

Universitatea POLITEHNICA din București
Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare

Universidade de Lisboa
Instituto Superior Técnico *



LUCRARE DE DISERTAȚIE

Impactul de performanță al optimizărilor bazate
pe comportament nedefinit în C/C++

Conducători Științifici:

Conf. Dr. Ing. Răzvan Deaconescu
Prof. Nuno Lopes *

Author:

Lucian Popescu

București, 2024

University POLITEHNICA of Bucharest
Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

Universidade de Lisboa
Instituto Superior Técnico *



MASTER THESIS

Performance Impact of Undefined Behavior Optimizations in C/C++

Scientific Advisers:

Conf. Dr. Ing. Răzvan Deaconescu
Prof. Nuno Lopes *

Author:

Lucian Popescu

Bucharest, 2024

Nuno and Răzvan offered invaluable support throughout my Master's years. Thank you for your patience and guidance!

Ana, my girlfriend, has been a constant source of encouragement and understanding. Thank you for being there!

Adrian Dorbică and Lucian Mogoșanu were always welcoming to my ideas and thoughts. Thank you!

I also want to thank Maxim Kuvyrkov from Linaro and to Oracle Research for offering us hardware resources and for offering us a grant worth €44k in cloud credits!

Abstract

Undefined behavior (UB) contributes to many serious problems, including security vulnerabilities. However, state-of-the-art compilers, such as Clang/LLVM, rely on UB to issue program optimizations. Ideally, compilers should not rely on such dangerous behavior while optimizing programs. But given the big corpus of programs whose main goal is performance, understanding the improvement that UB optimizations bring becomes an important problem.

In this work, we analyze the impact of 19 UB optimizations on 22 real-world C/C++ programs collected using Phoronix Test Suite (PTS) that contain over 125 performance tests. In 125 cases the absence of UB optimizations causes performance degradations and in 42 cases it results in performance improvements. To recover the lost performance we used Link Time Optimizations (LTO). The number of degradations is reduced from 125 to 43 and the number of improvements is increased from 42 to 115. We also analyze theoretical performance recovery methods that focus on run-time checks and improved alias analysis. Our results show that it is possible to stop using UB in compiler optimizations because part of the same performance can be achieved using other methods such as LTO, run-time checks and improved alias analysis.

As part of this project we implemented 13 flags out of 19 that we used and we added 1 new benchmark in PTS, i.e. Z3 which is a theorem prover. Then, we discovered 1 functional bug and opened 2 performance bugs in LLVM. Finally, we discovered 1 security bug in the john-the-ripper benchmark using a flag from our suite.

Contents

| | |
|--|-----------|
| Acknowledgements | i |
| Abstract | ii |
| 1 Introduction | 1 |
| 1.1 Contributions | 5 |
| 1.2 Outline | 6 |
| 2 Background | 7 |
| 2.1 Undefined Behavior | 7 |
| 2.2 Intermediate Representation | 9 |
| 2.3 LLVM's Representation of Undefined Behavior | 10 |
| 2.4 Examples of UB Optimizations | 12 |
| 2.5 Summary | 13 |
| 3 Related Work | 14 |
| 3.1 Security Impact of UB | 14 |
| 3.2 Performance Impact of UB | 15 |
| 3.3 Summary | 16 |
| 4 Flags used for controlling UB optimizations | 17 |
| 4.1 Description of the flags | 18 |
| 4.1.1 Arithmetic Operation | 18 |
| 4.1.2 Pointers | 19 |
| 4.1.3 Builtins | 20 |
| 4.1.4 Functions and Loops | 21 |
| 4.1.5 Values and Conversions | 21 |
| 4.1.6 Alias Analysis | 22 |
| 4.1.7 Summary of Intermediate Representation changes | 22 |
| 4.2 Implementation of the flags | 23 |
| 4.2.1 Implementation | 23 |
| 4.2.2 LLVM bugs and Alive2 modifications | 26 |
| 4.3 Summary | 27 |
| 5 Benchmarks | 28 |
| 5.1 Benchmark suite | 28 |
| 5.2 Benchmark infrastructure | 30 |
| 5.3 Bugs discovered in Phoronix Test Suite | 32 |
| 5.4 Summary | 33 |
| 6 Results | 34 |
| 6.1 Performance Impact | 34 |

| | | |
|----------|---|-----------|
| 6.2 | Binary Size Impact | 40 |
| 6.3 | Summary | 43 |
| 7 | Performance impact analysis | 45 |
| 7.1 | -fdrop-align-attr on simdjson | 47 |
| 7.2 | -fdrop-inbounds-from-gep + -disable-oob-analysis on jpegxl | 49 |
| 7.3 | -fcheck-div-rem-overflow on jpegxl | 51 |
| 7.4 | -fconstrain-shift-value on compress-zstd | 53 |
| 7.5 | -zero-uninit-loads on john-the-ripper | 53 |
| 7.6 | -disable-object-based-analysis on espeak | 55 |
| 7.7 | Discussion | 56 |
| 8 | Conclusion and Future Work | 58 |
| 8.1 | Future Work | 58 |

List of Figures

| | | |
|-----|---|----|
| 2.1 | Example of using <i>undef</i> in LLVM optimizations | 11 |
| 5.1 | Run-time in seconds of each benchmark present in our suite, the total run-time is greater than 7 hours | 30 |
| 6.1 | UB flags and the number of benchmarks per impact interval. Impact is divided into severe degradations ($< -5\%$), moderate degradations ($[-5\%, -2\%)$ and improvements ($> +2\%$). The vertical line divides the dataset in half | 35 |
| 6.2 | Impact distribution per flag in non-LTO and LTO modes. | 38 |
| 6.3 | Quantitative performance impact of each benchmark in non-LTO and LTO modes | 39 |
| 6.4 | Comparison of three categories of flags: which add checks in the IR, which delete UB constructs from the IR, which control the optimizer | 40 |
| 6.5 | Detailed view of the impact that each flag has on each performance test from our suite | 41 |
| 6.6 | Detailed view of the impact that each flag has on each performance test from our suite when compiled with Link Time Optimizations | 42 |
| 6.7 | Distribution of binary size impact for non-LTO and LTO benchmarks. Positive impact means that the binary size decreases and negative impact means that the binary size increases. | 43 |
| 6.8 | Changes in the binary size of the benchmark based on UB flag. Red means binary size increase, green means binary size decrease, white means no binary size modification | 44 |
| 7.1 | Number of benchmarks impacted either positively or negatively by each flag category presented in Table 4.1 | 45 |
| 7.2 | Instruction Decode Queue statistics comparison between 16-bits and 32-bits aligned loop. The number of different types of cycles decreases for 32-bits loops while the uops throughput remains the same. | 52 |

List of Tables

| | | |
|-----|--|----|
| 4.1 | Categories of UB controlled by flags | 18 |
| 4.2 | Flags that add checks, delete UB constructs from IR and directly control the optimizer | 22 |
| 5.1 | Benchmarks used in our experiments | 29 |
| 7.1 | Categories of performance modifications. Each performance modification is caused by a combination between a benchmark and a flag. | 46 |
| 7.2 | LLVM optimization passes that create the performance modifications. Only modifications from the "Recoverable Missed Opportunities" category are presented. . | 57 |

List of Listings

| | | |
|------|--|----|
| 1.1 | <code>srandomdev(3)</code> version which contains an uninitialized read from junk | 2 |
| 1.2 | <code>srandomdev(3)</code> compiled with Clang 18.0.1 | 2 |
| 1.4 | C code containing signed overflow | 3 |
| 1.5 | Equivalent assembly code for Listing 1.4 | 3 |
| 1.6 | Equivalent assembly code for Listing 1.4 if type of parameters is unsigned | 4 |
| 1.7 | A simple C loop that computes the sum of all elements inside an array using a 32 bit induction variable on a 64 bit system | 4 |
| 2.1 | Code which contains an UB triggered by a possible invalid dereference of pointer <code>p</code> | 8 |
| 2.2 | The ternary operator from Listing 2.1 is deleted because the compiler assumes <code>p</code> is valid | 8 |
| 2.3 | Linux 2.6.30 code which dereferences a pointer before checking if it is valid | 9 |
| 4.1 | Equivalent C code for the checks added before division and remainder | 19 |
| 4.2 | C code for <code>is_sb_aq_enabled</code> | 24 |
| 4.3 | Alive2 output on <code>is_sb_aq_enabled</code> | 24 |
| 4.4 | OpenSSL code where Alive2 cannot analyse locally-allocated pointers | 27 |
| 5.1 | Benchmarking process described in pseudocode | 32 |
| 7.1 | Name of the function where the performance degradation takes place | 47 |
| 7.2 | Diff on the <i>run</i> function between baseline and <i>-fdrop-align-attr</i> | 48 |
| 7.3 | Hoisting of the <code>%0</code> instruction from <code>while.body</code> to entry | 49 |
| 7.4 | Simplified version of IR that causes vectorization problems | 50 |
| 7.5 | Equivalent C code for the checks added before division and remainder | 51 |
| 7.8 | Uninitialized variable <code>ctx</code> used inside <code>scan_central_index</code> | 54 |
| 7.9 | <code>espeak-1.0.7</code> loop where performance degradation happens | 55 |
| 7.10 | Optimal IR for loop in Listing 7.9 | 56 |
| 7.11 | Unoptimal IR for loop in Listing 7.9 | 56 |

Chapter 1

Introduction

Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose.²

Undefined behavior is the source of many security vulnerabilities [20] and of various compiler optimizations [34]. The contradiction between security and performance stems from the diverging views of the users of UB. On one hand, programmers expect that their code will behave consistently in every environment [59, 45, 17, 50]. On the other hand, compiler writers take advantage of UB to create optimizations that make use of the best features of particular hardware platforms [18].

Projects such as Chromium [61], Firefox [43], Linux [4], the Python Interpreter [60] and Mono [11] were affected throughout the years by various types of UB. Consequently, much work has focused on detecting and analysing UB in languages such as C/C++ in order to reduce the security impact. Fuzzers, sanitizers [10, 46, 48] and code instrumentation tools have been created to enable more secure code in terms of UB. Furthermore, compiler options have been created to ensure that specific UBs are transformed in defined behavior, e.g. signed overflow is UB but `-fwrapv` transform signed overflow into two's complement wrap-around.³

However, few studies have inspected the performance impact of UB optimizations. Internally, UB optimizations work on a *reductio ad absurdum* principle. Compilers presume that UB can never happen in a program, because that would be illegal, and create assumptions which are used to generate better code. However, not all assumptions are inherently unsafe, e.g. lead to optimizations that delete security checks. Safe UB assumptions can prove loop termination properties or prove mathematical properties.

Furthermore, current literature only focuses on the performance impact of UB optimizations on micro-benchmarks. While they are valuable to showcase specific optimizations on a small and precise code fragment, they cannot always represent real-world usage scenarios [56]. To fill this gap we present the first study that analyses the impact of 19 UBs on a benchmark suite of 22 real-world C/C++ applications with more than 125 performance tests.

To better illustrate the significance of undefined behavior in both security vulnerabilities and performance optimizations, we provide a couple of examples from each category. First, UB can act as a security weakener when programmers make "use of a nonportable or erroneous program construct or of erroneous data" [26]. One of the most popular vulnerabilities caused by UB is CVE-2013-5180⁴. The `srandomdev(3)` function returns predictable values instead of intended random values because the code contains an uninitialized read.

²<http://catb.org/jargon/html/N/nasal-demons.html>

³<https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html>

⁴<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5180>

Listing 1.1 presents the code that triggers UB. In this implementation, the function assumes that it can use garbage values placed in the uninitialized value junk in order to initialize the random number generator of the system. This assumption may hold with a specific version of a compiler, but poses serious problems when a different compiler that optimizes-out the variable to zero, for example.

Listing 1.1: `srandomdev(3)` version which contains an uninitialized read from junk

```
void
srandomdev(void)
{
    ...
    struct timeval tv;
    unsigned long junk;

    gettimeofday(&tv, NULL);
    random((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
    ...
}
```

For example, `tcc 0.9.27`, which is not an optimizing compiler, successfully generates code as per the intentions of the programmer,¹ i.e., use an uninitialized value as part of the computation in the `random(3)` argument. However, optimizing compilers such as `GCC 14.1` and `Clang 18.1.0` produce code that deletes either partially² or completely³ the `srandom(3)` argument.

Listing 1.2: `srandomdev(3)` compiled with Clang 18.0.1

```
srandomdev():
    sub    rsp, 24
    lea    rdi, [rsp + 8]
    xor    esi, esi
    call   gettimeofday@PLT
    call   getpid@PLT
    add    rsp, 24
    jmp    srandom@PLT
```

Listing 1.3: `srandomdev(3)` compiled with GCC 14.1

```
srandomdev():
    sub    rsp, 24
    xor    esi, esi
    mov    rdi, rsp
    call   gettimeofday
    call   getpid
    mov    rdi, QWORD PTR
    [rsp+8]
    xor    edi, DWORD PTR
    [rsp]
    sal    eax, 16
    xor    edi, eax
    call   srandom
    add    rsp, 24
    ret
```

The code generated by GCC and Clang is presented in Listing 1.2 and Listing 1.3. The two generated functions differ by 5 instructions, mainly because Clang decides to completely delete the operation that involves the `junk` variable. Since in the Clang version the `srandom(3)` function is still called, the register that contains its argument is left in an unknown state.

This has serious security implications as the randomness of the whole system is compromised. Meanwhile, the problem has been solved by using standard C construct for generating random numbers.⁴ Even if the CVE that describes this issue is more than 10 years old, uninitialized

¹<https://godbolt.org/z/qdfP8GqxG>

²<https://godbolt.org/z/4ne7fv46f>

³<https://godbolt.org/z/aPorbKh6z>

⁴<https://github.com/freebsd/freebsd-src/commit/6a762eb23ea5f31e65cfa12602937f39a14e9b0c>

variables are still misused to this day. As part of this work, we have discovered an uninitialized read in the `john-the-ripper` program.¹

Furthermore, security problems caused by UB are not isolated, throughout the years, undefined behavior was a constant presence in the CWE Top Most Dangerous Software Weaknesses.² Moreover, the amount of discussions and technical problems generated by UB in C and C++ projects is not to be ignored. Two simple queries on GitHub reveal that there are more than 5k issues related to UB in the C language³ and more than 11k in the C++ language.⁴

Nevertheless, UB is a valuable resource for compiler optimizations [9, 32, 42]. Compilers cannot prove the inexistence of UB for a given program since this would require complete knowledge of every state of the running program, which is not available at compile time. Nevertheless, if compilers could do that, they could also solve the halting problem, but the halting problem is provably undecidable. Hence, detecting all possible UBs is impossible. Consequently, compilers assume that the program contains only defined behavior which allows them to optimize-out cases that produce UB.

For example, given a pointer dereference `*P`, UB gives the optimizer the ability to assume that `P` is never `NULL`. If `*NULL` would take place anywhere in the program, then the compiler is allowed to reason that the code that leads to the `NULL` dereference must not be reachable, otherwise UB would be triggered. This type of optimizations give the compiler more flexibility since it is not required to diagnose every occurrence of UB and allows it to generate smaller and faster code.

Besides pointer dereferences, UB also affects arithmetic operations with signed numbers since signed overflow is undefined behavior. Listing 1.4 shows a piece of C code that computes the result of an inequality with signed numbers. Since it would be illegal for the additions to overflow, the expression can be simplified to $a < b$. Effectively, this allows signed integers to inherit all properties of natural numbers. Both GCC 14.1⁵ and Clang 18.1⁶ optimize out the addition as presented in Listing 1.5.

Listing 1.4: C code containing signed overflow

```
bool foo(int a, int b, int c) {
    return (a + c) < (b + c);
}
```

Listing 1.5: Equivalent assembly code for Listing 1.4

```
foo(int, int, int):
    cmp     edi, esi
    setl    al
    ret
```

However the same C operation does not hold if we change the type of the parameters to *unsigned*. Unsigned addition is never UB per the C/C++ Standards, it contains wrap-around semantics that will invalidate the additive property mentioned above. Listing 1.6 presents this effect. Both Clang⁷ and GCC⁸ produce code that adds both parts of the inequality before performing the comparison.

¹<https://github.com/openwall/john/pull/5466>

²<https://cwe.mitre.org/top25/archive/>

³GitHub search query: "undefined behavior" language:C

⁴GitHub search query: "undefined behavior" language:C++

⁵<https://godbolt.org/z/49EsoTc9q>

⁶<https://godbolt.org/z/866PKd5TK>

⁷<https://godbolt.org/z/r9Y656bG9>

⁸<https://godbolt.org/z/s159E57ef>

Listing 1.6: Equivalent assembly code for Listing 1.4 if type of parameters is unsigned

```
foo(unsigned int, unsigned int, unsigned int):  
    add    edi, edx  
    add    esi, edx  
    cmp    edi, esi  
    setb   al  
    ret
```

In practice, compilers use signed overflow to optimize memory accesses based on induction variables (IV) inside loops. On current 64 bit machines, `int` size is 32 bits wide for compatibility reasons with older 32 bit machines. This causes problems since `int` is commonly used as IV in C/C++ code. On every memory access that is dependent on the IV, a sign extension from 32 bit to 64 bit is required since addresses are in 64 bit format. However, because sign extension is a costly operation, it would be ideal to get rid of it. To do that, the compiler can use the fact that signed overflow is UB.

In Listing 1.7 we present a simple loop where the IV is 32 bits wide and the limit of IV is 64 bits wide, also the memory addresses are 64 bits wide. In the generated assembly code, IV always needs to be sign extended to 64 bits before accessing any memory, virtually transforming `iv` from `int32_t` to `int64_t`. For this transformation to be correct, the compiler must prove that `iv` can never overflow in this loop or insert extra code that detects the overflow. Since both solutions are costly either because it puts more pressure on the compiler or the generated code, a simpler solution would be to assume that UB will never happen, hence the sign extension is always valid.

Listing 1.7: A simple C loop that computes the sum of all elements inside an array using a 32 bit induction variable on a 64 bit system

```
#include <stdint.h>  
  
int foo(int *x, uint64_t count) {  
    int sum = 0;  
    for (int32_t iv = 0; i < count; ++i)  
        sum += x[iv];  
    return sum;  
}
```

Listing 1.8 shows the generated assembly code for the above function.¹ The IV is placed in the `rcx` register and is treated as a 64 bit value for the whole duration of the loop. However, we compiled the same loop with `-fwrapv`, a compiler flag that forces signed overflow to be defined and to take two's complement wrap-around semantics. The resulted code, presented in Listing 1.8, is 2 instructions longer. The additional instructions, i.e. `"mov ecx, 1"` and `"movsxd rdx, ecx"` deal with the sign extension of IV from 32 bits to 64 bits. For the whole duration of the loop, IV is treated as a 32 bit value and only when needed it is extended to 64 bits.

Consequently, UB managed to reduce the size of the entire function by 2 instructions, i.e. 18% of the function compiled without `-fwrapv`. However, this optimization has other far reaching consequences. The above example was compiled with `-O1` where vectorization is not enabled. When vectorization is enabled in `-O2`, for example, the instruction difference between the two versions is of 12 instructions.²

¹<https://godbolt.org/z/dz4YbvW3n>

²<https://godbolt.org/z/da88TfKWv>

Listing 1.8: Loop from Listing 1.7 compiled without -fwrapv

```

foo(int*, unsigned long):
    test    rsi, rsi
    je      .LBB0_1
    xor     ecx, ecx
    xor     eax, eax
.LBB0_4:
    add     eax, dword ptr
        [rdi + 4*rcx]
    inc     rcx
    cmp     rsi, rcx
    jne     .LBB0_4
    ret
.LBB0_1:
    xor     eax, eax
    ret

```

Listing 1.9: Loop from Listing 1.7 compiled with -fwrapv

```

foo(int*, unsigned long):
    test    rsi, rsi
    je      .LBB0_1
    xor     edx, edx
    mov     ecx, 1
    xor     eax, eax
.LBB0_4:
    add     eax, dword ptr
        [rdi + 4*rdx]
    movsxd  rdx, ecx
    inc     ecx
    cmp     rdx, rsi
    jb      .LBB0_4
    ret
.LBB0_1:
    xor     eax, eax
    ret

```

In summary, UB has been a constant source of security vulnerabilities in the last decades, one example of this is CVE-2013-5180 which deletes security checks in the program when uninitialized variables are used.¹ However, UB also plays an important role in compiler optimizations, enabling both performance improvements and binary size reduction.

