [17], UBsan [14] and various compiler flags such as GCC's -ftrapv and Clang's -fcatch-undefined-behavior.

Another approach for run-time checking is to compare the unoptimized code with the optimized code generated by the compiler. However program equivalence is undecidable [27]. Also, decompilation might be used to compute the semantic distance between the original C code and the decompiled optimized assembly code. Doing so we could spot the introduced UB optimizations and delete them later. However decompilation is a hard problem in general [16] because of type erasure.

Besides the introduction of compiler improvements, another solution would be to issue additions to the standard that would provide more robustness to the definition of undefined behavior. At the moment, state-of-the-art compilers, such as GCC and Clang/LLVM, take a liberal view of the standard and interpret it in a way that allows them to push various dangerous optimizations. The opposite view is the constructivist one, where the compiler implementations construct a robust definition of undefined behavior, even if the standard imposes no strong requirements. Until the standard makes it clear what approach it would take in the future, implementations and developers need to decide their approach based on the loose definition provided in the standard.

B. Programmer intentionality

To issue UB optimizations the compiler is required to have knowledge of the programmer's intention in order to

generate relevant code, i.e. code that is equivalent to the expectations of the programmer, not to the unsteady definition of undefined behavior presented in the standard. This is a complicated task because intention detection is a hard problem in psychology [20].

Given this problem, the safest thing the compiler can do in this case is not to reason about intentions in any way. Doing this, the risk of losing programmer intentionality is lost.

For code that is free of undefined behavior this problem is not relevant as the compiler is expected to generate code that preserves the intention of the programmer. Here, the compiler is free to do whatever code transformations that increase the performance of the system and that preserve the semantics of the code.

However, most real-life projects make use of non-trivial code constructs that trigger undefined behavior in order to help the programmer communicate various intentions [23], [30]. Code transformations in this case introduce more unwanted consequences that expected results.

III. Related Work

Little work has been done in the area of detecting the performance speedup of UB optimizations in real-life software projects. Wang et al. [28] and Ertl [19] provide metrics for this class of optimizations based on SPECint benchmark.

Wang et al. state that they observed a decrease in performance of 7.2% with GCC and 9.0% with Clang for 456.hammer and 6.3% with GCC and 11.8% with Clang for 462.libquantum. The experiments were conducted with UB optimizations turned off.

Ertl states that with Clang-3.1 and UB optimizations turned on the speedup factor is 1.017 for SPECint 2006. Furthermore, for a specific class of programs, i.e. Jon Bentley's traveling salesman problem, the speedup factor can reach values greater than 2.7 if the programmer issues source-level optimizations by hand, surpassing the UB optimizations issued by the compiler.

IV. Research Plan

Given the little research done in the field of analysing the performance of UB optimizations, this study aims to provide insights of the performance of these optimizations on a specific class of software applications, i.e. operating systems.

The first step of our work is to filter out all undefined behavior instances presented in the standard and focus on the undefined behaviors that present a potential for being used in compiler optimizations. Our filtering strategy is based on the assumption that all undefined behaviors that conflict with the intentionality of the programmer shall not be used to issue code optimizations.

Then we either modify the compiler implementation or use compiler-specific flags to turn off these optimizations. A preliminary list of such undefined behaviors extracted from the standard [22] is:

- An arithmetic operation is invalid (such as division or modulus by 0) or produces a result that cannot be represented in the space provided (such as overflow or underflow) (\$3.3).
- An invalid array reference, null pointer reference, or reference to an object declared with automatic storage duration in a terminated block occurs (\$3.3.3.2).
- A pointer is converted to other than an integral or pointer type (\$3.3.4).

The first undefined behavior could lead to code being eliminated if the compiler detects that the arithmetic operation is incompatible with the standard [28]. The second undefined behavior could discard security checks for NULL pointers [4] and the third undefined behavior could break manual optimizations on floating point numbers [25].

To analyze the role of these optimizations in real-life software, we take a self-contained operating system with focus on robustness and security, i.e. OpenBSD, and compile it on one hand with UB optimizations turned on and on the other hand with UB optimizations turned off. After this stage, the result will be two comparison candidates which will be tested against various benchmarks that will highlight the advantages and disadvantages of the UB optimizations.

Furthermore, we analyze the role of UB optimizations in the various hardware architectures that OpenBSD supports [2]. We suspect that there are hardware setups on which undefined behaviors play a bigger role in compiler optimizations. At the same time, we want to see how the compiler treats robustness and security for each hardware architecture.