1.1 Contributions

Much of the work regarding UB has been focused on the security impact of UB. However, to understand the real consequences of UB, we propose a comprehensive analysis of the performance impact of UB optimizations in C/C++. In this study, using 22 C/C++ real-world applications that contain more than 125 performance tests, we analyze the performance impact and the binary size impact of 19 wide-used and popular UBs. Our study is focused on the Clang/LLVM 16 compiler.

Each UB is controlled by a compiler flag, we cover behaviors such as: signed overflow, NULL pointer dereference, uninitialized loads, etc. As part of this project we implemented 13 compiler flags and we used 6 flags that already existed.

We discovered that in 125 cases the performance decreases and in 42 the performance increases when the flags that disable UB are activated. However to recover the lost performance we used Link Time Optimizations (LTO) that reduce the number of degradations from 125 to 43 and increase the number of improvements from 42 to 115. To further reduce the number of degradations we propose theoretical techniques that add run-time checks or improve the alias analysis subsystem in order to get rid of UB optimizations while keeping maximum performance.

Furthermore, we discovered that lack of UB optimizations increase the binary size by at most +2%, however this number can be reduced by making use of LTO.

Then, we analyzed 6 performance degradations and 1 performance improvement to better understand the impact of UB optimizations. In one case, the performance degradation was caused by a bug in the source code that we reported in the john-the-ripper benchmark. In the rest of 5 cases the performance was caused by the inability of LLVM to speculatively execute instructions, to vectorize loops and to inline functions. The only improvement was caused by

¹<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5180>

a SkyLake specific feature where the uops cache gets better performance if the loop that it decodes is 32 bits aligned instead of 16 bits.

The performance of all the degradations that we analyzed can be recovered either by using LTO or by using theoretical techniques such as run-time checks and improved alias analysis. This results shows that the compiler can generate programs with similar performance without issuing UB optimizations which are dangerous.

1.2 Outline

This work is structured as follows. In Chapter 2 we introduce undefined behavior and how it affects various optimizations inside compilers, focusing mostly on LLVM. Then, in Chapter 3, we present the work that has been previously done in assessing the security and performance impact of UB optimizations. Chapter 4 and Chapter 5 describe the flags that we use for controlling UB optimizations and the benchmark infrastructure we prepared.

Chapter 6 and Chapter 7 present the results of the performance impact of UB optimizations and an analysis of 6 degradations and 1 improvement caused by our flags. Chapter 7 ends with a discussion about the performance recovery for the degradations that we encountered using techniques such as Link Time Optimizations, additional run-time checks and improved alias analysis.

Chapter 2

Background

In this chapter we introduce the concept of undefined behavior and present its relation with the LLVM compiler infrastructure.

2.1 Undefined Behavior

The C and C++ standards introduce UB for describing the behavior of non-portable and erroneous program constructs, or of erroneous data, or of intermediately-valued objects. Compiler implementations are free to ignore the presence of UB completely or to behave in a documented manner. Possible sources of undefined behavior come from language constructs such as signed overflow, NULL pointer dereference, access of an object through a different pointer type, and many other sources.

To determine if a language construct triggers UB, the C and C++ standards use a series of phrases such as:

- the behavior of the program is undefined
- has undefined behavior
- results in undefined behavior
- the behavior is undefined
- have undefined behavior
- is undefined
- result has undefined behavior

However, some UBs are implicit rather than explicit. For example, the current draft of the C++ standard, N4928 [1], contains the following wording in [intro.progress]: "The implementation may assume that any thread will eventually do one of the following:". Then a list of 4 assumptions is listed, but it is not explicitly stated that violating the assumptions leads to UB. Fortunately, there are currently submitted proposals to mitigate against this and make all UB explicit [6].

The C99 standard contains no more than 191 undefined behaviors that do not require specific actions from the compiler. However, since some of them are easy to report, most of the compilers choose to issue a warning or an error [28] when they encounter this type of UB. Examples of such UBs are: unmatched ' or ", identifiers with external and internal linkage, and multiple declarations of the main function.

On the contrary, there are UBs that are hard or impossible to diagnose. They require run-time information that is not available at compile time. For example, an integer value or a pointer cannot be proven to stay in the realm of defined behavior for the whole duration of the program. Consequently, compilers use an *reductio ad absurdum* method to deal with UB. By working on the assumption that the program does not contain UB, the transformation that the compiler employ become simpler in terms of UB checking.

Assuming that the program does not trigger UB gives birth to UB optimizations. The compilers use the fact that the rules of the language cannot be violated to deduce properties about the language constructs. For example, because a pointer has already been dereferenced and after that it is checked against NULL, then the NULL check is redundant and can be deleted. Because it would be UB to dereference a NULL pointer, the compiler assumes that the pointer is already valid and the second check is redundant.

The effects of this type of optimizations range from deletion of dead code which decreases the binary size to improvements in the run time of the program. However, UB optimizations are risky and can render unexpected results [53]. In some cases, code that is intended to act as security checks gets deleted, leading to program vulnerabilities.

A simple, yet effective, example of C code that produces unexpected results because of UB is presented in Listing 2.1.¹ The value_or_fallback function returns the pointed-to value if the pointer is valid, otherwise it returns 42. Furthermore, in the software development life-cycle it is common to add print statements to debug the code. However, the print statements adds UB to our function because it dereferences the pointer without knowing if it is valid.

Listing 2.1: Code which contains an UB triggered by a possible invalid dereference of pointer p

```
int value_or_fallback(int *p)
{
    printf("The value of *p is %d\n", *p);
    return p ? *p : 42;
}
```

As a result the function is optimized into the code presented in Listing 2.2, effectively deleting the NULL pointer check. The compiler observed that the pointer was dereferenced before the NULL pointer check and as a consequence it concluded that the pointer is required to be valid since NULL pointer dereference cannot take place.

Listing 2.2: The ternary operator from Listing 2.1 is deleted because the compiler assumes p is valid

```
int value_or_fallback(int *p)
{
    printf("The value of *p is %d\n", *p);
    return p;
}
```

GCC 14.1² and Clang 18.1³ both delete the check at -O3. To keep the pointer check while also dereferencing the pointer beforehand, both compilers use a flag called *-fno-delete-null-pointer-checks*. Using this flag in combination with -O3 generates the expected NULL pointer check.⁴

This type of pattern, i.e. dereferencing the pointer before it is NULL checked, is common in real-life software as well. The Linux 2.6.30 kernel suffered from a deleted check which left the kernel in an exploitable state [4]. The vulnerable code is presented in Listing 2.3

¹<https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633>

²<https://godbolt.org/z/ETsx44YMz>

³<https://godbolt.org/z/G9E5c13xT>

⁴<https://godbolt.org/z/jhbfznxYr>, <https://godbolt.org/z/nPeTMr5fs>

Listing 2.3: Linux 2.6.30 code which dereferences a pointer before checking if it is valid

```
struct tun_file *tfile = file->private_data;
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
unsigned int mask = 0;

if (!tun)
    return POLLERR;
```

First, the file argument was dereferenced without any NULL pointer check. Then then tun pointer was resulted as part of a possible invalid operation. Finally, after all variables were declared, tun was checked against NULL. Since NULL pointer dereference generates UB and the compiler assumes that UB can never happen, then the compiler concludes that the pointer must be valid since otherwise the code would produce UB. To solve this problem, the *-fno-delete-null-pointer-checks* was used to mitigate against the deletion of the check. We also use this flag in our benchmark experiments.

In summary, we presented the origins of undefined behavior and how some UBs are easy to catch while others are hard or impossible to catch. For example, syntactic UBs are easy to catch, however catching UB related to pointer dereferencing, for example, would require run-time information or code instrumentation that adds overhead to the compiled programs. Since there are many UBs that lie in the second category, compilers assume that UB never happens in order to create transformations that become simpler in terms of UB checking. Consequently, this gives birth to UB optimizations which we demonstrate with a simple example which contains a possibly invalid pointer dereference in Listing 2.1.

Next, we present an introduction to the Intermediate Representation (IR) used by LLVM and how UB is modeled in this IR.

2.2 Intermediate Representation

To apply optimizations that are based on UB, or any optimizations whatsoever, compilers use a set of intermediate languages to translate the source code into machine code. Multiple intermediate languages have been proposed, usually they are referred as Intermediate Representations (IR) [12, 51]. The existence of IR in current compilers allows great levels of portability. Usually, a compiler is divided into front-end, which is responsible for transforming source code into IR, middle-end, which applies optimizations on the IR, and back-end, which translates the IR into machine code. In this context, a new language can be added into the front-end or a new hardware architecture can be added into the back-end without losing any optimization from the middle-end.

LLVM employs three different IRs: the LLVM IR,¹ where most of the target-independent optimizations such as inlining and global value numbering are taking place, the SelectionDAG,² which provides support for instruction selection and scheduling, and the MachineIR,³ which contains machine instructions and target-specific optimizations.

Currently, LLVM IR uses the Static Single Assignment (SSA) form [21]. In SSA representation each variable can be assigned only once which enables efficient implementations of various types of analyses. Since each variable can be assigned only once, different versions need to be created on variable updates, depending on the basic blocks they were assigned in (a basic block is a sequence of instructions with a single entry and a single exit). This causes problems when we

¹<https://llvm.org/docs/LangRef.html>

²<https://llvm.org/docs/CodeGenerator.html#introduction-to-selectiondags>

³<https://llvm.org/docs/MIRLangRef.html>

refer to variables from previous basic blocks. For this, the ϕ node solves this problem by taking into account previous values of the variable and choosing the suitable value.

The code below presents a simple C if statement, that conditionally initializes a variable, which is translated into LLVM IR's SSA form. The phi instruction in the `cont` basic block chooses the value that will be placed in `a` based on the control flow of the program.

```

int a;
if (c)
    a = 0;
else
    a = 1;
return a;

entry:
    br %c, %ctrue, %cfalse
ctrue:
    br %cont
cfalse:
    br %cont
cont:
    %a = phi [0, %ctrue],
             [1, %cfalse]
    ret i32 %a

```

Other representations have been proposed to extend SSA, including Static Single Information (SSI) [13] which in addition to ϕ nodes adds σ nodes at the end of each basic block indicating where the value goes, and Gated Single Assignment (GSA) [40, 52] which replaces ϕ nodes with functions that represent loops and conditional branches. Another variant of SSA is MemorySSA which enables identification of redundant loads and reorganization of memory-related code.

Recently, Horn clauses have been proposed as an IR for compilers, as an alternative to SSA, since despite leading to duplicated analysis efforts, they solve most problems associated with SSA: path obliviousness, forward bias, name management, etc [27].

2.3 LLVM's Representation of Undefined Behavior

Since the LLVM Intermediate Representation (IR) needs to work with low-level unsafe procedures such as pointer dereferencing, memory reading or arithmetic operations, it also needs to represent the concept of undefined behavior.

The LLVM IR has two categories of UB: immediate UB and deferred UB. They are used to encode the severity of unsafe operations. On one hand, immediate UB is a strong form of UB that is propagated directly from the target languages, in our case C and C++. The operations that it encapsulates range from division by zero which generate a processor trap or dereferencing an invalid pointer which causes possible memory corruption. Since immediate UB has severe implications on the target machine, compilers are allowed to issue defensive mechanisms against this. As a consequence, the entire code path that leads to immediate UB can be deleted.

On the other hand, LLVM also encapsulates a weaker form of UB which is called deferred UB. This type of UB refers to operations that produce non-deterministic values but otherwise are safe to execute. Examples of deferred UB are: overflowing a signed integer or reading uninitialized memory. Deferred UB is critical for allowing speculative execution of operations, among other optimizations, since it allows the free movement of operations that produce possibly undefined values.

Furthermore, deferred UB comes in two forms: *undef* and *poison*. The *undef* value corresponds to an arbitrary bit pattern of a given type and may return a different value each time it is used. This allows transformations to choose the most profitable value based on the operation where *undef* is used.

An example of *undef*'s usage in LLVM optimizations is presented in Figure 2.1. For each operation that uses the `x` variable, LLVM chooses the most profitable value, for the multiplication

it chooses $x=0$ and for the bitwise or it uses -1 . Finally, this results in a and b holding constant values which are folded into the return value.

Before `undef` was added in LLVM [5], all `undef` values were converted to 0 by the `mem2reg` pass which resulted in various missed optimization opportunities. The example from Figure 2.1 would only transform the multiplication into 0 and the bitwise or would result in y . Since the return value is not anymore a constant value, further optimizations such as constant folding are not possible in the callers of this function.

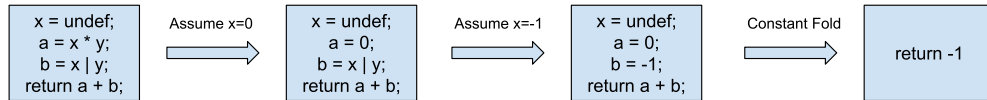


Figure 2.1: Example of using *undef* in LLVM optimizations

However, there are transformations that are not valid when *undef* exists. For example, it is common to convert a multiplication instruction such as `%y = mul %x, 2` to `%y = add %x, %x` on architectures where multiplication is more expensive than addition. However, the transformation is not valid because in the first instruction `%y` will always hold an even number, but in the second instruction `%y` will contain an arbitrarily bit pattern. This is the direct effect of the fact that *undef* is not required to return the same bit pattern on each usage.

On the bright side, the ability to return a different value on every usage of *undef* means that there is no need to have a hardware-backed source of non-determinism. The compiler is free to assume that *undef* can hold any value that is convenient for a particular transformation.

The second type of deferred UB, i.e. *poison*, is a more powerful form UB that propagates through the program and triggers immediate UB if it reaches a side-effecting operation. Compared to *undef*, *poison* is not bound to an arbitrarily value of a given type, this allows certain optimizations such as sign-extension elimination and arbitrary speculation of instructions.

For example, in C/C++ the expressions $a + b > a$ and $b > 0$ are equivalent because signed overflow is undefined, assuming that a and b are signed integer values. In LLVM, the first expression translates to:

```
%add = add %a, %b
%cmp = icmp sgt %add, %a
```

However, it is not valid to transform the two instructions to:

```
%cmp = icmp sgt %b, 0
```

because the `add` instruction would wrap around on overflow, effectively acting as an unsigned integer.

Moreover, the optimization cannot be legalized by defining a version of `add` that returns *undef* on overflow. Suppose that $a = \text{INT_MAX}$ and $b = 1$, then the addition overflows and the expression becomes $\text{undef} > \text{INT_MAX}$.¹ Since there is no bit pattern that *undef* returns which is greater than INT_MAX , the results of the expression is false. However, the optimized version, i.e. $b > 0$, becomes invalid because it would return true.

¹<https://alive2.llvm.org/ce/z/R4XICN>

To solve this problem, *poison* was added. If we define a version of the add instruction that returns *poison* on overflow then the optimization becomes valid.¹ Since *poison* is not tied to a specific bit pattern of the given type and *poison* can propagate in the cmp instruction, then the result of the cmp instruction is *poison* as well if the addition overflows. Then, to make the transformation valid, *poison* can be materialized into `%cmp = icmp sgt %b, 0`.²

Furthermore, speculative execution is another example that showcases the limitations of *undef* compared to *poison*. The following code snippet:

```
if (k != 0) {
    while (c) {
        use(1/k);
    }
}
```

can be transformed to:

```
if (k != 0) {
    m = 1/k;
    while (c) {
        use(m);
    }
}
```

Because `1/k` is loop invariant, it is natural to assume that it can safely be moved outside the loop to save computational cycles. This operation can be considered safe since `k` is checked against 0 before the division. However, if `k` is *undef* the check can pass and the division would trigger immediate UB since different bit patterns can be chosen for the two usages. *poison* allows this transformation to be correct because if `k` is *poison* then `m` will be *poison* as well instead of generating immediate UB.

LLVM also introduced an instruction that stops the propagation of *undef* and *poison*, called freeze. Calls to freeze return an arbitrarily, but fixed, value of the given type if the argument is *undef* or *poison*. Otherwise, the instruction is a no-op since it returns the input argument. freeze can also be used in the speculative execution example to make the movement of `1/k` valid. However, that would be unoptimal since further optimization opportunities would effectively be killed:

```
fk = freeze k
if (fk != 0) {
    m = 1/fk;
    while (c) {
        use(m);
    }
}
```

2.4 Examples of UB Optimizations

To get a better feel about the types of source code changes UB optimizations might imply, we provide in this section 5 examples of UB optimizations that have direct impact at the level of the C/C++ language. All examples are focused on signed overflow for simplicity. The examples are extracted from [9] and were presented by Ian Lance Taylor in 2006.

The examples are listed below:

¹<https://alive2.llvm.org/ce/z/dq8t36>

²<https://alive2.llvm.org/ce/z/gmpCtf>

1. Fold $(-(A / B))$ to $(A / -B)$ or $(-A / B)$. It is especially profitable if A or B are easily negated.
2. Fold $((A + C1) \text{ cmp } (B + C2))$ to $(A \text{ cmp } (B + C1 + C2))$ where $C1$ and $C2$ are constants, and likewise for $-$. UB is able to preserve the mathematical properties of inequalities.
3. Fold simple comparisons like $(X - C > X)$ to false.
4. Assume that if $X \geq 0$ and $Y \geq 0$, then $X + Y \neq 0$ exactly when $X \neq 0$ or $Y \neq 0$. This is used primarily when simplifying comparisons against NULL. This assumes that the memory address space cannot wrap around.
5. Compute the number of iterations in a loop of the form $X = \text{INIT}; X \text{ cmp } \text{END}; X \text{ OP} = Y$ by assuming that X will not wrap around. If X would wrap around, the loop might become infinite.

Consequently, signed overflow UB reaches a wide set of optimizations, ranging from rewriting of division operations, proving mathematical properties on signed comparisons and computing the number of iterations in a loop.

2.5 Summary

In summary, we introduced the concept of UB as described in the C/C++ Standards and its implications on the Intermediate Representation (IR) and optimizations of LLVM. Then, we present a handful of optimizations that can be done both at the level of the IR and at the level of the source code which are based on UB.

LLVM makes use of two types of UB to optimize code. It uses immediate UB to represent severe operations such as dereferencing invalid pointers, then it uses a weaker form of UB, called deferred UB, to represent operations that produce non-deterministic values but which are safe to execute.

The deferred UB is as well divided into *undef* and *poison*. *undef* corresponds to an arbitrarily bit pattern of the given type, while *poison* corresponds to a special value that is not part of the given type and is able to propagate through the program. They are used to enable optimizations such as peephole optimizations or speculative execution. Additionally, LLVM also introduced a freeze instruction to stop the propagation of *undef* and *poison*.

Next, we present the literature focused on understanding the security and performance impact of UB optimizations.

Chapter 3

Related Work

As presented in Chapter 2, undefined behavior affects the design of both the source languages, in our case C/C++, and of the compiler IR. For source languages, much work has been focused on efficiently finding and fixing UB using fuzzers [35, 23] or sanitizers [62, 30]. Then, fuzzing techniques were also applied on compilers to make sure that they handle consistently programs that contain UB [33, 58]. Finally, efforts have been put into defining solid semantics for handling UB in compilers [34] and into assuring the correctness of the compiler transformations [37, 36].

As such, the security and correctness implications of UB have been thoughtfully studied. However, few studies focus on the performance implications of UB in both the source languages and the compiler IR [53, 25]. Yet, to measure the performance, they use synthetic benchmarks which are not necessary representative for real-life workloads. In this context, we offer the first detailed examination of the impact of UB optimizations on non-synthetic workloads.

Furthermore, C99 Rationale [8] places UB in close relation with optimizations and current versions of the C/C++ Standards plan changes to reduce the number of UBs, e.g. signed overflow becomes legal in C23 [2] and erroneous behavior partially replaces UB in C++26 [3]. In this context, understanding the performance impact of UB optimizations can become a decision mechanism for further changes regarding UB.

The rest of this chapter presents how UB has affected software systems and compilers over the years, focusing on the security bugs they introduce and the applied mitigations. Then, we use the work of Lee et al. [34] that defines the semantics of UB in LLVM to prepare the stage for our performance study.

3.1 Security Impact of UB

Since UB is the source of security bugs and miscompilations [57, 53], most of the efforts have been put into analyzing and reducing the amount of UB in C/C++ codebases [24, 47, 54] and into the consistent handling of UB in compiler implementations [36, 37, 33, 44].

A recent study [57] presents a number of 120 security bugs which are caused by UB, among other cases. 59 of them were caused because the compiler deleted security related code. Moreover, the authors present that from 2007 until 2021, the number of security bugs caused by UB is almost constant. Also, in their study, a survey on the understanding of UB implication by 62 C programmers with an average of 7 years of experience is conducted. Even if 90% of them are aware of UB, only 48% of them know the implications of UB on compiler optimizations and 42% of them are aware of the security bugs UB can lead to.

In this context, the importance of understanding and controlling the effects of UB increased.

Dietz et al. [22] present a study on the impact of integer overflow in C/C++, which over the years was a constant presence in MITRE’s CWE Top 25 Most Dangerous Software Weaknesses¹ and caused security breaches that allowed arbitrarily code execution.² However, Dietz et al. found more than 200 distinct locations in the SPEC CINT2000 benchmark suite where integer overflow was intentionally used. This makes the detection of this type of UB harder because compilers have no method of knowing if the program uses intentional or unintentional integer overflow.

Various fuzzing techniques have been proposed to find and understand compiler bugs based on UB [15, 33, 44, 58]. Le et al. [33] discovered more than 100 bugs in GCC and LLVM using a fuzzing technique that takes into account both the program generated by the fuzzer and a the input of the program. Yang et al. [58] reported more than 300 compiler bugs using a fuzzing technique that takes into account only the fuzzed programs. Finally, Regehr et al. [44] created a tool to reduce the test cases that trigger compilation bugs and Chen et al. [15] provided methods for selecting the most relevant issues reported by compiler fuzzers.

Besides fuzzers, sanitizers are also used to detect problems related to UB [46, 48, 10]. On one hand, Serebryany et al. [46] discovered more than 300 bugs in Chromium, Firefox and LLVM using AddressSanitizer (Asan). Asan uses a custom memory allocator and code instrumentation to detect out-of-bounds accesses in objects allocated on heap and stack. On the other hand, Stepanov et al. [48] found more than 500 bugs in a Google server-side application with +100 MLOC using MemorySanitizer (Msan). Msan uses both code instrumentation and a shadow memory to catch uninitialized reads. Finally, UBSan [10] uses code instrumentation for catching a wider class of UB: overflowing bitwise shifts, signed integer overflow, etc.

3.2 Performance Impact of UB

We use the work of Lee et al. [34] that described the semantics of UB in LLVM as a starting point for our performance study. In their work, Lee et al. present a tradeoff in the design of UB in the LLVM IR. On one hand, UB is required to exist in order to reflect inherently unsafe operation such as memory accesses in languages such as C/C++. On the other hand, its design must be not too over-constraining with regards to optimizations since it would prohibit desirable transformations. For example, even if reading from uninitialized memory is technically UB, it does not have severe implications on the program, hence an optimization that removes the read is desirable.

Furthermore, the authors presented and formalized an important type of UB which is heavily used in LLVM’s transformation, i.e. deferred UB, which take the form of two special LLVM values: *undef* and *poison*. Using *undef*, LLVM can choose the most profitable bit pattern for a variable that contains UB, while *poison* can propagate the UB to specific points in the program where an optimization would be the most profitable. However, it is not clear what is the performance impact of deferred UB on real-life programs and what transformations inside LLVM benefit the most from this type of UB.

To fill this gap, Wang et al. [53] and Ertl [25] provided performance numbers that demonstrate the impact of UB optimizations on synthetic workloads. The first study measured the performance of GCC 4.7 and Clang 3.1 on the SPECint 2006 benchmark with three flags that disable UB optimizations: `-fwrapv`, `-fno-delete-null-pointer-checks` and `-fno-strict-aliasing`. They report performance modifications on only two benchmarks, i.e. 456.hmmmer and 462.quantlib. In both cases, the flags produce performance degradations between -6.3% and -11% and the root cause lies in the inability of the compilers to optimize critical loops. In the first case, `-fwrapv` disables the conversion of the loop induction variable from 32 to 64 bits, while in the second case,

¹<https://cwe.mitre.org/top25/archive/>

²<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753>, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0639>

-fno-strict-aliasing disables movement of loop invariants outside the loop.

The next study explores the performance impact of UB optimization on a set of programs that implement the Traveling-Salesman Problem (TSP) extracted from the Writing Efficient Programs book [14]. Each program is compiled with various versions of GCC and version 3.5 of Clang. For GCC, 7 flags that disable UB optimizations were enabled, while for Clang only 3 were enabled, i.e. -fno-strict-aliasing -fno-strict-overflow and -fwrapv. Out of 8 programs compiled with GCC 5.2.0, 5 show no performance modifications, 2 shows performance degradations of -1% and 1 show a performance improvement of 2%. For Clang, no modifications in performance were observed.

3.3 Summary

Currently, most of the work focuses on the security impact of UB. Sanitizers, fuzzers and methods for understanding UB have been proposed to reduce the associated risks. However, very little efforts have been put in analysing the performance impact of UB on real-life software. We fill this gap by providing the first study that analyses the performance implications of undefined behavior in LLVM, offering a comprehensive view on how this type of behavior is used in today's software.

Next, we present the flags and the benchmarks that we used in our study.

Chapter 4

Flags used for controlling UB optimizations

In this chapter we present the flags that we used for controlling the behavior of Clang/LLVM with regards to UB optimizations. We put together a suite of 19 flags that control two important subsystems of the compiler, i.e. the code generation and the optimization pipeline. The suite consists of both flags that were already implemented in Clang/LLVM and flags that we implemented. More than half of them were implemented as part of this project. We divided the flag suite in 6 categories which can be found in Table 4.1.

Each category focuses on different language constructs that can generate undefined behavior. For example, the Arithmetic Operations category contains flags that generate defined behavior for the following operations: division, remainder, shifting, addition and multiplication. By adding guards, or by deleting LLVM attributes that generate undefined behavior, we make sure that in every point of the program the aforementioned operations do not trigger undefined behavior.

Most of the flags, however, are focused on undefined behavior generated by pointer operations. In this category we have 8 flags that cover an extensive range of pointer functionalities: alignment, dereferenceability, inbounds indices, etc. This type of flags occupies the largest space in our suite with 8 flags out of 19 possible. Addressing as many undefined behavior sources related to pointer improper usage was an important goal for us because in the past decades this was a common source of severe security problems [49].

The next two categories focus on builtins, function attributes and loops that can trigger undefined behavior. The wrong usage of builtins such as `assume_aligned` and function attributes such as *pure* and *const* can render the program ill-formed. Furthermore, the usage of infinite loops is disallowed by the C/C++ standards which causes undefined behavior if a program makes use of them. By making use of the flags in this category we make sure to generate defined behavior for all the constructs related to builtins, function attributes and loops.

The Values and Conversions category focuses on undefined behaviors related to conversions from and to types such as `bool` and `enum`. On one hand, a boolean variable contains only two values, however we offer more flexibility and allow booleans to be valid for a wider range of values. On the other hand, using the already implemented *-fstrict-enums* flag we enable stricter rules for converting enums to other types and vice-versa. This flag is the only flag from our suite that enables undefined behavior instead of disabling it. Finally, in this category we also include a flag that disables the undefined behavior related to reading uninitialized variables, effectively making uninitialized reads equal to 0.

The last category focused on aliasing rules implemented inside LLVM. The effect of the flags present in this category is to disable any reasoning based on undefined behavior when proving aliasing information. For example, accessing values at indices outside an array, commonly referred as an out-of-bounds (OOB) access, is considered undefined behavior and it is used by LLVM to prove that two pointers do not alias. *-disable-oob-analysis* disables such reasoning, forcing LLVM to find different methods for proving that the pointers do not alias.

| Category | Flags | Short Name |
|------------------------|---------------------------------------|------------|
| Arithmetic Operations | -fcheck-div-rem-overflow | AO1 |
| | -fconstrain-shift-value | AO2 |
| | -fwrapv | AO3 |
| Pointers | -fdrop-align-attr | P1 |
| | -fdrop-deref-attr | P2 |
| | -fdrop-inbounds-from-gep | P3 |
| | -fno-delete-null-pointer-checks | P4 |
| | -fdrop-noalias-restrict-attr | P5 |
| | -fno-strict-aliasing | P6 |
| | -fno-use-default-alignment | P7 |
| | -Xclang -no-enable-noundef-analysis | P8 |
| Builtins | -fdrop-ub-builtins | B1 |
| Functions and Loops | -fignore-pure-const-attrs | FL1 |
| | -fno-finite-loops | FL2 |
| Values and Conversions | -fno-constrain-bool-value | VC1 |
| | -fstrict-enums | VC2 |
| | -mllvm -zero-uninit-loads | VC3 |
| Alias Analysis | -mllvm -disable-oob-analysis | AA1 |
| | -mllvm -disable-object-based-analysis | AA2 |

Table 4.1: Categories of UB controlled by flags

Finally, we also had plans to add a flag that disables optimizations based on use-after-free, however, we did not find any pass in LLVM that exploits the usage of deallocated objects. We also did not add any flag for UB that comes from fixed points operations because our benchmarks did not make use of them and we did not address UB coming from standard library operations.

The rest of this chapter is divided in two parts. In the first part we offer a detailed description of the flags in each category mentioned above. We cover the C/C++ constructs that the flag affects as well as the implementation details inside LLVM. In the second part we present the implementation process of each flag and the tools used in this process.

4.1 Description of the flags

This section presents all 19 flags used in our experiments. For each flag presented in Table 4.1 we cover the specific undefined behavior that it addresses and how it affects the LLVM Intermediate Representation (IR).

4.1.1 Arithmetic Operation

-fcheck-div-rem-overflow

When generating code for division and remainder, an additional check is added to ensure that the operation is well defined. The operations are not well defined when either the divisor is 0 or when the dividend is INT_MIN and the divisor is -1 as this would render INT_MAX+1. This equivalent C code for the check is presented in Listing 7.5.

When the operands are unsigned, the expression simplifies as we only need to verify if the second operand is 0.

Listing 4.1: Equivalent C code for the checks added before division and remainder

```
if (b == 0 || (a == INT_MIN && b == -1))
    abort();
else
    a [/ or %] b;
```

-fconstrain-shift-value

When generating any kind of shift, we constrain the right operand of the shift to not exceed the length in bits of the left operand. In LLVM, this adds an extra *and* instruction before every *shift* instruction. In rare cases, when the bitwidth of the shifted type is not a power of 2, a *urem* instruction is added instead of *and*.

This mimics the result of x86 [19] and RISC-V [55] processors where a shift operation on a 32 bit register "a » b" is equivalent with "a » (b & 0x1f)". However, on ARM [38] processors, a shift overflow results in 0, hence the extra *and* instruction is not needed. Since we only did benchmark experiments on x86, we kept the extra *and* instruction.

However in C/C++, for 32-bits integer types, a shift with a value greater than 31 renders the program ill-formed because the result would not fit in the integer type.

-fwrapv

Normally, when addition, subtraction or multiplication between two integer types do not fit in the result integer type the behavior is undefined.

This option instructs the compiler to change the semantics of signed arithmetic overflow so it wraps around using two's complement representation. In LLVM, the *nsw* attribute is dropped when this flag is enabled.

4.1.2 Pointers

-fdrop-align-attr

The LLVM *align* attribute indicates that a pointer value has a specified alignment. If the specified alignment does not match the actual alignment of the pointer then UB is triggered. To mitigate against this, we remove all such attributes from all the pointers in the program so that UB does not get triggered.

-fdrop-deref-attr

The LLVM *dereferenceable* and *dereferenceable_or_null* attributes indicate how many bytes from a pointer value can be dereferenced. If the user of this pointer tries to dereference more bytes than allowed, then UB is triggered. To mitigate against this, we remove all such attributes from all the pointers in the program so that UB does not get triggered.

This attribute is very common in C++ code because LLVM assumes that the whole object is dereferenceable if there is a pointer that points to it, especially the *this* pointer.

-fdrop-inbounds-from-gep

Accessing memory outside the bounds of an object in UB. To access memory, either for writing or for reading, LLVM uses two instructions. The first instruction is called *getelementptr* (GEP)

and it is used to calculate the address where the memory operation will take place. Then the result of the GEP is fed into either a load or a store instruction.

GEP has an attribute called *inbounds* that controls the behavior of the result when the computed address is going out of bounds of its underlying object. If *inbounds* is enabled then the computation of such address is UB. To mitigate against this, we remove all such attributes from all the GEP instructions in the program so that UB does not get triggered.

-fno-delete-null-pointer-checks

This flag assumes that programs can safely dereference null pointers, and that code or data can reside at address zero. This does not assume that if a pointer is checked after it has already been dereferenced then the pointer cannot be null. In LLVM, this adds the *null_pointer_is_valid* function attribute and deletes the *nonnull* attribute on function parameters.

-fdrop-noalias-restrict-attr

Restrict pointers are used to specify that only one pointer is used to access the object to which it points. In LLVM, this is translated into a *noalias* attribute. If the rules of the *noalias* attribute are violated then UB is triggered. To mitigate against this, we remove all such attributes from all the pointers in the program so that UB does not get triggered.

-fno-strict-aliasing

Disallows the compiler to use type information to disambiguate aliasing information. In particular, an object of one type is not assumed to reside at a different address of an object of a different type, unless the types are compatible. For example, an unsigned int can alias an int, but not a void* or a double. The char type may alias with any other type. In LLVM, *tbaa* metadata is dropped when this flag is enabled.

-fno-use-default-alignment

Any memory operation, including load, store, memcpy, etc., that use a misaligned pointer will result in undefined behavior. This flag pessimistically uses the weakest alignment for all types to discard the risk of UB. In LLVM, the alignment of every pointer involved in a memory operation is forcefully set to 1 by modifying the *align* attribute.

-Xclang -no-enable-noundef-analysis

This flag stops the analysis of function arguments and return type for mandatory definedness. In LLVM, this drops the *noundef* attribute from function arguments and return values.

4.1.3 Builtins

-fdrop-ub-builtins

Programmers can encode assumptions in their programs. The assumptions can be focused either on values, using `__builtin_assume`, or on alignment, using `__builtin_assume_aligned`. If the code that is dependent on these assumptions breaks them then the behavior is undefined. To mitigate against this, we remove all such assumptions from the program so that UB does not get triggered.

Another builtin that we cover in this flag is `__builtin_unreachable`. In LLVM, the unreachable instruction has no defined semantics. We replace this builtin with `__builtin_trap` so that we get defined semantics in code that uses such builtins.

4.1.4 Functions and Loops

-fignore-pure-const-attr

pure and *const* attributes are used to decorate functions that have no observable effect on the state of the program other than returning a value. As such, any function that is decorated with any of the two attributes, and modifies the state of the program by writing into global pointers or into pointers received as arguments triggers UB. Another way of triggering UB is to call a non-const or non-pure function from a const or pure function.

In LLVM, the *pure* attribute is translated into the following LLVM IR attributes:

1. *memory(none)*, the function does not access any memory
2. *nounwind*, the function never raises an exception
3. *willreturn*, the execution continues after the function exits

For *const* the following LLVM IR attributes are added:

1. *memory(readonly)*, the function only reads memory
2. *nounwind*, the function never raises an exception
3. *willreturn*, the execution continues after the function exits

To mitigate against the possible UB generated by these attributes, we remove them from all the functions in the program.

-fno-finite-loops

Assume that a loop with an exit point will not take the exit point and will loop indefinitely. This does not allow the compiler to delete loops without side effects. In LLVM, this drops the *mustprogress* attribute from functions.

4.1.5 Values and Conversions

-fno-constrain-bool-value

Allow boolean values to act as integers and not be constrained in the interval $\{0, 1\}$. This flag drops the *range* metadata from operations that involve boolean values.

-fstrict-enums

Allow the compiler to optimize using the assumption that a value of enumerated type can only be one of the values of the enumeration. This assumption may not be valid if the program uses a cast to convert an arbitrary integer value to the enumerated type. In LLVM, this adds *range* metadata to instructions that use enum values.

-mllvm -zero-uninit-loads

To remove the UB related to accessing uninitialized values, this flag replaces uninitialized loads with zero loads. Compared with "-ftrivial-auto-var-init=choice",¹ our flag does not explicitly sets to 0 all uninitialized variables, it only replaces uninitialized loads with zero loads for performance reasons. In LLVM, this replaces the *undef*, resulted from operations that load from a fresh *alloca*, with 0.

This flag implements the ideas in P2723R1 [7]. However, compared to P2723R1, we also handle heap-allocated objects.

¹<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-ftrivial-auto-var-init>

4.1.6 Alias Analysis

-mllvm -disable-oob-analysis

The alias analysis (AA) subsystem of LLVM exploits UB related to out of bounds indices. Below are two exploitations employed by AA with regards to GEP instructions:

1. If an inbounds GEP would have to start from an out of bounds address for the two pointers to alias, then we can assume that the two pointers do not alias.
2. If the size of one access is larger than the entire object on the other side, then we know such behavior is undefined and can assume that the two pointers do not alias.

This flag disables such exploitations and as such offers more pessimistic aliasing results.

-mllvm -disable-object-based-analysis

To conclude that two pointers do not alias, LLVM's AA also checks the types of the underlying objects located at these pointers. If the two objects are different then LLVM generates NoAlias information.

This flag disables such reasoning and consequently enables a flat memory model where objects can be situated everywhere in memory, even under the same pointer. This effectively disables any UB caused by accessing an object with a inappropriate pointer.

4.1.7 Summary of Intermediate Representation changes

As presented above, the effect of each flag is to guard against undefined behavior either by adding more checks in the LLVM IR or by deleting IR constructs that can possibly trigger undefined behavior. Flags that are part of the two categories are presented in Table 4.2. An exception is the *-fstrict-enums* flag that enables undefined behavior by adding an IR construct that can possibly trigger UB, i.e. the *range* metadata.

We also have a third category for flags that control directly the optimizer. To avoid the cost of setting each object to 0, we only replace the loads of uninitialized objects with 0. Then, because the Alias Analysis (AA) subsystems of LLVM uses UB, we created the rest of 2 flags to control the decision of AA that are based on out-of-bounds accesses and comparison between objects.

| Adds checks into IR | Deletes UB constructs from IR | Controls the optimizer |
|--|---|--|
| -fcheck-div-rem-overflow -fconstrain-shift-value -fno-delete-null-pointer-checks -fstrict-enums | -fwrapv -fdrop-align-attr -fdrop-deref-attr -fdrop-inbounds-from-gep -fdrop-noalias-restrict-attr -fno-strict-aliasing -fno-use-default-alignment -Xclang -no-enable-noundef-analysis -fdrop-ub-builtins -fignore-pure-const-attrs -fno-finite-loops -fno-constrain-bool-value | -mllvm -zero-uninit-loads -mllvm -disable-oob-analysis -mllvm -disable-object-based-analysis |

Table 4.2: Flags that add checks, delete UB constructs from IR and directly control the optimizer

Almost three quarters of our flags delete UB constructs from the IR. Normally, LLVM uses this information to enable certain optimizations and to provide better performance for the compiled programs. However, the semantics of the IR constructs that LLVM uses contain undefined

behavior if their rules are not respected. By removing completely the information we make sure that only defined behavior is generated with the drawback of reducing the performance of the program.

Furthermore, all flags but *-disable-oob-analysis*, *-disable-object-base-analysis* and *-zero-uninit-loads* affect the code generation subsystem either by adding more instructions or by deleting attributes. The rest of 3 flags have direct effect on the optimization pipeline from LLVM, either by affecting the alias analysis subsystem or the individual passes inside the pipeline.

4.2 Implementation of the flags

In this section, we present the methods and tools for discovering undefined behavior constructs in LLVM and the implementation process for the flags that control the UB constructs. We also present 1 bug in LLVM and 5 issues that we found in Alive2 while implementing the feature that helped us finding UB constructs. Alive2 is an automated verification tool for LLVM optimizations.

4.2.1 Implementation

We used a combination of both manual and automatic discovery of constructs that trigger undefined behavior in LLVM. The first discovery strategy consisted of manually inspecting LangRef,¹ the LLVM Language Reference Manual, and creating flags for the attributes or metadata that contain UB as part of their semantics. The second discovery strategy consisted of compiling all benchmarks in our suite with Alive2 [36] to automatically locate UB constructs. Using the two strategies, we assured, on one hand, that we cover the most common UB constructs, and on the other hand, that we cover the whole the range of UB constructs that is present in our benchmark suite.

In the first phase, i.e. manually going through LangRef, we were interested in constructs that contained the following keywords in their description: "undefined behavior", "undefined value", "undef" and "poison". In LLVM, undef and poison values are special values used for representing undefined behavior.