At this moment we do not know which components of the systems will be modified when UB optimizations are turned off so we cannot provide the benchmarks that we intend to use. However after we get this information, we plan to create a modification map that will help us visualize the components with the highest rate of modification. After this step, what we will do is to provide benchmarks that will focus on those specific components.

The final result will be a fine grained comparison between the two test candidates that will focus on one hand of speed and performance and on the other hand on robustness and security.

V. Conclusions

The definition of undefined behavior is used by compiler implementations to issue aggressive optimizations. We argue that this class of optimization is very dangerous as it conflicts with programmer intentionality and with a robust definition of code semantics. In this study analyze the performance impact of undefined behavior optimizations in real-life software projects. By doing so we evaluate if the advantages of issuing UB optimizations surpass the security risks that they introduce. In order to do this we take a robust and secure implementation of operating system, i.e. OpenBSD, and compare the system that contains UB optimizations with the same system without UB optimizations. The comparison is done on multiple hardware architectures to inspect what role UB optimizations have for various hardware setups.

REFERENCES

[1] OpenBSD. https://www.openbsd.org/, last visited Friday 23rd December, 2022.

[2] OpenBSD platforms. https://www.openbsd.org/plat.html, last visited Friday 23rd December, 2022.

[3] cert/cc vulnerability note vu162289, apr 2008. https://www.kb.cert.org/vuls/id/162289/, last visited Friday 23rd December, 2022.

[4] Add -fno-delete-null-pointer-checks to GCC CFLAGS, July 2009. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a3ca86aea507904148870946d599e07a340b39bf, last visited Friday 23rd December, 2022.

[5] Fun with NULL pointers, part 1, July 2009. https://lwn.net/Articles/342330/, last visited Friday 23rd December, 2022.

[6] Solution for UB from listing 1, Feb 2012. https://github.com/torvalds/linux/commit/e87c5e35a92e045de75fb6ae9846a38bdd0f92bd, last visited Friday 23rd December, 2022.

[7] BORINGCC, Dec 2015. https://groups.google.com/g/boring-crypto/c/48qa1kWignU/m/o8GGp2K1DAAJ, last visited Friday 23rd December, 2022.

[8] Undefined behavior and fermat's last theorem, March 2015. https://web.archive.org/web/20201108094235/https://kukuruku.co/post/undefined-behavior-and-fermats-last-theorem/, last visited Friday 23rd December, 2022.

[9] Solution for UB from listing 3, Aug 2016. https://marc.info/?l=openbsd-tech&m=147263379303192&w=2, last visited Friday 23rd December, 2022.

[10] Gcc 10.1 compiler optimization benchmarks, May 2020. https://www.phoronix.com/news/GCC-10.1-Compiler-Optimizations, last visited Friday 23rd December, 2022.

[11] Gcc 11 compiler performance benchmarks with various optimization levels, lto, June 2021. https://www.phoronix.com/review/gcc11-rocket-opts, last visited Friday 23rd December, 2022.

[12] Cve records on undefined behavior vulnerabilities, 2022. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=undefined+behavior, last visited Friday 23rd December, 2022.

[13] Solution for UB from listing 2, Jan 2022. https://marc.info/?l=openbsd-tech&m=164332561831177&w=2, last visited Friday 23rd December, 2022.

[14] Undefined behavior sanitizer, 2022. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, last visited Friday 23rd December, 2022.

[15] Raymond Chen. Undefined behavior can result in time travel, June 2014. https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633, last visited Friday 23rd December, 2022.

[16] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[17] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–29, 2015.

[18] Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optimization. In *2015 IEEE Security and Privacy Workshops*, pages 73–87. IEEE, 2015.

[19] M Anton Ertl. What every compiler writer should know about programmers or optimization based on undefined behaviour hurts performance. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, 2015.

[20] Peter M Gollwitzer. Goal achievement: The role of intentions. *European review of social psychology*, 4(1):141–185, 1993.

[21] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 336–345, 2015.

[22] Programming languages — C. Standard, International Organization for Standardization, Dec 1990.

[23] Stephen Kell. Some were meant for c: the endurance of an unmanageable language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 229–245, 2017.

[24] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. Taming undefined behavior in llvm. *ACM SIGPLAN Notices*, 52(6):633–647, 2017.

[25] Chris Lomont. Fast inverse square root. *Tech-315 nical Report*, 32, 2003.

[26] John Regehr. Proposal for a friendly dialect of c, Aug 2014. https://blog.regehr.org/archives/1180, last visited Friday 23rd December, 2022.

[27] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

[28] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 1–7, 2012.

[29] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275, 2013.

[30] Victor Yodaiken. How iso c became unusable for operating systems development. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, pages 84–90, 2021.