The constructs that we focused on lie in the following categories:

- function attributes: `mustprogress`, `willreturn`, etc.
- pointer and values attributes: `align`, `dereferenceable`, etc.
- instructions: `lshr`, `alloca`, etc.

For each construct we created a new flag which required modification either in Clang or the optimization pipeline of LLVM. This phase resulted in 7 new flags, namely: `-fcheck-div-rem-overflow`, `-fconstrain-shift-value`, `-mllvm -zero-uninit-loads`, `-fdrop-noalias-restrict-attr`, `-fno-constrain-shift-value`, `-fno-use-default-alignment` and `-mllvm -disable-object-based-analysis`. Out of these 7 flags, 3 flags lie in the Arithmetic Operations category, 3 flags lie in the Pointers category and 1 flag lies in the Alias Analysis category.

There are also UBs that we did not address, for example: exploitations based on accessing use-after-free objects, exploitations based on fixed point operations and exploitations based on undefined behavior arising from standard library operations, such as `abs(INT_MIN)`. Since either LLVM did not exploit them or our benchmarks did not make use of these features, we choose to not include them in our study.

In the second phase, we compiled the whole benchmark suite with Alive2 to find missed opportunities for UB flags. Alive2 is an automatic verification tool for LLVM optimizations. Briefly,

¹<https://llvm.org/docs/LangRef.html>

after each optimization pass it checks whether the IR contains undefined behavior. This helps us find places in the optimization pipeline that exploit UB.

We decided to patch Clang whenever we found an UB exploitation in the optimization pipeline of LLVM. By generating code free of UB directly from Clang, we assured that further optimization cannot take any advantage or further propagate the exploitation of undefined behavior. For example, instead of patching each place in LLVM that takes advantage by division by 0, we generated code from Clang that guarded each division so that LLVM cannot further optimize it.

The following is an example of undefined behavior detected using Alive2. Listing 4.2 contains a C function from the aom-av1-3.7.0 benchmark that returns a boolean value based on a pointer received as parameter.

Listing 4.2: C code for `is_sb_aq_enabled`

```
static bool is_sb_aq_enabled(const AV1_COMP *const cpi) {
    return cpi->rc.sb64_target_rate >= 256;
}
```

Listing 4.3 contains the corresponding Alive2 output for `is_sb_aq_enabled`. The output can be broken down in multiple parts:

1. The IR before the optimization and the IR after the optimization, they are delimited by `=>`
2. Information about the validity of the transformation, in this case we get an "Transformation doesn't verify! (unsound) ERROR: Source has guardable UB"
3. Detailed information about the cause of the error including values that triggered the error and the memory state

Listing 4.3: Alive2 output on `is_sb_aq_enabled`

```
====> Part 1 <====
define i1 @is_sb_aq_enabled(ptr noundef %cpi) null_pointer_is_valid zeroext
{
  %entry:
    %cpi.addr = alloca i64 8, align 8
    store ptr noundef %cpi, ptr %cpi.addr, align 1
    %0 = load ptr, ptr %cpi.addr, align 1
    %rc = gep inbounds ptr %0, 677024 x i32 0, 1 x i64 427536
    %sb64_target_rate = gep inbounds ptr %rc, 248 x i32 0, 1 x i64 16
    %1 = load i32, ptr %sb64_target_rate, align 1
    %cmp = icmp sge i32 %1, 256
    ret i1 %cmp
}
=>
define i1 @is_sb_aq_enabled(ptr noundef %cpi) null_pointer_is_valid zeroext
{
  %entry:
    %rc = gep inbounds ptr noundef %cpi, 677024 x i32 0, 1 x i64 427536
    %sb64_target_rate = gep inbounds ptr %rc, 248 x i32 0, 1 x i64 16
    %0 = load i32, ptr %sb64_target_rate, align 1
    %cmp = icmp sge i32 %0, 256
    ret i1 %cmp
}

====> Part 2 <====
Transformation doesn't verify! (unsound)
```

```

ERROR: Source has guardable UB

====> Part 3 <====
Example:
ptr noundef %cpi = pointer(non-local, block_id=0, offset=-388623)

Source:
ptr %cpi.addr = null
ptr %0 = pointer(non-local, block_id=0, offset=-388623)
ptr %rc = poison
ptr %sb64_target_rate = poison
i32 %1 = UB triggered!

SOURCE MEMORY STATE
=====
NON-LOCAL BLOCKS:
Block 0 >      size: 601368      align: 9223372036854775808      alloc type:
0      address: 0
Block 1 >      size: 8968        align: 4                  alloc type: 0      address:
601368

LOCAL BLOCKS:
Block 2 >      size: 601368      align: 9223372036854775808      alloc type:
0      address: 0

Target:
ptr %rc = poison
ptr %sb64_target_rate = poison
i32 %0 = UB triggered!

```

Because the pointer might point to an object that is not allocated, the code triggers undefined behavior. As such, %rc and %sb64_target_rate contain poison values in Part 3 of the output. They cause the subsequent "UB triggered!" in %1. The %1 variable is the result of a load instruction that has the following semantics in LLVM IR: "if pointer is not a well-defined value, the behavior is undefined".

The source of undefinedness stems from the presence of the *inbounds* attribute in the *getelementptr* (GEP) instructions that compute the %rc and %sb64_target_rate variables. When this attribute is present, GEP generates poison if the following condition is not respected: "The base pointer has an in bounds address of an allocated object, which means that it points into an allocated object, or to its end. Note that the object does not have to be live anymore; being in-bounds of a deallocated object is sufficient."

Alive2 is not capable of using inter-procedural analysis that would allow it to assess whether the address points to an allocated object by inspecting the caller of the function. As such, based only on the information available in the current function, Alive2 concludes that the aforementioned *inbounds* rule is not respected, hence undefined behavior is triggered.

To stop the generation of the poison values, we patched Clang to not emit *inbounds* attribute to any GEP instruction. We integrated the functionality of removing *inbounds* from GEP instructions in a flag called `-fdrop-inbounds-from-gep`. We ran similar analyses on other LLVM IR constructs, resulting in the following 6 flags: `-fdrop-noalias-restrict-attr`, `-fdrop-align-attr`, `-fdrop-deref-attr`, `-fdrop-ub-builtins`, `-fdrop-pure-const-attr`, `-mllvm -disable-oob-analysis`.

We also had plans to drop the *noundef* attribute but fortunately there already existed a flag for that, called `"-Xclang -no-enable-noundef-analysis"`.

Out of the 8 flags generated in this phase, 5 flags are part of the Pointers category, 1 is part of

the Builtins category, 1 is part of the Functions and Loops category and 1 is part of the Alias Analysis category.

4.2.2 LLVM bugs and Alive2 modifications

As part of this project we implemented in Alive2 a feature that enables the discovery of UBs that can be guarded from Clang. As discussed earlier in this chapter, we preferred to generate code free of UB directly from Clang instead of patching each place in LLVM that exploits UB optimizations. Furthermore, we discovered 1 LLVM bug when we ran Alive2 on our benchmarks suite.

Below is the bug found in LLVM:

1. [Vectorization of loop reduction introduces an aligned store incorrectly](#)

We found this bug while developing the *-fno-use-default-alignment* flag. For this flag our goal was to align all memory operations, including loads, stores and memcpy, to 1. However, the Loop Vectorizer pass inside LLVM changed the alignment from 1 to 2 in a store instruction. This is not a valid transformation as LLVM is not allowed to over-align memory addresses.

Below are 4 false positives that we fixed in Alive2 in order to discover guardable UBs in our benchmark suite, then we also list one Alive2 missing feature that we discovered but did not implement:

1. [UB exploit mode: fix false-positive with memcpy SROA transformation](#)
2. [UB exploit mode: fix false positive with byval fncall](#)
3. [UB exploit: work around false positives with poison ptrs](#)
4. [rework support for arbitrary-sized globals to support 0-sized ones](#)
5. Alive2 doesn't analyse locally-allocated pointers on function calls

The first three issues fix false-positives reported by Alive2 in the benchmarks that we analyzed. The third issue adds support for zero-sized arrays which is a feature that LLVM supports but Alive2 did not at that moment. The last issue is a missing feature that we discovered but did not address. Alive2 is not able to analyse pointers declared as local variables passed in a function call. An example of this problem is presented in Listing 4.4.

The `xname_cmp` function is part of the `openssl` benchmark. In the function, `abuf` and `bbuf` are local pointers which are passed to the `i2d_X509_NAME` function. However, Alive2 cannot continue to analyse the pointers inside the new function and prove that they are well defined after the inner function returns. Consequently, Alive2 concludes that the called function might introduce UB in `abuf` and `bbuf`. This is not a fundamental limitation of Alive2, but a feature that is currently not implemented.

Listing 4.4: OpenSSL code where Alive2 cannot analyse locally-allocated pointers

```

static int xname_cmp(const X509_NAME *a, const X509_NAME *b)
{
    unsigned char *abuf = NULL, *bbuf = NULL;

    ...

    // Alive2 cannot propagate the locally allocated pointers in
    // i2d_X509_NAME
    alen = i2d_X509_NAME((X509_NAME *)a, &abuf);
    blen = i2d_X509_NAME((X509_NAME *)b, &bbuf);

    ...
    memcmp(abuf, bbuf, alen);
    ...
}

```

4.3 Summary

In this chapter we have presented a detailed description of all the 19 compiler flags that we use in our benchmarks for controlling the behavior of LLVM with regards to undefined behavior optimizations. We divided the 19 flags in 6 categories: Arithmetic Operations, Pointers, Builtins, Functions and Loops, Values and Conversions, and Alias Analysis. These categories group flags by the language constructs they affect.

We also showed that 4 flags out of 19 add checks to the IR while 11 flags delete UB constructs from the IR and 3 flags control directly the LLVM optimizer so that the compiled programs do not exhibit undefined behavior. Furthermore, we presented how we implemented each flag, how we employed Alive2 in finding UB constructs and the bugs that we found in LLVM plus the issues found while implementing guardable UB in Alive2.

As part of this project we implemented 13 new flags for LLVM and we used 6 existing flags. Next, we present the benchmark suite that we designed for our performance study.

Next, we present the benchmark suite that we used and the best practices we followed to get stable results.

Chapter 5

Benchmarks

For our experiments we use one 2x Intel Xeon CPU E5-2680 v2 @ 2.80GHz (IvyBridge) server bundled with 64GB DDR3 RAM (@ 1600 MHz) running Debian GNU/Linux 11 (bullseye). The benchmarks are compiled using a forked version of the Clang/LLVM 16.0.6 that can be found at <https://github.com/lucic71/llvm-project/tree/release/16.x-ub>. All the benchmarks are collected and ran using a modified version of Phoronix Test Suite 10.8.4 (PTS) [31] that can be found at <https://github.com/lucic71/phoronix-test-suite>

We collected a benchmark suite of 22 C/C++ mature applications that already contained performance tests as part of their infrastructure. In total we collected 129 performance tests. In the experiments that we conducted we did not touch the source code or the build systems of the applications. Table 5.1 presents all 22 benchmarks that we use, the corresponding number of lines of code and the corresponding measurement scale for each benchmark. As part of this project, we added the Z3 benchmark in the official PTS benchmarks repository.

Most of the benchmarks in our suite measure the performance by the time spent in running the underlying application. 11 benchmarks use time as a measurement scale. Next, Mflops, MB/s and megapixels/second are the second most popular categories with 4 benchmarks per category. We also have 1 benchmarks that are not part of any of the mentioned categories, i.e. john-the-ripper which measures the performance in checks/second.

In terms of lines of code (LOC) per benchmark, the benchmark that contains the most LOC is build-llvm with over 2 milion LOC, at the other end compress-pbzip2 contains only 6 kLOC. The mean lies 259 kLOC with a standard deviation of 419 kLOC.

In this study, we considered performance modifications within the -2% and +2% as noise. We exclude from our analysis any data point in this range because minor fluctuations are not relevant for the real-world applications that we are benchmarking. Compared to microbenchmarks, where the noise interval can be reduced significantly, the applications in our suite present moderate variability in the results.

The rest of this chapter presents the building process of the benchmark suite and the techniques we employed to create a benchmark infrastructure resilient to $\pm 2\%$ noise.

5.1 Benchmark suite

We chose PTS as our benchmarks source because of it rich number of benchmarks. It contains more than 500 benchmarks, however, for our experiments we looked only at applications written in C/C++, targeted on CPU workloads, with diverse run-times and easily compilable by Clang.

After applying these filters we selected only 22 benchmarks from PTS. The benchmarks are presented in Table 5.1.

To filter benchmarks based on our aforementioned requirements we used the public GitHub repository that contains information about all the benchmarks that are part of PTS.¹ Each benchmark is represented as an XML file. Exploring C/C++ benchmarks is done by searching the string "build-utilities" in the the *ExternalDependencies* XML node of each benchmark.

However, "built-utilities" is often used in conjunction with other dependencies such as "cuda", "opencl", "openmpi-development", "vulkan-development". Since we were only interested in running plain CPU workloads, we ignored benchmarks that make use of the additional external dependencies.

To ensure that the benchmarks are compiled with Clang and that our compiler flags are correctly used we employed two techniques. On one hand we set the CC/CXX environment variables to point to our modified version of Clang where the UB flags are implemented. On the other hand, we created a wrapper script over Clang whose role is to append the UB flags at the end of the compilation command. This helped in ensuring that the UB flags have the greatest priority in the command.

| No | Benchmark | kLOC | Measurement scale |
|----|------------------------|-------|-------------------|
| 1 | aom-av1-3.7.0 | 508 | frames/second |
| 2 | botan-1.6.0 | 148 | MB/second |
| 3 | build-llvm-1.5.0 | 2.139 | seconds |
| 4 | compress-pbzip2-1.6.0 | 6 | seconds |
| 5 | compress-zstd-1.6.0 | 85 | MB/second |
| 6 | draco-1.6.0 | 50 | seconds |
| 7 | encode-flac-1.8.1 | 59 | seconds |
| 8 | espeak-1.7.0 | 44 | seconds |
| 9 | fftw-1.2.0 | 255 | Mflops |
| 10 | graphics-magick-2.1.0 | 263 | iterations/second |
| 11 | john-the-ripper-1.8.0 | 315 | checks/second |
| 12 | jpegl-1.5.0 | 106 | megapixels/second |
| 13 | lua-jit-1.1.0 | 69 | Mflops |
| 14 | ngspice-1.0.0 | 514 | seconds |
| 15 | openssl-3.1.0 | 472 | bytes/s |
| 16 | primesieve-1.9.0 | 9 | seconds |
| 17 | quantlib-1.2.0 | 395 | Mflops |
| 18 | rnnoise-1.0.2 | 14 | seconds |
| 19 | simdjson-2.0.1 | 74 | MB/s |
| 20 | sqlite-speedtest-1.0.1 | 249 | seconds |
| 21 | tjbench-1.2.0 | 57 | megapixels/second |
| 22 | z3-1.0.0 | 496 | seconds |

Table 5.1: Benchmarks used in our experiments

A final goal of our benchmark suite was to incorporate benchmarks with various run-times since we are interested in seeing the impact of the UB flags on long-running application as well as on short-running applications. We collected the run-times on the same machine that we ran the benchmarks on. From 22 benchmarks, the mean run-time is 951.5 seconds and the standard deviation is 1772.0 seconds. Figure 5.1 presents a detailed view of the run-time of each benchmark.

¹<https://github.com/phoronix-test-suite/test-profiles>

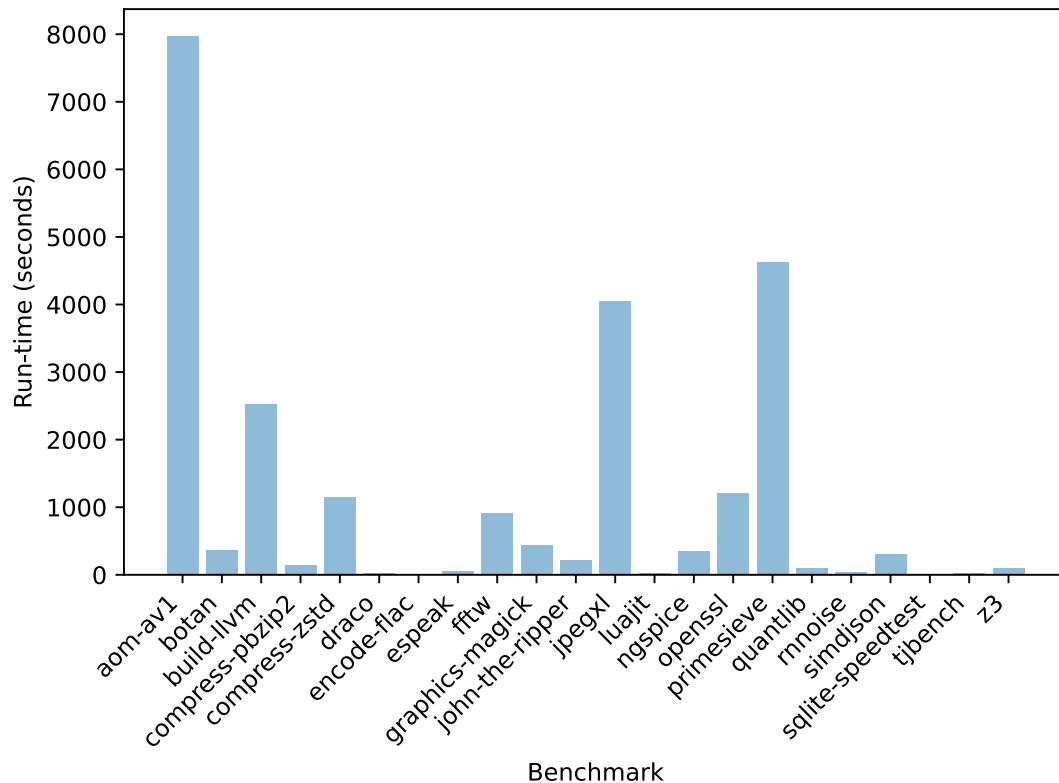


Figure 5.1: Run-time in seconds of each benchmark present in our suite, the total run-time is greater than 7 hours

The presented running times are collected only for a single run and a single flag. Given that the total reported time is 7 hours and we benchmark no more than 20 flags then a complete run would last approximately 6 days. However, to reduce the variance in the results, PTS uses a dynamic number of run-times which ensures that the standard deviation of the results does not reach a predefined threshold. In consequence, the time per benchmark increases. We have observed that usually 3 runs are sufficient per benchmark. This results in 2.5 weeks to complete a full benchmark session.

In summary, we presented how we gathered a collection of 22 C/C++ benchmarks already available in PTS, the methods we used for ensuring that our UB flags take effect on the benchmark suite and a report of the time spent per each benchmark, with a total of 2.5 weeks to complete a full benchmark session. Next we present the infrastructure used for benchmarking, focusing on techniques that we used to stabilize the results of the benchmarks.

5.2 Benchmark infrastructure

Mytkowicz et al. [39] discuss the importance of setting up a robust benchmark infrastructure before starting any performance experiment. Since the environment can add measurement bias into the results, we took care to reduce the noise generated by components such as the operating system kernel, other applications running on the system and hardware interactions. Consequently, we designed a benchmark infrastructure where the standard deviation between the results of one benchmark is at most 3% and where we consider noise any performance modifications that fall into the interval $[-2\%, +2\%]$.

In their work, Mytkowicz et al. prove that the memory layout of the application that is benchmarked is a common root for measurement bias. Two factors that they use in their experiments for changing the memory layout of the benchmark are the size of the UNIX environment variables and the linking order of the object files. They saw differences of 33% between various memory layouts controlled by the two above factors.

In our experiments, we used the default UNIX environment size that comes with the OS where we ran the benchmarks, i.e. Debian GNU/Linux 11 (bullseye). We also did no modification to the link order of the applications. However, we followed the best practices in benchmarking and enabled the following noise reduction mechanisms:

- Run benchmarks in single-threaded mode to reduce inter-threads communication noise
- Disable Turbo Boost and Hyper Threading to reduce CPU noise
- Use *taskset* and *nice* Linux utilities to reduce CPU and scheduler noise
- Set the CPU to 80% of the total frequency to reduce thermal throttling noise
- Disable ASLR to reduce kernel noise
- Use statistical methods provided by PTS to reduce noise from other sources

These modifications were only applied on the running environment and on the scripts that control the PTS benchmarks. No modification to the source code or the build system of the application were applied.

The first and most important modification that we employed was to run all benchmarks in single-threaded mode. By default, PTS runs benchmarks in multi-threaded mode to gain as much performance as possible but sacrificing results stability. As locking mechanisms and inter-threads communication add noise in the final results, we wanted to make sure that we reduce this class of noise to a minimum. The environment modifications can be found at <https://github.com/lucic71/test-profiles>.

Then, we disabled Intel technologies such as Turbo Boost and Hyper Threading which are known for their capabilities of improving the performance of the application at the cost of higher variability of the results. Furthermore, we set the CPU frequency to a fixed value, i.e. 80% of the maximum CPU capacity, to avoid thermal throttling and avoid dynamic frequency scaling.

Moreover, we used the *taskset* and *nice* utilities to control the behavior of the benchmark Linux process. Setting the process CPU affinity and controlling the priority of the Linux process assured that we reduce the noise generated by kernel operations such as moving the process to another CPU core or scheduling another Linux process while our benchmark process runs. Another protection mechanism we used was to disable ASLR globally.

Finally, we used statistical methods to improve the stability of the benchmark results. PTS uses a dynamic run count that will get triggered every time the standard deviation of a benchmark exceeds a predefined threshold. This ensures the statistical significance of the results. The default value for the standard deviation threshold is 3.5. We also used this value in our benchmarks.

After we defined all noise sources and applied techniques for reducing them, we defined the benchmarking process using a series of Bash scripts which can be found at <https://github.com/lucic71/ub-benchmark-infra>. The summarized process is presented in Listing 5.1.

Listing 5.1: Benchmarking process described in pseudocode

```
prepare_benchmark_environment()

flags = [baseline, ..., -fwrapv, -fconstrain-shift-value, ..., all]
for flag in flags:
    for b in benchmarks:
        compile_benchmark_with_flag(b, flag)
        gather_binary_size_information()
        run_benchmark(b)
        gather_performance_information()
```

In this process, we iterate over all 19 flags that we present in Chapter 4 plus 2 more special flags, i.e. *baseline* and *all*. First, the entire benchmark suite is compiled with an UB flag, then we run and collect the results from the whole suite.

baseline and *all* are special flags that we used in our scripts. On one hand, when *baseline* is enabled, no UB flag is activated, this is useful to compute the performance impact of each UB flag. On the other hand, when *all* is enabled, all flags that disable UB exploitations are activated, except *-fstrict-enums* which enables UB exploitations.

In summary, we presented the noise reduction mechanisms we employed when running the benchmarks and the high-level process of compiling the benchmark suite with the UB flags and then collecting binary size and performance numbers. Next, we present a series of bugs that we encountered in PTS while preparing our benchmark infrastructure.

5.3 Bugs discovered in Phoronix Test Suite

While preparing our benchmark infrastructure, we reported and fixed 4 bugs in PTS. The bugs target the wrong usage of benchmark names, wrong usage of environment variables and wrong usage of the application that is benchmarked:

- [Commas in test names break csv files](#)
- [\[redis-1.4.0\] NUM_CPU_PHYSICAL_CORES shall not be greater than 128](#)
- [Test profiles don't respect CC/CXX environment variables](#)
- [CXXFLAGS should be CXXFLAGS](#)

We encountered the first bug while parsing the benchmark results. Our goal was to convert all the results into CSV format to evaluate the impact of each flag relative to the baseline. However, the presence of commas in the option names for *compress-zstd* caused errors in parsing our CSV files.

We discovered the second bug while running *redis*, a benchmark that we eventually discarded from our suite, on a machine with more than 128 cores. Because the PTS scripts were not handling the case where *NUM_CPU_PHYSICAL_CORES*, a special PTS variable that controls the number of available cores, is greater than 128, our benchmark stopped working.

The last two bugs were encountered when searching for C/C++ benchmarks that would easily be compilable with Clang. Our strategy was to set the *CC*, *CXX*, *CFLAGS* and *CXXFLAGS* environment variables and check which benchmarks respected them. In this process we found more than 50 benchmarks that did not respect these variables. This violates the official documentation of PTS which states “The Phoronix Test Suite does respect *CC/CXX* environment variables and test profiles are expected to honor *CFLAGS/CXXFLAGS* and other compiler

settings”.¹

Finally, the last bug, which targets the `aircrack-ng` benchmark, refers to a typo in the `CXXFLAGS` environment variable.

5.4 Summary

This chapter presented the process of gathering, running and collecting results from a benchmark suite of 22 C/C++ real-world applications that contains 129 performance tests. We used Phoronix Test Suite to create the benchmark suite. Even if PTS has more than 500 benchmarks, we selected our candidates based on the following prerequisites: to be implemented in C/C++, to target CPU workloads, to have diverse run-times and to be compilable by Clang.

Furthermore, we made the necessary modifications to have robust and predictable results, the modifications include: running the benchmark single-threaded mode, disabling Turbo Boost and Hyper Threading and other techniques to reduce noise coming from different sources.

Finally, a complete run of the 21 flags, including *all* and *baseline*, takes around 2.5 weeks. 11 benchmarks use time spent to run the application as the measurement scale, other measurement scales are: MB/s, Mflops and megapixels/second.

In the next chapter, we present a comprehensive view of the results that we gathered based on the benchmarks presented in this chapter and the flags presented in Chapter 4.

¹<https://github.com/phoronix-test-suite/phoronix-test-suite/blob/master/documentation/phoronix-test-suite.md>

Chapter 6

Results

This chapter presents the impact of the 19 flags that control UB optimizations on a benchmark suite with 22 C/C++ applications that contains more than 125 performance tests which results in more than 2500 data points. We benchmark each individual flag presented in Chapter 4 and also combine all flags that disable UB optimizations. In 125 cases the flags result in a performance degradation and in 42 cases the flags result in a performance improvement.

To recover the lost performance resulted from the deactivation of UB optimizations, we use Link Time Optimizations (LTO). When the UB flags and LTO are enabled at the same time, performance degradations reduce from 125 to 43 and performance improvements increase from 42 to 115.

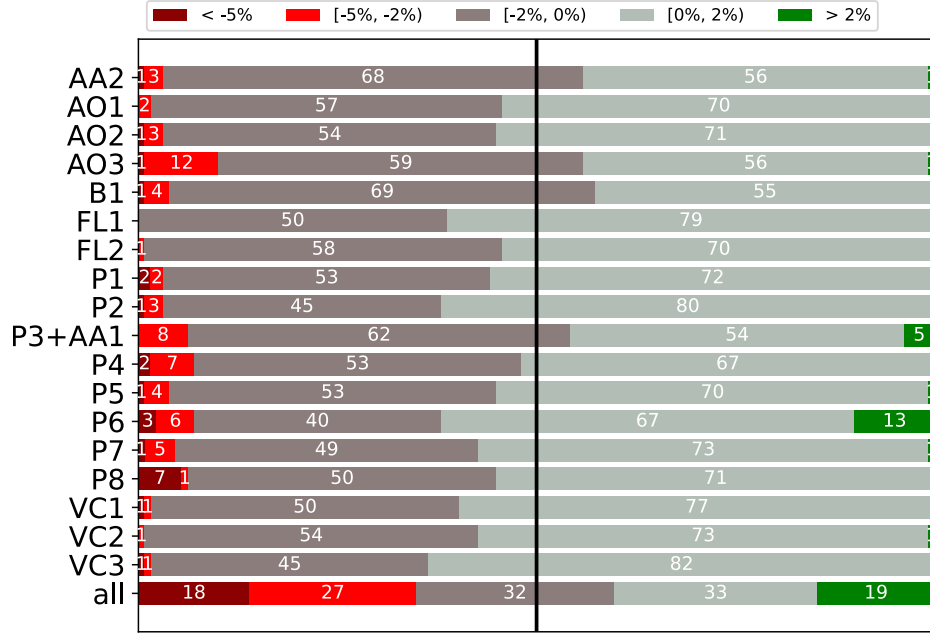
The rest of this chapter first presents how UB flags interact with the 22 benchmarks, then we show how LTO can recover the lost performance caused by the UB flags. Besides performance, we also include information about the effect of the UB flags on the binary size. In the non-LTO benchmarks, only pbzip2 presents severe degradations, of -6%, for the *all* and *-fdrop-inbounds-from-gep* flags. The best benchmark in terms of binary size is primesieve with improvements of +4% in the *all*, *-fdrop-inbounds-from-gep* and *-disable-object-based-analysis* flags.

6.1 Performance Impact

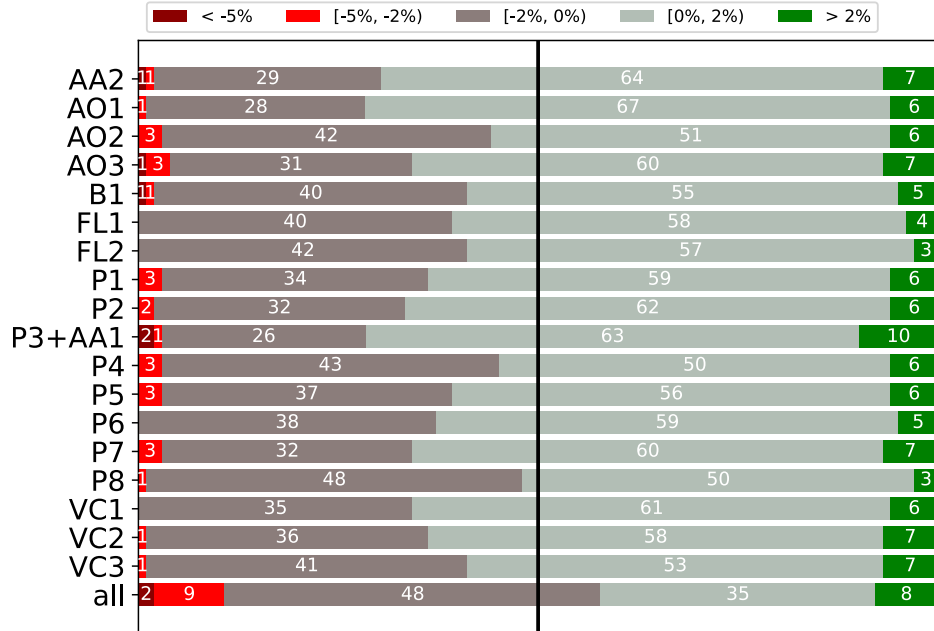
For each benchmark presented in Chapter 5 we compute the baseline performance and the performance when the flags are enabled. To compute the baseline we disable all UB flags. Then, one by one, we activate the flags to compute the performance when UB optimizations are disabled. One exception is the combination between *-fdrop-inbounds-from-gep* and *-disable-oob-analysis* which needs to be activated at the same time to disable UB exploitations related to out-of-bounds accesses. We also benchmark the *-fstrict-enums* which enables UB optimizations, instead of disabling them, as the rest of the flags do. Finally, *all* activates all flags that disable UB optimizations. For each setup we also enable the optimization level -O2.

We gathered a total number of 129 performance tests that are spread across 22 benchmarks. In Figure 6.1a, we divided the performance impact in multiple performance segments. Since positive performance impacts are not common, we only have one segment that covers improvements. Then, as discussed in Chapter 5, performance results that lie in the interval [-2%, +2%] are considered noise. At last, the negative performance impact is divided in two: moderate degradations ([-5%, -2%]) and severe degradations (< -5%). For each performance interval, we present the number of benchmarks that are affected.

We run the performance tests for 19 times, corresponding to the number of flags, which results in almost 2500 data points. Out of this number, 43 tests result in severe degradations, 92 tests



(a) non-LTO



(b) LTO

Figure 6.1: UB flags and the number of benchmarks per impact interval. Impact is divided into severe degradations ($< -5\%$), moderate degradations ($[-5\%, -2\%)$) and improvements ($> +2\%$). The vertical line divides the dataset in half

result in moderate degradations and 42 result in improvements.

All flags but one exhibit more performance degradations than performance improvements. *-fno-strict-aliasing* (P6) is the only flag that does not follow this trend since it exhibits 14 improvements and 10 degradations.

In the Pointers (P) and *all* flag categories the trend towards negative performance is the most obvious. At the opposite end, categories such as Functions and Loops (FL), Values and Conversions (VC) and Alias Analysis (AA) exhibit a low amount of performance modifications with at most 5 modifications per category. At the middle, the Arithmetic Operations (AO) flags exhibit a moderate number of negative performance modification with almost no performance improvement.

To recover the lost performance caused by the flags that disable UB optimizations, we use Link Time Optimizations (LTO). When LTO and the UB flags are enabled at the same time, performance degradations reduce from 125 to 43 and performance improvements increase from 42 to 115. Figure 6.1b shows the results for the LTO builds. Compared to Figure 6.1a, the LTO builds present more improvements per flag. Furthermore, the severe degradations are reduced to only 1 and the number of moderate degradations decreases per flag.

However, because not all the benchmarks could be built with LTO, we could not gather results for all of them. Either their build system was not designed to support LTO or the changes we needed to do in the build system were not trivial. As such the following benchmarks are missing LTO results: aom-av1, build-llvm, openssl.

Most of our flags delete information that could produce UB from the IR. However, when LTO is enabled, information such as function and arguments metadata is recovered using interprocedural analysis. This type of analysis assures that metadata is only added in places that would not produce UB since the compiler is not allowed to add UB if the original program did not contain. As a consequence, certain optimizations are re-enabled in the places where information is recovered.

In the LTO setup, certain flag categories such as Builtins (B) and Functions and Loops (FL) do not exhibit any performance degradation. The Pointers (P) category shows a reduction in the number of degradation of 4.5 times, the Arithmetic Operations (AO) shows a reduction of 2.5 times and the Values and Conversions (VC) shows a reduction of 1.66. Finally, degradations in *all* are drastically reduced by 5.8 times and the only flag that still has a severe degradation, i.e. Alias Analysis 2 (AA2), reduced the moderated degradations from 4 to 2.

Still, even in the LTO benchmarks, the P and AO categories show the biggest number of degradations. To further reduce this number, other techniques such as improved alias analysis or introduction of run-time checks need to be added. We discuss these techniques in Chapter 7.

Figure 6.2a and Figure 6.2b present the distribution of the performance impact per flag in the non-LTO and LTO setups. In the non-LTO setup, the median of almost all the flags in the negative side of the plot, the only exceptions are *-fno-strict-aliasing* (P6) and *-fstrict-enums* (VC2). We expected VC2 to provide better performance since it enables UB, however for P6 we expected a more pronounced negative impact since it type based alias analysis information from the IR which is used in many optimizations.

Pointer (P), Values and Conversions (VC) and *all* categories present the most spread-out distributions. Even tough categories such as Arithmetic Operations (AO) impact more benchmarks than VC, the performance of the first is more predictable than the performance of the latter. For *all*, we expected more diverse results since it activates all flags that disable UB optimizations which can have a wide range of effects on the optimization pipeline.

However, when LTO is enabled, the median trend moves towards positive values. The only exception is the *all* flag which slightly moves its median, but moves worst performance degra-

dation. Moreover, in the non-LTO setup, the majority of the medians were placed in the $[-5\%, 0\%]$ interval, but in the LTO setup almost all of them move in the $[0\%, +5\%]$ interval. For LTO the best performance improvement also increases.

Besides the increased performance median, the flags in the VC category improve their stability for the entire benchmark suite as their minimum and maximum are both closer to the median. However, the majority of the flags in the AO and P present a weaker stability when compiled with LTO. A curious case are the flags from the Builtins (B), Functions and Loops (FL) categories and *-fno-use-default-alignment* (P8) flag which in non-LTO mode barely show any performance modifications, but in LTO mode, besides the performance modifications they also present a high variance in the results.

Figure 6.3a offers a different perspective. For each of the 22 benchmarks, we compute the number of severe and moderate degradations as well as the number of improvements. *fftw* and *jpegxl* lead the top of the benchmarks with the most performance modifications, with 24 improvements and 3 moderate degradations for *fftw* and 22 moderate degradations and 6 severe degradations for *jpegxl*. On the other hand, benchmarks such as *aircrack-ng*, *botan*, *draco* and *ngspice* do not exhibit any type performance modifications.

We also present the number of affected performance tests per benchmark when LTO is enabled in Figure 6.3b. For the flags that showed the most performance degradations in the non-LTO setup, such as *fftw*, *graphics-magick*, and *jpegxl*, the LTO setup reduces degradations and increases improvements. While the benchmarks that exhibited a moderate number of performance modifications decreases as well.

In Figure 6.4a, we compare the amount of performance modification between the flags that add checks to the IR and the flags that delete UB constructs from the IR, a complete list of the flags in each category is found in Table 4.2. Only 4 flags are part of the first category, they present 3 severe degradations, 13 moderate degradations and only 1 improvement. On the other side, the flags that delete information from the IR present 21 severe modifications, 47 moderate modifications and no more than 22 improvements. Additionally, the flags that affect directly the optimizer present 2 severe modification, 4 moderate modifications and 1 improvement. This is a total of 17 modifications for the flags that add information and 9 modifications for the flags that delete information and 7 modifications for the last category.

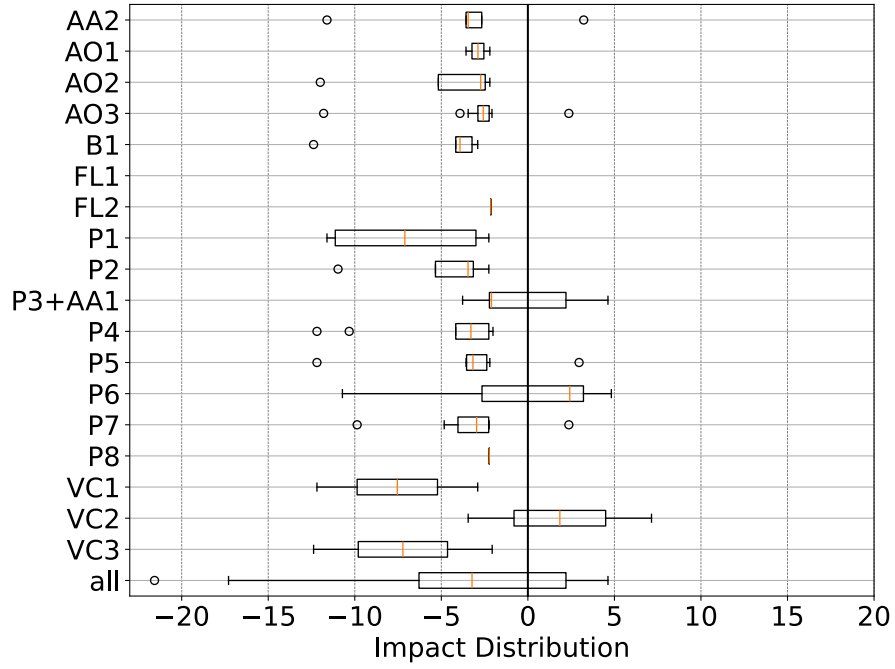
Figure 6.4b presents the impact of each flag category introduced in Table 4.2 when compiled with LTO. The number of severe degradations is visibly reduced compared to the non-LTO version in Figure 6.4a, only 4 severe degradations in the second category and 1 in the third category remain. The proportion of moderate impact remains constant, i.e. the second category still has more moderate degradations than the first and third. Also, the minimum number of improvements in a category for LTO almost reaches the maximum number of improvements in the non-LTO benchmarks.

At last, Figure 6.5 and Figure 6.6 presents a detailed view of the impact of each flag on the performance tests in our suite for both the not-LTO and LTO versions.

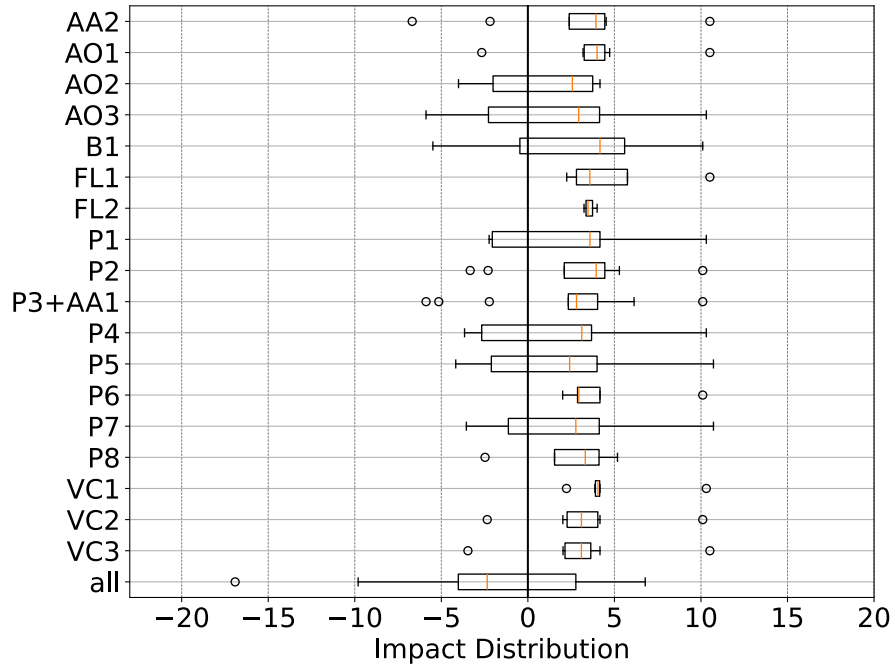
In non-LTO mode, performance tests from benchmarks such as *aom* and *zstd* present performance modifications for *all* even if when compiled with particular flags the performance does not change at all.

The performance improvements of *-fno-strict-aliasing* (P6) are also transferred to *all* in the *fftw* benchmark. Since almost no other flags impact the performance of *fftw*, *all* also benefits from the impact present in P6.

A curious case is present in the GraphicsMagick-2 performance test. Many of the individual flags are severely degrading the performance, however, when all of the flags, including the ones that cause degradations, are combined in *all*, no performance modification is present at all.

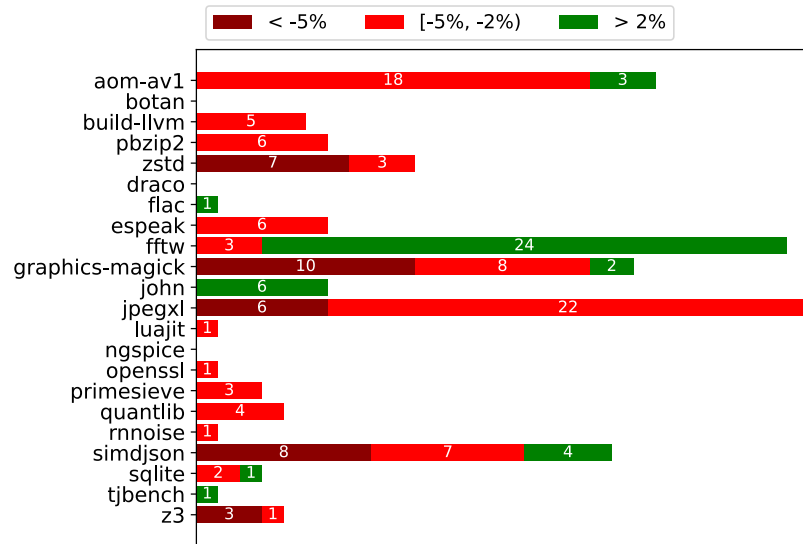


(a) non-LTO

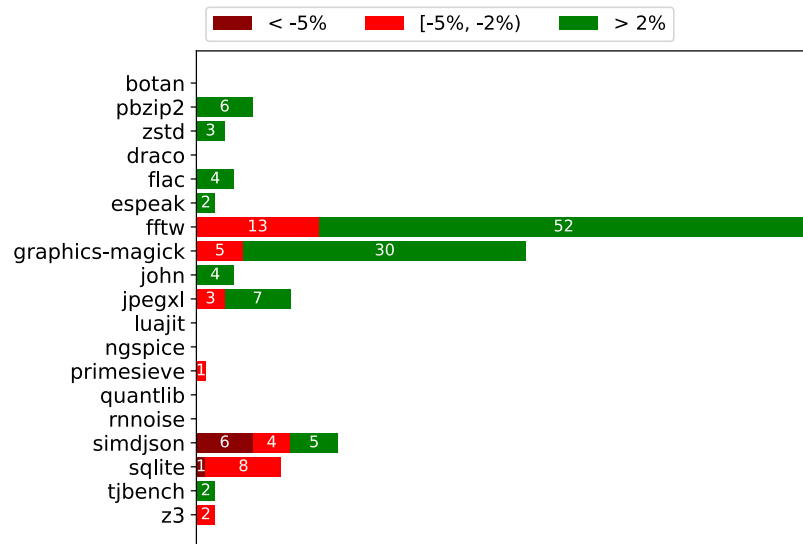


(b) LTO

Figure 6.2: Impact distribution per flag in non-LTO and LTO modes.



(a) non-LTO



(b) LTO

Figure 6.3: Quantitative performance impact of each benchmark in non-LTO and LTO modes

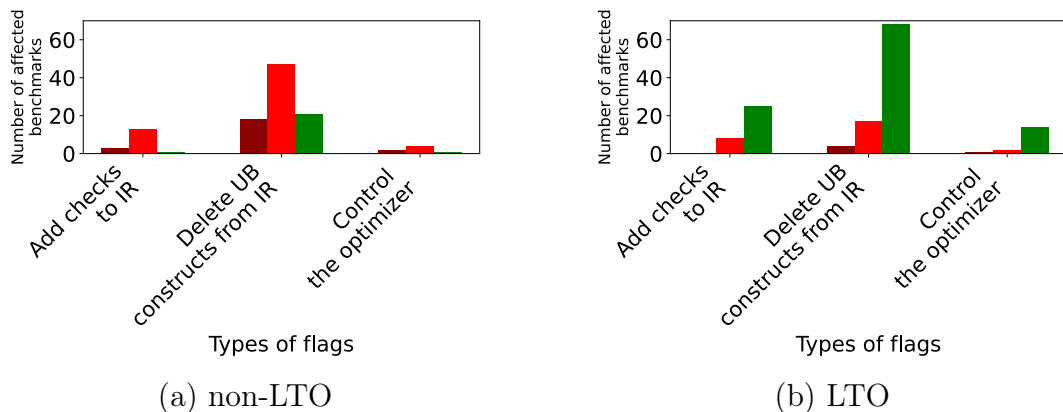


Figure 6.4: Comparison of three categories of flags: which add checks in the IR, which delete UB constructs from the IR, which control the optimizer

In LTO mode, `fftw` and `graphics-magick` contain performance tests where almost for each flag there is a performance improvement. However, in `graphics-magick`, even if for each individual UB flag there is a performance improvement, when combining all of them in the *all* flag the performance decreases. `z3` does not present any performance modifications for the individual UB flags, however, when *all* is used, the performance moderately decreases.

In summary, we presented the performance impact of disabling UB optimizations on our benchmark suite which contains 22 C/C++ applications with more than 125 performance tests. In 125 cases the performance decreases while in 42 the performance increases. When enabling LTO, to recover the lost performance, the performance degradations are reduced from 125 and 43 and the improvements are increased from 42 to 115. Next, we analyze the binary size impact of UB optimizations.

6.2 Binary Size Impact

We also look at the binary size modifications to better understand the impact of UB optimizations. Figure 6.8a and Figure 6.8b show the impact of the 19 UB flags on each of the 22 benchmarks in non-LTO and LTO mode. For more than half of the benchmarks, disabling UB optimizations does not cause a modification in neither of the 2 modes.

As presented in Figure 6.7 LTO manages to transform part of the binary size degradations into improvements. As a consequence, the number of degradations decreases. Moreover, LTO transforms the 2 improvement outliers into degradation outliers as they appear in the same benchmark, i.e. `pbzip2`. However, in both non-LTO and LTO, more than half of the benchmarks do not show any binary size modification at all.

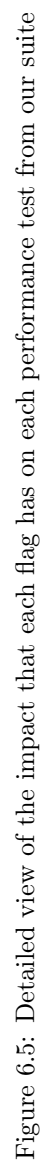


Figure 6.5: Detailed view of the impact that each flag has on each performance test from our suite

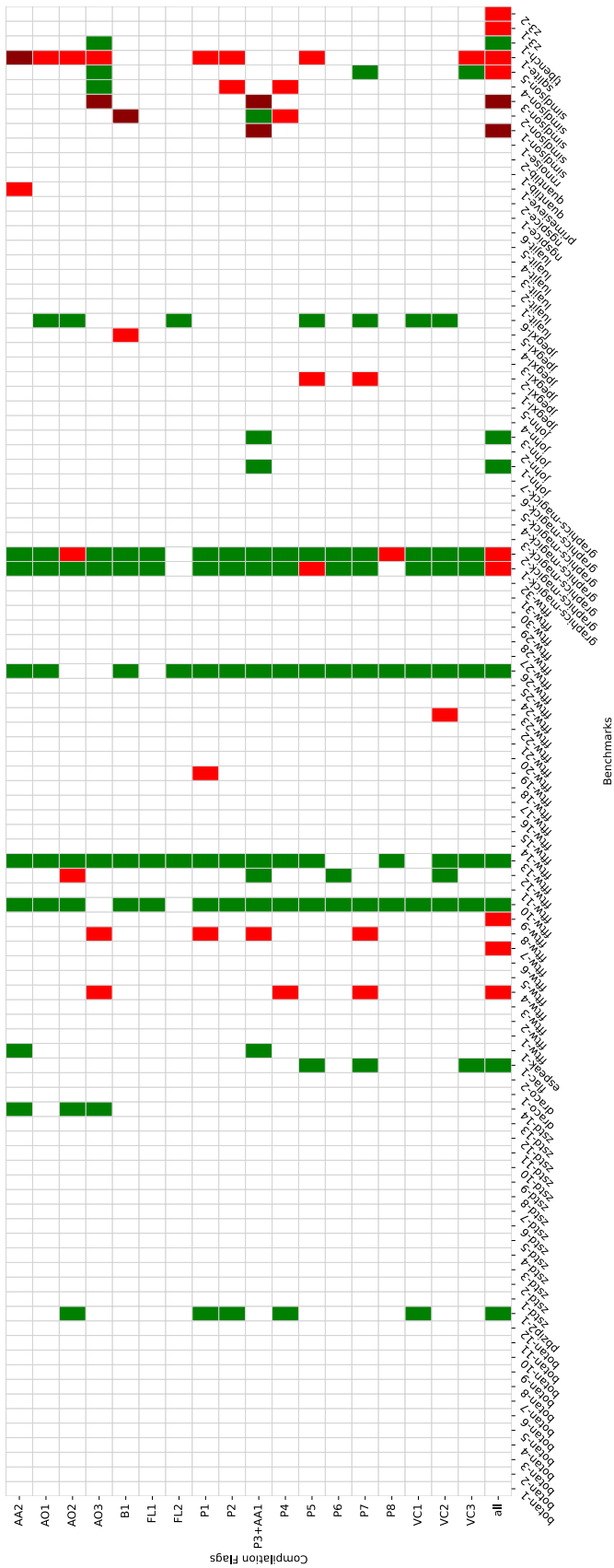


Figure 6.6: Detailed view of the impact that each flag has on each performance test from our suite when compiled with Link Time Optimizations

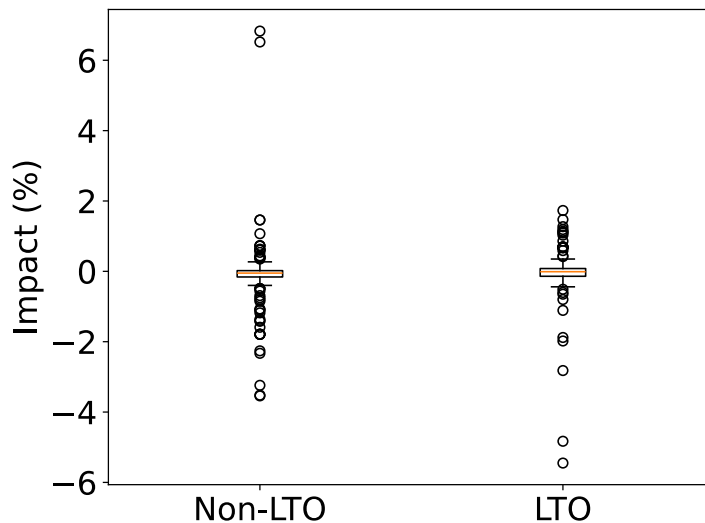


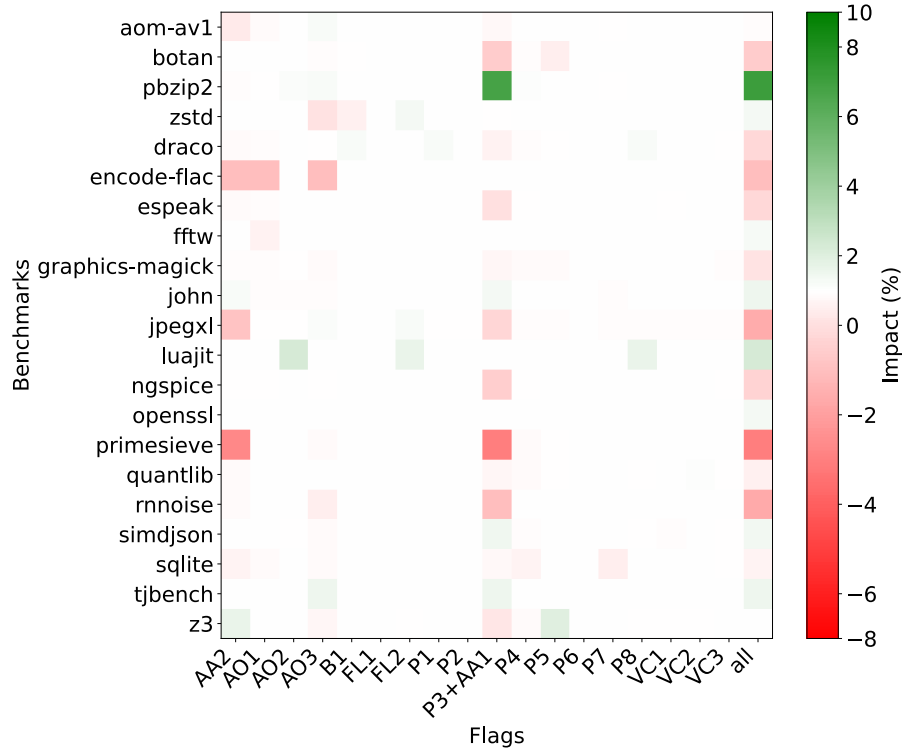
Figure 6.7: Distribution of binary size impact for non-LTO and LTO benchmarks. Positive impact means that the binary size decreases and negative impact means that the binary size increases.

In non-LTO mode, *-fdrop-inbounds-from-gep* + *-disable-oob-analysis* (P3+AA1) and *all* present the highest number of binary size degradations. However, in LTO mode, these two flags provide the highest number of binary size improvements. Furthermore, LTO presents more improvements per flag than non-LTO. We expected the binary size to get smaller with LTO because this type of optimization is known for eliminating unused functions, eliminating dead code inside functions and providing better inline decisions.

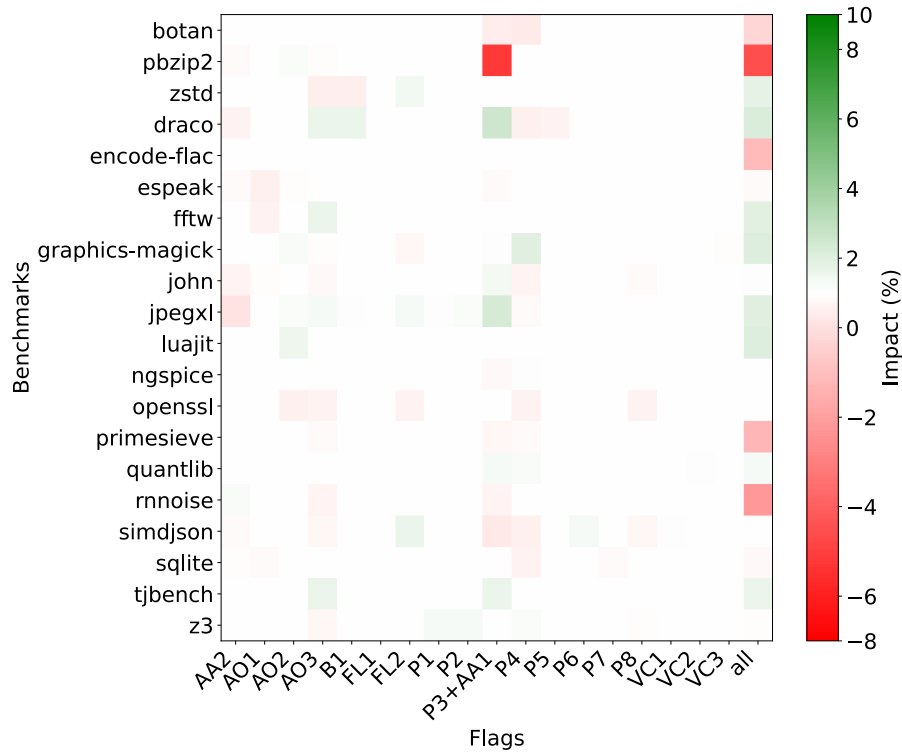
6.3 Summary

In this chapter we presented the performance impact and the binary size impact of disabling UB optimizations. In 125 cases the performance decreases while in 42 the performance increases. When enabling LTO, to recover the lost performance, the performance degradations are reduced from 125 and 43 and the improvements are increased from 42 to 115. Also, more than half of the benchmarks are not affected in terms of binary size. For the benchmarks that are negatively affected in terms of binary size, LTO can be used to reduce this number.

Next, we analyse 7 performance modifications to understand which optimizations passes from LLVM are affected by disabling UB optimizations.



(a) non-LTO



(b) LTO

Figure 6.8: Changes in the binary size of the benchmark based on UB flag. Red means binary size increase, green means binary size decrease, white means no binary size modification

Chapter 7

Performance impact analysis

In this chapter, we explore 7 of the major performance modifications caused by the flags in our benchmark suite. For each analysis presented here, we explore methods for performance recovery that do not rely on UB. In 5 out of 7 cases, the performance can be recovered using methods such as LTO, run-time checks or improved alias analysis. In 1 the performance degradation is caused by a bug in the source code. The remaining 1 case is a performance improvement.

We identified 3 categories of performance modifications in our benchmarks. First, we identified a category of performance modifications caused by an error or a bug in the source code. Benchmarks that make use of erroneous constructs, e.g. reading an uninitialized value, are part of this category. Benchmarks that show performance degradation when compiled with this flag signal that there might be a security problem caused by UB in the source code.

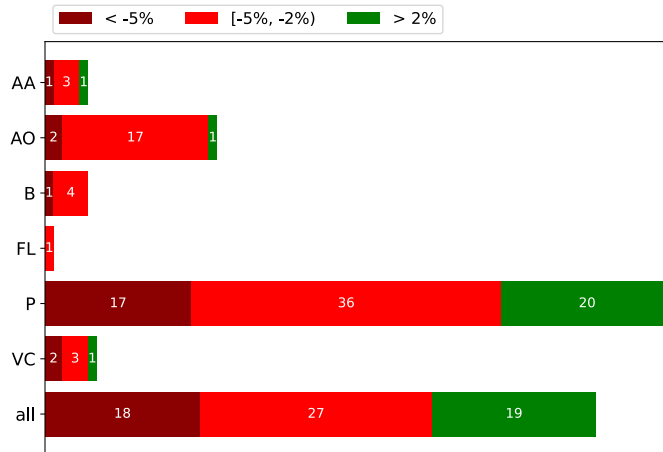


Figure 7.1: Number of benchmarks impacted either positively or negatively by each flag category presented in Table 4.1

The rest of the two categories focus on the recoverability of the missed optimization opportunities caused by our flags. As seen in Figure 7.1, enabling a flag is in 80% of cases is synonym with a performance degradation caused by a missed optimization opportunity inside LLVM. Missed optimization opportunities are caused by two sources. On one hand, the IR does not contain the constructs, which might generate UB, that we deleted which are useful for certain optimizations such as loop vectorization or speculative execution. On the other hand, the checks that we added restrict the domain of undefinedness in the program, which disables optimizations

that exploit UB, such as division by zero.

However, there are methods for recovering the performance in certain situations. Performance degradation for which a recovery method exists are placed in a category called Recoverable Optimization Opportunities. For example, a missed opportunity based on the lack of pointer attributes can be recovered using inter-procedural analysis and LTO. On the contrary, the third category, called Unrecoverable Optimization Opportunities, covers issues that are theoretically hard or impossible to recover. A summary with the flags and benchmarks that are part of each category is found in Table 7.1.

| Root cause | Flag | Benchmark | Impact |
|------------------------------------|---|-----------|--------|
| Bug in the source code | -zero-uninit-loads (VC3) | john | -1% |
| Recoverable Missed Opportunities | -fdrop-align-attr (P1) | simdjson | -13% |
| | -fdrop-deref-attr (P2) | simdjson | -13% |
| | -fdrop-inbounds-from-gep + -disable-oob-analysis (P3 + AA1) | jpegxl | -4% |
| | -fconstrain-shift-value (AO2) | zstd | -2.1% |
| | -disable-object-based-analysis (AA2) | espeak | -4.2% |
| Unrecoverable Missed Opportunities | None | None | None |
| Improvement | -fcheck-div-rem-overflow (AO1) | jpegxl | +7% |

Table 7.1: Categories of performance modifications. Each performance modification is caused by a combination between a benchmark and a flag.

The performance degradation of flags such as *-zero-uninit-loads* stems from a programming error in the C/C++ source code. In this particular case, the usage of an uninitialized variable combined with our flag triggered the addition of extra instructions used for initialization. For this reason, loops placed under the extra code became misaligned which decreased the performance of the application. Because this poses a security risk, we opened a bug in the john-the-ripper project,¹ which contains the affected benchmark.

Most of the flags present in the Recoverable Missed Opportunities category are directed to flags from category Pointers (P). The effect of all P flags in Table 7.1 is to delete pointer metadata that is susceptible to trigger UB. For example, we delete *align* and *dereferenceable* attributes from all the pointers in the program and delete *inbounds* metadata from all pointer computations. As a consequence, optimization passes such as Loop Invariant Code Motion or Loop Vectorizer fail to optimize the code because their preconditions are not fulfilled due to the missing IR information.

The Arithmetic Operations (AO) flags from Recoverable Missed Opportunities work by stopping all arithmetic operation from generating UB. AO1 adds extra instructions to guard each division and remainder operation, while AO2 adds extra instructions to guard each shift operation. However, the effect of the two flags is parallel. On one hand, AO1 increases the performance by +7% because the extra code triggers better loop alignment for the code placed below the guard instructions. On the other hand, AO2 decreases the performance because the extra instructions do not allow certain functions to be inlined.

Up until this moment, we found no performance modification that was caused by an unrecoverable missed opportunity.

The rest of the chapter focuses on detailed descriptions for each performance modification present in Table 7.1.

¹<https://github.com/openwall/john/pull/5466>

7.1 -fdrop-align-attr on simdjson

`-fdrop-align-attr` has a negative performance impact of -13% on the simdjson benchmark. When this flag is enabled, the Loop Invariant Code Motion (LICM) pass in LLVM fails to move constant instructions outside the loop. In consequence, critical loops inside the program are heavier in terms of executed instructions. However, the performance can be recovered for this benchmark using LTO and inter-procedural analysis.

This flag is added as an attempt to discard undefined behavior related to misalignment of pointers. In LLVM, if the alignment declared using the `align` attribute does not match the real alignment of the pointer, the behavior is undefined. Removing the attribute ensures that UB is not generated, at the cost of performance degradation. We have seen a performance degradation in all the benchmarks that are affected by this flag.

In simdjson, the performance hotspot is placed in a function called `partial_tweets::simdjson_ondemand::run`. When the flag is enabled, the content of this function causes the performance to drop. Listing 7.1 presents the full prototype of the function. The prototype is particularly interesting because we observed that only by removing alignment information from the first argument of the function while preserving the alignment to the rest of the arguments decreases the performance.

Listing 7.1: Name of the function where the performance degradation takes place

```
partial_tweets::simdjson_ondemand::run(
    simdjson::padded_string &this,
    std::vector<partial_tweets::tweet<std::basic_string_view<char,
        std::char_traits<char>>> json,
    std::allocator<partial_tweets::tweet<std::basic_string_view<char,
        std::char_traits<char>>>>& result
)
```

The first argument of the function controls indirectly an instruction inside a loop which fails to get speculatively executed. Since the other two parameters do not play a role in the computation of the not-moved pointer, this explains why the performance degradation is directly dependent on the alignment information of the first parameter.

The effect of the missing alignment information is show in Listing 7.2. In the LLVM IR code snippet, when the flag is disabled, i.e. the baseline, the 3 instructions that were previously inside the `while.body basic` block got successfully moved outside the loop in the `while.body.ph basic` block, i.e. the preheader of the loop.

However, when the flag is enabled, the instructions fail to get moved outside the loop, even though their result does not change from iteration to iteration. As a consequence, the loop becomes heavier in terms of instructions per iteration by 3 additional instructions. In both cases, the 3 instructions are dependent on the `%impl` variable which is derived from the first parameter of the function.

The movement of loop-invariant instructions outside the loop is commonly referred to as hoisting. LICM uses two conditions to assess the validity of the hoisting operation on an instruction. The first condition focuses on the dereferenceability information while the second focuses on alignment information.

Listing 7.2: Diff on the *run* function between baseline and *-fdrop-align-attr*

```

- -fdrop-align-attr disabled
+ -fdrop-align-attr enabled

; Preheader
while.body.ph:
    %impl = gep inbounds %"ondemand::parser", ptr %this, i64 0, i32 1
    ...
- %39 = load ptr, ptr %impl, align 8
- %var1 = gep inbounds %"dom_parser_impl", ptr %39, i64 0, i32 1
- %var2 = gep inbounds %"dom_parser_impl", ptr %39, i64 0, i32 2
    br label %while.body

; Loop:
while.body:
    ...
+ %50 = load ptr, ptr %impl, align 8
+ %var1 = gep inbounds %"dom_parser_impl", ptr %50, i64 0, i32 1
    ...
+ %var2 = gep inbounds %"dom_parser_impl", ptr %50, i64 0, i32 2

```

The first condition checks that the pointer that is part of the instruction contains bytes that are dereferenceable. If the pointer contains no dereferenceable bytes, then moving the instruction outside the loop would introduce UB earlier in the program, which is not desirable. The second condition checks that the pointer has at least the same alignment as the alignment of the instruction that uses it. Otherwise, loading or storing from a misaligned pointer causes UB.

In our case, the second condition was failing because the pointer had no alignment information. In consequence, the comparison with the alignment of the instruction failed and the hoisting operation was not executed.

Practically, on architectures which allow unaligned loads or stores, such as x86, hoisting the instruction is allowed. To validate this, we recreated the hoisting example from `simdjson` with a minimal example presented in Listing 7.3.

We first manually hoisted the `"%0 = load ptr, ptr %this, align 8"` outside the loop. Then to validate that on x86 the same assembly code is generated regardless of the presence of the alignment information, we removed the alignment of the hoisted instruction. Indeed, there was no difference between the code with the hoisted instruction and the alignment information present and the code with the hoisted instruction and the alignment information absent ¹. Hence, for x86 the hoisting operation is valid.

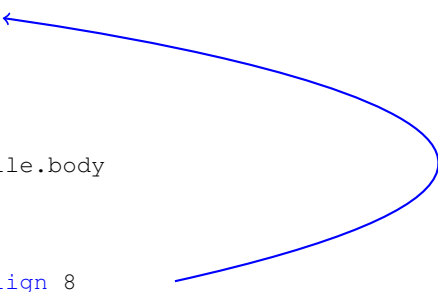
¹<https://godbolt.org/z/1Tq9rqW31>

Listing 7.3: Hoisting of the %0 instruction from while.body to entry

```
define i1 @foo(ptr align 8 %this, i1 %cond) {
entry:
    [
        br label %while.cond
    ]

while.cond:
    ...
    br i1 %cond, ..., label %while.body

while.body:
    ...
    %0 = load ptr, ptr %this, align 8
    %1 = load i32, ptr %0, align 4
    %cmp2 = icmp eq i32 %1, 0
    br i1 %cmp2, ..., label %while.cond
```



As a result, we opened a bug report in LLVM, explaining the missed optimization opportunity.¹ After discussing with the LLVM team on this matter, we learned that it is not common for the pointers generated by Clang to have present dereferenceable information but absent alignment information. Currently, Clang takes care to add both attributes to all the pointers in the program.

To infer the alignment information we used the inter-procedural analysis offered by LTO optimizations. Using this type of analysis, pointer attributes can be deduced by inspecting the interaction between the functions that use the respective pointers. We observed an improvement of +4.5% when the program was compiled with LTO and *-fdrop-align-attr* was enabled vs the baseline which was also compiled with LTO. This did not restore all the performance lost by enabling the flag, however it proved that inter-procedural analysis can save part of the performance lost due to missing pointer alignment information.

Figure 6.5 shows that for the *simdjson* benchmarks there is a one-to-one correspondence between the results for *-fdrop-align-attr* and *-fdrop-deref-attr*, i.e. P1 and P2. This is not arbitrarily, both flags trigger the same missed optimization opportunity in LICM, i.e. the inability to hoist instructions. We also experimented with the recovery of the performance using LTO on *-fdrop-deref-attr*. The regained performance percent is the same as for *-fdrop-align-attr*, i.e. +4.5%.

Furthermore, based on the results presented in Chapter 6, all flags in the Pointers (P) category, which *-fdrop-deref-attr* is part of, benefit from a reduction in the performance degradations when built with LTO. Since for most of the flags in the P category we delete pointer attributes, the inter-procedural analysis issued by LTO manages to recover some of the lost attributes. Hence, part of the performance is recovered.

7.2 *-fdrop-inbounds-from-gep* + *-disable-oob-analysis* on *jpegxl*

-fdrop-inbounds-from-gep and *-disable-oob-analysis* have a negative performance impact of -4% on the *jpegxl* benchmark. When the flags are enabled, the Loop Vectorizer pass in LLVM fails to vectorize certain loops because it cannot compute their number of iterations. Consequently, the loops take longer to execute. However, the performance can be recovered using run-time checks which prove that the loops can be vectorized.

¹ <https://github.com/llvm/llvm-project/issues/80616>

We use a combination of two flags to cover the whole range of UBs related to pointers that overflow the limits of the objects they point to. This type of accesses are referred as out-of-bounds (OOB) accesses. The first flag assures that the UB-generating *inbounds* attribute is absent from pointer computations using *getelementptr* (GEP) instructions and the second flag assures that alias analysis based on OOB pointers is disabled.

Figure 6.5 shows that for the most benchmarks that this flags affect, the performance decreases. For 9 benchmarks the performance decreases while for only 4 benchmarks the performance increases.

For this benchmark, we have observed that only by compiling *jpegxl* with the first flag, i.e. *-fdrop-inbounds-from-gep*, the performance decreases. The performance hotspot of this benchmark is found in the function called *jxl::FindTextLikePatches* which contains more than 20 loops. The primary issue is that loops whose induction variables depend on GEP instructions without *inbounds* information do not get vectorized.

A simplified example of such a loop is presented in Listing 7.4. In the preheader of the loop, the *%init* variable contains the initialization value of the induction variable. Then, inside the loop, the exit condition is derived from a GEP instruction that does not contain *inbounds* information. Because of this, LLVM cannot compute the exit count of the loop which is essential for vectorization.

To decide if the loop is vectorizable, LLVM uses Scalar Evolution (SCEV) analysis on the instructions that are part of the exit condition, i.e. *%c* in our example. When SCEV translates LLVM instructions into its internal representation, i.e. SCEV expressions, *inbounds* plays an important role on the final result. If *inbounds* is present then the SCEV expression will contain the no-self-wraparound (NW) flag. This flag indicates that the induction variable will never reach its initial value when the step is non-zero.

On the contrary, if the NW flag is absent the induction variable might reach its initial value risking to create an infinite loop. Because infinite loops cannot be vectorized, LLVM fails in optimizing the loops from *jxl::FindTextLikePatches* that contain induction variables derived from GEP instructions without *inbounds* information.

Listing 7.4: Simplified version of IR that causes vectorization problems

```
preheader:
  ...
  %init = ...
  br %loop

loop:
  %ptr = phi [%inc, %loop], [%init, %preheader]
  ...
  %inc = gep inbounds ptr, ...
  %c = cmp %inc, %end
  br %c, %loop, %exit

exit:
  ...
```

To allow the loop optimization, run-time checks are required to ensure that the loop satisfies all the preconditions for vectorization. Using the checks, we can create both the vectorized variant of the loop and the unvectorized one. Then, at run-time the proper loop variant will be selected. If the loop is provably finite at run-time, then the vectorized variant is chosen, otherwise the unvectorized variant is chosen.

For selecting the correct loop at run-time, the following formula can be used:

$$init \% increment_size == 0 \ \&\& \ end \% increment_size == 0$$

If the condition is true then we can select the vectorized loop otherwise we stick to the unvectorized one. Essentially, we guarantee that the loop terminates if the loop induction variable starts from a value that divides exactly by the increment size and ends at a multiple of the increment size. The sequence of values for the loop induction variable can be visualized as follows: $init \% increment_size == 0$, $init + increment_size * 1$, $init + increment_size * 2$, ..., $init + increment_size * n$ ($== end$)

However, this solution increases the binary size of the application. For every loop that could not be vectorized because of the missing *inbounds* information the following 3 components will be added:

- The check for selecting the type of loop at run-time
- The vectorized loop
- The unvectorized loop

To sum up, the presence of *inbounds* is critical so that loops can be vectorized. If the attribute is absent, then LLVM cannot compute the number of iterations of the loop, hence vectorization is not possible. Nevertheless, the vectorization can still be enabled using run-time checks.

7.3 -fcheck-div-rem-overflow on jpegxl

Figure 6.5 shows that for *-fcheck-div-rem-overflow* (AO1) there is no performance improvement. However, we also tested the flag on a 1 x Intel Core i7-8850H CPU @ 2.60GHz (uarch: SkyLake) machine bundled with 32GB DDR4 RAM (@ 2400 MHz) running Debian GNU / Linux 12 (bookworm) where we have seen an improvement of +7% on the jpegxl benchmark.

The additional code that this flag adds in the binary indirectly affects the alignment of loops that follow the extra code. As such, small loops, with the size between 16 and 31 bytes which are initially aligned to 16 bits, become aligned to 32 bits, which increases the performance of the Instruction Decode Queue (IDQ) inside the CPU.

Initially, our hypothesis was that this flag will decrease the performance because it adds more instructions in the final binary. For each division or remainder operation a check similar to the one in Listing 7.5 is added. However this theory was rejected because the extra instructions provided better alignment for loops placed under our extra code. On micro-architectures such as SkyLake the performance of the IDQ increases if the decoded loop is small enough and is properly aligned.

Listing 7.5: Equivalent C code for the checks added before division and remainder

```
if (b == 0 || (a == INT_MIN && b == -1))
    abort();
else
    a [/ or %] b;
```

We observed this effect on a single loop in our program. The loop, with the size of 18 bytes, was placed in a compiler-generated destructor that also acted as the performance hotspot of our application. Normally, when the flag was disabled, the alignment of the loop was declared to 16 bits, however when extra code coming from our flag was generated, the loop became aligned to 32 bits.

We reproduced this behavior with a minimal example presented in Listing 7.6 and Listing 7.7. In both cases the size of the loop is equal to 18 bytes, however, In Listing 7.7 the address of

the first instruction of the loop is artificially increased from 16 bits to 32 bits using a NOP instruction of size 10 bytes.

Listing 7.6: Loop of 18 bytes aligned to 16 bits

```
11b0: test    %rsi,%rsi
11b3: jle     11c8 <test+0x18>
11b5: lea     -0x1(%rsi),%rax
11b9: cmp     %edx,-0x4(%rdi,%rsi,4)
11bd: mov     %rax,%rsi
11c0: jne     11b0 <test>
11c2: mov     $0x1,%eax
11c7: ret
11c8: mov     $0xffffffff,%eax
11cd: ret
```

Listing 7.7: Loop of 18 bytes aligned to 32 bits using NOPs

```
11d0: cs nopw 0x0(%rax,%rax,1)
11da: nopw    0x0(%rax,%rax,1)
11e0: test    %rsi,%rsi
11e3: jle     11f8 <test+0x28>
11e5: lea     -0x1(%rsi),%rax
11e9: cmp     %edx,-0x4(%rdi,%rsi,4)
11ed: mov     %rax,%rsi
11f0: jne     11e0 <test+0x10>
11f2: mov     $0x1,%eax
11f7: ret
11f8: mov     $0xffffffff,%eax
11fd: ret
```

The effect of the alignment on the performance of the IDQ is presented in Figure 7.2. We collected IDQ statistics using the Linux perf tool focusing on the throughput of the Decode Stream Buffer (DSB), which represents the uops cache of IDQ. The plot effectively presents that by increasing the alignment of the loop, the number of DSB cycles is reduced while providing the same number of decoded uops. The number of any uops decreases by almost a half while the number of 4-uops batches remains the same, this signals that IDQ makes better use of caching when the alignment of the loop is 32 bits.

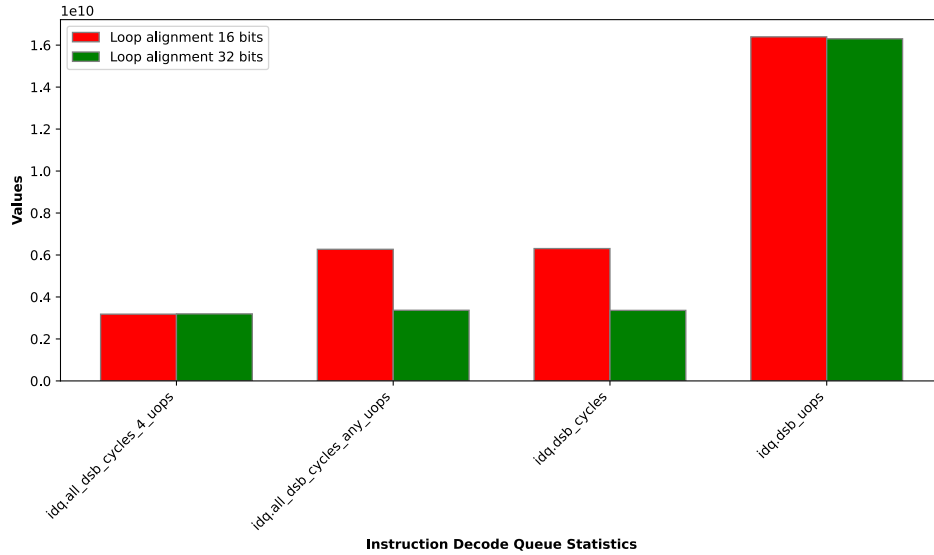


Figure 7.2: Instruction Decode Queue statistics comparison between 16-bits and 32-bits aligned loop. The number of different types of cycles decreases for 32-bits loops while the uops throughput remains the same.

Benchmarking the loops presented in Listing 7.6 and Listing 7.7 on the SkyLake machine showed a performance improvement of +12% for the 32 bits aligned loop compared to the same 16 bits aligned loop. Similar behavior can be observed on the Sandy Bridge microarch. However, on micro-architectures such as Westmere or IvyBridge we have not seen a difference.

We reported these numbers to the LLVM team¹ and also found an older report² where the difference between the 16 bits and 32 bits aligned loops is greater than the one we observed. In the other issue `all_dsb_cycles_4_uops` is 500 times better and `dsb_cycles` is 13 times better.

The author of the older issue proposed multiple heuristics for aligning this kind of loops when generating machine code, the cases that the author notes are:

- They [the loops] are innermost;
- Size of loop mod 32 is between 16 and 31 (only in this case alignment by 32 will strictly reduce the number of 32 window crossings by 1);
- (Optional) The loop is small, e.g. less than 32 bytes;
- (Optional) We could make even sharper checks trying to ensure that all other conditions of DSB max utilization are met (may be very complex!)

At the moment, LLVM does not integrate any of these heuristics, however users are able to specify the alignment of the loop in multiple ways. One method is to declare the alignment directly from C/C++ using the `[[clang::code_align(N)]]` attribute.³ Another way is to use the `-x86-experimental-pref-innermost-loop-alignment` or `-branches-within-32B-boundaries` compiler flag to align all the loops in the program.

In summary, the addition of extra instructions for guarding the division and remainder operation produces a positive performance impact on micro-architectures such as SkyLake and Sandy Bridge, while no performance impact is seen on Westmere and IvyBridge. Small loops, between 16 and 31 bytes, that follow the extra checks are aligned to 32 bits which consequently improves the performance of the Instruction Decode Queue because the Decode Stream Buffer is able to provide the cached instructions in less cycles.

7.4 -fconstrain-shift-value on compress-zstd

`-fconstrain-shift-value` has a negative performance impact of -2.1% on the compress-zstd benchmark. The extra code that this flag adds for protecting shift instructions restricts the inliner to take effect on functions that contain shifts.

As seen in Figure 6.5 this flag affects 4 benchmarks. In one of the benchmarks, i.e. compress-zstd, the guarding instruction added before each shift cause certain functions to not be inlined. We have observed that the inability of LLVM to inline the `ZSTD_getOffsetInfo` function, which contains 4 shift operations, causes a negative performance impact.

We have observed that the computed inline threshold for the function is 45. When the flag is disabled the cost of `ZSTD_getOffsetInfo` is 40, but when the flag is enabled the cost of the function increases to 50.

However, this is not a severe issue. The lost performance can be recovered either by using the `always_inline` attribute to `ZSTD_getOffsetInfo` directly in the source code or tweaking the inline heuristics of LLVM.

7.5 -zero-uninit-loads on john-the-ripper

`-zero-uninit-loads` has a negative performance impact of -1% on the john-the-ripper benchmark. Normally, we ignore performance impact in the range [-2, +2] because we consider it noise.

¹<https://github.com/llvm/llvm-project/issues/89937>

²<https://lists.llvm.org/pipermail/llvm-dev/2021-January/148177.html>

³<https://releases.llvm.org/18.1.0/tools/clang/docs/ReleaseNotes.html#attribute-changes-in-clang>

However, this performance impact is strongly tied to a security risk discovered using this flag. We discovered that in `john-the-ripper` an uninitialized read is performed on certain code paths.

This flag only affects 2 benchmarks, besides `john-the-ripper`, `graphics-magick` is also affected, we suspect that `graphics-magick` also contains an unintended uninitialized read in its codebase. The effect of this flag is to replace all *undef* values that are resulted from loading an uninitialized value with 0. Security-wise, loading an uninitialized value is very dangerous as it can render the program meaningless.

`john-the-ripper` only contains a function that makes use of uninitialized read, namely `scan_central_index`. A reduced snippet of the function is presented in Listing 7.8. If the else branch is not taken, `ctx.archive.zip64` contains an uninitialized value in the second if statement.

Listing 7.8: Uninitialized variable `ctx` used inside `scan_central_index`

```
static void scan_central_index(const char *fname)
{
    zip_context ctx;
    ...
    if (this_disk != 0 || cd_start_disk != 0) {
        ...
    } else {
        ctx.archive.zip64 = zip64;
        break;
    }
    ...
    if (ctx.archive.zip64) {
        ...
    }
}
```

Because we replace the *undef* value resulted from reading `ctx.archive.zip64` with a 0, the `JumpThreading` pass in LLVM creates a specialized version of the control flow that is optimized for `ctx.archive.zip64 == 0`. In consequence, 33 more bytes which represent instructions are added in the final binary. The extra instructions, as in the case of `-fconstrain-shift-value` on `jpegxl`, affect the alignment of the loops.

However, as opposed to `-fconstrain-shift-value`, the extra instructions decrease the performance.

One curious thing about the impact of this flag is that the `scan_central_index` is not even used at run-time, however, it affects the performance of the application because of the extra instructions that it adds in the final binary.

Since this performance impact also signals a security problem, we have opened an issue in the `john-the-ripper` project.¹ The solution for solving this problem is to simply initialize the structure that contains the possibly undefined value. We discovered that this problem can be as well caught by static analyzers such as `clang-tidy` which report the usage of possibly uninitialized variables in the program before they reach the optimization pipeline of LLVM.

Since this is a bug in the source code, we employ no methods for recovering the lost performance.

Summarizing, the negative performance impact created by `-zero-uninit-loads` stems from a bug in the source code. More code is added in the final binary as a mechanism to avoid an uninitialized read in the program. The extra code breaks the alignment of subsequent loops which creates a performance degradation. However, the security implications of this flag are more important hence we decided to open an issue in the `john-the-ripper` project which we also fixed.

¹<https://github.com/openwall/john/pull/5466>

7.6 -disable-object-based-analysis on espeak

-disable-object-based-analysis has a negative performance impact of -4.2% on the espeak benchmark. When this flag is enabled, loops that contain expressions which make use of multiple objects at the same time cannot be vectorized. To recover the performance, better alias analysis which do not rely on UB need to be implemented.

The effect of this flag is to disable alias analysis that is based on the comparison between the objects that two pointers are referring to. Virtually, the consequence of enabling this flag is to create a flat memory model where objects can overlap without causing undefined behavior. As Figure 6.5 shows, in 4 cases the performance decreases when the flag is used while in only 1 case the performance increases.

Because, when the flag is enabled, aliasing information cannot be generated simply by comparing the objects resulted by dereferencing two pointers, loops that contain expressions which make use of such objects cannot be vectorized by LLVM. For example, inside the PeaksToHarmspect function, LLVM fails to vectorize the second loop in Listing 7.9.

Listing 7.9: espeak-1.0.7 loop where performance degradation happens

```
#define N_LOWHARM 30
static int harm_inc[N_LOWHARM];
static int *harmspect;

#define MAX_HARMONIC 400
static int hspect[2][MAX_HARMONIC];

int PeaksToHarmspect(wavegen_peaks_t *peaks, int pitch, int *htab, int
control) {
    ...
    /* hmax can be at most MAX_HARMONIC */
    for (h = 0; h <= hmax; h++)
        htab[h] = 0;
    ...
    /* Loop that cannot be vectorized */
    for (h = 1; h < N_LOWHARM; h++)
        harm_inc[h] = (htab[h] - harmspect[h]) >> 3;
}
```

When PeaksToHarmspect is called, htab points to one of the two values inside hspect, while harmspect points to the remaining value. For example if htab points to hspect[1], then harmspect will point to hspect[0]. Then, before the unvectorized loop is reached, multiple operations involving htab are executed in PeaksToHarmspect. An example of such an operation is presented in the first loop from Listing 7.9. These operations utilize the whole size of the array, which is MAX_HARMONIC.

When the unvectorized loop is reached and the flag is enabled, LLVM fails to prove that the RHS of the expression inside the loop, i.e. (htab[h] - harmspect[h]) >> 3, and the LHS, i.e. harm_inc[h], do not alias. Proving this property is crucial for vectorizing the loop, otherwise the vectorization is not valid.

Inside the second loop, at most N_LOWHARM elements are accessed inside all the arrays. While N_LOWHARM is the limit for the array in LHS, for the arrays in RHS, N_LOWHARM is less than the actual limit which is MAX_HARMONIC. Since htab accesses memory at an index greater than N_LOWHARM before reaching the second loop and undefined behavior would be triggered if harm_inc would access memory at an index greater than N_LOWHARM then LLVM concludes that RHS and LHS cannot alias.

However, our flag prohibits LLVM to use the fact that UB would be triggered if `harm_inc` would access memory at an index greater than `N_LOWHARM`, hence LLVM cannot conclude that RHS and LHS do not alias. The effect of this change is presented in Listing 7.10 and Listing 7.11. On one hand, Listing 7.10 presents the vectorized form of the loop where 4 elements are fetched at the same time from `htab` and `harm_spect`. On the other hand, in Listing 7.11 elements need to be fetched individually because the memory they point to might change on the next iteration.

Listing 7.10: Optimal IR for loop in Listing 7.9

```
%43 = load <4 x i32>, ptr %arrayidx165.4, align 4
%44 = load <4 x i32>, ptr %arrayidx167.4, align 4
%45 = sub <4 x i32> %43, %44
%46 = ashr <4 x i32> %45, <i32 3, i32 3, i32 3, i32 3>
store <4 x i32> %46, ptr %getelementptr inbounds ([30 x i32], ptr @harm_inc,
        i64 0, i64 5), align 4
```

Listing 7.11: Unoptimal IR for loop in Listing 7.9

```
%arrayidx165.5 = %getelementptr i32, ptr %htab, i64 6
%43 = load i32, ptr %arrayidx165.5, align 4
%arrayidx167.5 = %getelementptr i32, ptr %33, i64 6
%44 = load i32, ptr %arrayidx167.5, align 4
%sub168.5 = sub i32 %43, %44
%shr169.5 = ashr i32 %sub168.5, 3
store i32 %shr169.5, ptr %getelementptr inbounds ([30 x i32], ptr @harm_inc,
        i64 0, i64 6), align 8

%arrayidx165.6 = %getelementptr i32, ptr %htab, i64 7
%45 = load i32, ptr %arrayidx165.6, align 4
%arrayidx167.6 = %getelementptr i32, ptr %33, i64 7
%46 = load i32, ptr %arrayidx167.6, align 4
%sub168.6 = sub i32 %45, %46
%shr169.6 = ashr i32 %sub168.6, 3
store i32 %shr169.6, ptr %getelementptr inbounds ([30 x i32], ptr @harm_inc,
        i64 0, i64 7),
```

To recover the performance, we attempted to use the SCEV alias analysis of LLVM.¹ However, this type of AA was not able to vectorize the loop by proving that RHS and LHS are not aliasing. Since AA results can be improved in LLVM as shown by Hückelheim et al [29] and current research is done in this field [41, 16], it is possible that future version of LLVM can vectorize the loop in the absence of UB.

In summary, when the flag is enabled a performance degradation of -4.2% is triggered because LLVM cannot vectorize loops that load and store to objects which cannot be proved to not alias. Since LLVM cannot prove that the objects do not alias, vectorization is not allowed to be performed. However, to recover the performance, better alias analysis results which do not rely on UB must be generated, which we were not able to get with Clang 16.

7.7 Discussion

In this chapter we presented 7 performance modifications that resulted from disabling undefined behavior exploitations in LLVM. In all situations but one the modification resulted in a negative performance impact. The positive performance impact of +7% was caused by a micro-architecture specific feature in the Instruction Decode Queue of the CPU.

¹<https://llvm.org/docs/AliasAnalysis.html#the-scev-aa-pass>

| LLVM opt pass | Flag | Benchmark | Impact recovery |
|----------------------------|--------------------------------------|-----------|--------------------------|
| Loop Invariant Code Motion | -fdrop-align-attr (P1) | simdjson | LTO |
| | -fdrop-deref-attr (P2) | simdjson | LTO |
| Inliner | -fconstrain-shift-value (AO2) | zstd | Tweak inliner heuristics |
| Loop Vectorizer | -disable-object-based-analysis (AA2) | espeak | Improve alias analysis |
| | -fdrop-inbounds-from-gep (P3) | jpegxl | Add run-time checks |

Table 7.2: LLVM optimization passes that create the performance modifications. Only modifications from the "Recoverable Missed Opportunities" category are presented.

For each negative performance impact we offered a root cause analysis and methods for recovering the lost performance. In all the situations that we encountered the performance was recoverable either using Link Time Optimizations or employing theoretical solutions.

During our analysis, we also discovered an application bug inside the john-the-ripper benchmark. Using the *-zero-uninit-loads* flag we discovered a read of an uninitialized value that we reported and fixed. The activation of the flag also produced a performance degradation of -1%.

2 out of the 5 performance degradations, i.e. *-disable-object-based-analysis* and *-fdrop-inbounds-from-gep*, are caused because LLVM is not able to vectorize critical loops in the benchmarks. However, the performance can be recovered either by adding run-time checks that help LLVM vectorize the loop or by providing better alias analysis mechanisms.

2 of the 5 performance degradations, i.e. *-fdrop-align-attr* and *-fdrop-deref-attr*, are caused because LLVM is not being able to move loop invariants outside loops. The inability to move the instructions is caused by the lack of pointer metadata on the loop invariants. However, the performance can be recovered using Link Time Optimizations which generate the missing metadata using inter-procedural analysis.

Lastly, the final performance degradation is caused by the lack of inlining. *-fconstrain-shift-value* add extra instructions to each function that makes use of shifting which breaks the inlining heuristics inside LLVM. However, the performance can be recovered by tweaking the inlining thresholds of LLVM.

In our analysis, we have found no case where the performance could not be recovered either practically or theoretically. In this context, it is possible to have compiler optimizations that do not rely on UB as the same performance or part of the same performance can be regained using other techniques. This result can lead to more secure applications in terms of UB while still being able to optimize applications.

Chapter 8

Conclusion and Future Work

C/C++ programs benefit from the performance improvements provided by optimizing compilers such as Clang/LLVM. However, these compilers are known to use UB to issue optimizations. Since UB is a dangerous concept that can lead to security vulnerabilities, it is important to also analyze its role from a performance perspective to understand the associated tradeoffs.

We present the first study that thoroughly analyzes the performance impact of UB in C/C++. Using 19 flags that disable UB optimization and a benchmark suite of 22 benchmarks with more than 125 performance tests, we show that in 125 cases the absence of UB creates a performance degradation and in 42 cases it causes a performance improvement.

We also offer methods to recover the lost performance. By using link-time optimizations (LTO), we reduce the number of degradations from 125 to 43 and increase the number of improvements from 42 to 115. LTO offers inter-procedural program analysis that recovers part of the lost information when UB optimizations are disabled.

We also analyzed 6 performance degradations and 1 performance improvement. For all degradations we managed to recover the lost performance using LTO or other techniques. To further increase the performance we propose theoretical techniques such as addition of run-time checks and improvement of alias analysis results. As part of the performance analysis, we opened 2 LLVM performance issues and 1 security bug in *john-the-ripper*.

In this study we showed that it is possible to have a compiler design that does not use UB optimizations at a small performance cost. The cost can be further reduced by using better alias analysis and allowing run-time checks to be introduced. This results in more secure applications in terms of UB at a small performance cost.

8.1 Future Work

Currently, we analyzed the impact of UB optimizations only on a hardware platform and for a single baseline, i.e. -O2. To better understand the impact, this study must be conducted on other hardware platforms such as ARM and AMD, with more baseline flags, such as -O3, -Os, -Oz. Having 3 available platforms with 4 baselines results in 12 different configurations. Given that we have more than 2500 results for a single platform and a single baseline, this would result in more than 30k results which will provide a better view of the performance impact.

Next, we analyzed 19 flags in this study, however, there are still flags that we did not benchmark, for example *-ftrivial-auto-var-init* which initializes automatic variables to 0. Compared to *-zero-uninit-loads* which only replaces attempts to load from an uninitialized variable with 0, we expect that the first flag will have a worse performance. Besides that, we only analyzed the

performance of individual flags and the performance of combining all the flags. Further efforts must be put into analyzing different groups of flags to understand how they affect the suite.

Our benchmark suite can also be improved. It currently focuses mostly on applications that have tight processing loops. However, there are C/C++ applications that are not loop-intensive and still put accent on performance. The benchmark suite must contain these kind of applications as well in order to understand the UB impact on multiple application categories.

Finally, we only used LTO as a recovery mechanism for the lost performance, but we also proposed theoretical recovery mechanisms such as the addition of run-time checks and improving alias analysis results. Future work must focusd on implementing these theoretical results in order to asses the amount of the recovered performance.

Bibliography

- [1] C++ working draft n4928. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4928.pdf>.
- [2] C23 working draft n3220. <https://open-std.org/JTC1/SC22/WG14/www/docs/n3220.pdf>.
- [3] Correct and incorrect code, and "erroneous behaviour". <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2795r0.html>.
- [4] Fun with null pointers, part 1. <https://lwn.net/Articles/342330/>.
- [5] New llvm ‘undef’ value. <https://nondot.org/sabre/LLVMNotes/UndefinedValue.txt>.
- [6] P1705r1 enumerating core undefined behavior. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html>.
- [7] P2723r1 zero-initialize objects of automatic storage duration. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2723r1.html>.
- [8] Rationale for international standard — programming languages — C. <https://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf>.
- [9] Re: Gcc optimizes integer overflow: bug or feature? <https://lists.gnu.org/archive/html/bug-gnulib/2006-12/msg00151.html>.
- [10] Undefinedbehaviorsanitizer. <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>.
- [11] Possible undefined behavior on initialization. <https://github.com/monero-project/monero/issues/1690>, 2017.
- [12] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, techniques and tools*, 2nd editio, 2007.
- [13] C Scott Ananian. *The static single information form*. PhD thesis, Massachusetts Institute of Technology, 2001.
- [14] Jon Louis Bentley. *Writing efficient programs*. Prentice-Hall, Inc., 1982.
- [15] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 197–208, 2013.
- [16] Khushboo Chitre, Piyus Kedia, and Rahul Purandare. The road not taken: exploring alias analysis based optimizations missed by the compiler. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):786–810, 2022.
- [17] GCC community. Implementing boringcc as a new flag of gcc. <https://gcc.gnu.org/wiki/boringcc>.
- [18] LLVM Community. Langref. <https://llvm.org/docs/LangRef.html>.

- [19] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, June 2023. Volume 1: Basic Architecture, Volume 2: Instruction Set Reference, A-Z, Volume 3: System Programming Guide.
- [20] The MITRE Corporation. 2023 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.
- [21] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
- [22] 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.
- [23] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142. IEEE, 2021.
- [24] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.
- [25] Martin Anton Ertl. What every compiler writer should know about programmers. In *Programmiersprachen und Grundlagen der Programmierung*, pages 112–133. Schriftenreihe des Instituts für Computersprachen, TU Wien, 2015.
- [26] International Organization for Standardization. Iso/iec 26300:2006 open document format. http://std.dkuug.dk/keld/iso26300-odf/is26300/iso_iec_26300:2006_e.pdf, December 2006.
- [27] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming*, 15(4-5):526–542, 2015.
- [28] Vasileios Gemistos. A study on undefined behavior in C. Master’s thesis, Universiteit Van Amsterdam, 2010.
- [29] Jan Hueckelheim and Johannes Doerfert. Oraql—optimistic responses to alias queries in llvm. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 655–664, 2023.
- [30] Raphael Iseman, Cristiano Giuffrida, Herbert Bos, Erik Van Der Kouwe, and Klaus von Gleissenthall. Don’t look ub: Exposing sanitizer-eliding compiler optimizations. *Proceedings of the ACM on Programming Languages*, 7(PLDI):907–927, 2023.
- [31] Michael Larabel. Phoronix test suite. <https://www.phoronix-test-suite.com/>, 2024.
- [32] Chris Lattner. What every c programmer should know about undefined behavior. *LLVM. May*, 13, 2011.
- [33] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.
- [34] 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.
- [35] Shaohua Li and Zhendong Su. Ubfuzz: finding bugs in sanitizer implementations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 435–449, 2024.

- [36] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.
- [37] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2015.
- [38] ARM Ltd. *ARM® Architecture Reference Manual*, December 2022. ARMv8, for ARMv8-A architecture profile.
- [39] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.
- [40] Karl J Ottenstein, Robert A Ballance, and Arthur B MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 257–271, 1990.
- [41] Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. Ooelala: Order-of-evaluation based alias analysis for compiler optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 839–853, 2020.
- [42] John Regehr. A guide to undefined behavior in c and c++. *Embedded in Academia*. July, 2010.
- [43] John Regehr. Issue 633560: integer undefined behaviors in firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=633560, 2010.
- [44] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346, 2012.
- [45] Dennis Ritchie. Dennis ritchie. noalias comments to x3j11. <http://www.lysator.liu.se/c/dmr-on-noalias.html>, 1988.
- [46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.
- [47] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.
- [48] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.
- [49] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.
- [50] Linus Torvalds. Re: [isocpp-parallel] proposal for new memory_order_consume definition. <https://gcc.gnu.org/legacy-ml/gcc/2016-02/msg00381.html>, 2016.
- [51] Jean-Paul Tremblay and Paul G Sorenson. *Theory and Practice of Compiler Writing*. McGraw-Hill, Inc., 1985.

- [52] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 47–55, 1995.
- [53] 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.
- [54] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. A differential approach to undefined behavior detection. *ACM Transactions on Computer Systems (TOCS)*, 33(1):1–29, 2015.
- [55] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual*. RISC-V Foundation, volume i: user-level isa, version 2.2 edition, 2019.
- [56] Thomas Wuerthinger. Micro-benchmarking considered harmful. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 1–1, 2017.
- [57] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3655–3672, 2023.
- [58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.
- [59] 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.
- [60] Nickolai Zeldovich. Issue 17016: `_sre`: Avoid relying on pointer overflow. <https://bugs.python.org/issue17016>, 2013.
- [61] Nickolai Zeldovich. Issue 12079010: Avoid undefined behavior when checking for pointer wraparound. <https://codereview.chromium.org/12079010/>, 2016.
- [62] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.