

Exploiting Undefined Behavior in C/C++ Programs for Optimization: A Study on the Performance Impact

LUCIAN POPESCU, Politehnica University of Bucharest, Romania and INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal

NUNO P. LOPES, INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal

The C and C++ languages define hundreds of cases as having undefined behavior (UB). These include, for example, corner cases where different CPU architectures disagree on the semantics of an instruction and the language does not want to force a specific implementation (e.g., shift by a value larger than the bitwidth). Another class of UB involves errors that the language chooses not to detect because it would be too expensive or impractical, such as dereferencing out-of-bounds pointers.

Although there is a common belief within the compiler community that UB enables certain optimizations that would not be possible otherwise, no rigorous large-scale studies have been conducted on this subject. At the same time, there is growing interest in eliminating UB from programming languages to improve security.

In this paper, we present the first comprehensive study that examines the performance impact of exploiting UB in C and C++ applications across multiple CPU architectures. Using LLVM, a compiler known for its extensive use of UB for optimizations, we demonstrate that, for the benchmarks and UB categories that we evaluated, the end-to-end performance gains are minimal. Moreover, when performance regresses, it can often be recovered through small improvements to optimization algorithms or by using link-time optimizations.

CCS Concepts: • **Software and its engineering** → **Compilers; Software performance; Semantics.**

Additional Key Words and Phrases: Undefined Behavior, Compiler Optimizations, C, C++, LLVM

1 Introduction

The C programming language, now over 50 years old, carries with it a significant amount of historical legacy. One of the key design choices made by its creators was to ensure that C programs could run efficiently across all hardware platforms available at the time [9]. This required that each language construct, such as an integer addition, be translatable into a single assembly instruction in order to maximize performance on the relatively simple and slow CPUs of that era.

As a consequence of this design goal, the language was standardized to reflect only the least common denominator semantics of several CPU architectures. For instance, although two’s complement arithmetic is now universally adopted, it was not at the time. This led both C and C++ to define signed integer overflow as undefined behavior (UB), granting compilers the flexibility to map arithmetic operations to a single assembly instruction across most CPU architectures.

While all modern CPUs implement two’s complement arithmetic, theoretically allowing the C/C++ standards to define all cases of integer overflow, certain disparities between CPU architectures persist. For example, the result of a shift by an amount equal to or greater than the bitwidth varies between ARM and x86. Standardizing one behavior over the other would necessitate additional instructions for some architectures, undermining the original goal of efficiency.

In addition to handling CPU architecture differences, the C and C++ standards use UB to avoid mandating the detection of certain errors that would be too costly or impractical to detect. For

Authors’ Contact Information: [Lucian Popescu](#), Politehnica University of Bucharest, Romania and INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal, lucian.popescu187@inesc-id.pt; [Nuno P. Lopes](#), INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal, nuno.lopes@tecnico.ulisboa.pt.



This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

example, detecting out-of-bounds memory accesses would require emitting additional code, which would likely reduce performance, contrary to the languages' design goals.

Over time, compiler developers realized that UB could be leveraged for more than just eliminating a few assembly instructions. For instance, the UB in integer overflows allows compilers to optimize $a + b > a$ into $b > 0$. While this transformation is correct for pure integers, it does not hold in two's complement arithmetic. Ironically, such expressions are often incorrectly used to check if $a + b$ overflows, leading to unexpected results.

Another example where compilers take advantage of UB in integer overflows for optimization is demonstrated in the following program (shown on the left):

<pre>for (int i=0; i <= n; ++i) { a[i] = 42; }</pre>	<pre>for (int i=0; i <= n; ++i) { *(a + (long)i) = 42; }</pre>	<pre>long n2 = (long)n; for (long i=0; i <= n2; ++i) { *(a + i) = 42; }</pre>
---	---	--

When compiling the left program for a 64-bit CPU, compilers must generate assembly code that resembles the program in the middle because pointers have 64 bits while the `int` type is usually defined to have just 32 bits. This results in an implicit cast in the original program.¹ Changing the induction variable `i` to have 64 bits and removing the sign-extend cast from the loop body (program on the right) improves the performance of this loop by up to 40% in some micro-architectures. However, this transformation is only legal because integer overflow is UB, and therefore the compiler is allowed to assume there is no overflow.

Over time, compilers have evolved to exploit UB for optimization, operating under the assumption that programs are well-defined. However, real-world programs often contain bugs that can trigger UB. A notable example (below) from the Linux kernel highlights the dangers of such optimizations [10]. Since dereferencing a null pointer triggers UB, the compiler can assume that after line 4, the pointer `tun` is non-null. Consequently, the compiler optimizes away the `if` statement, creating a security vulnerability.

```
1 unsigned tun_chr_poll(struct file *file) {
2     struct tun_file *tfile = file->private_data;
3     struct tun_struct *tun = __tun_get(tfile);
4     struct sock *sk = tun->sk; // dereferences tun; implies tun != NULL
5     if (!tun) // always false
6         return POLLERR;
7     ...
8 }
```

In response to such issues, some compilers introduced flags to disable the removal of null-pointer checks, even when the compiler can infer that the check is unnecessary. While this flag could have prevented the vulnerability in the example above, such mechanisms are not a panacea. If a program triggers UB, the resulting behavior remains unpredictable, and the effect of such flags may not align with the developer's expectations.

More recently, a study by Xu et al. [60] found over 4,000 bugs related with UB in open-source software. This and other studies have led to the development of several flags in the compilers to disable or detect certain classes of UB. For example, there is a proposal to initialize local (stack) variables in C++ [5]. Reports on the performance impact range from a 4% slowdown in the Linux kernel [42] to a 13% slowdown in a benchmark suite [62].

Despite the growing interest in mitigating UB in both programming languages and compilers, there has been no comprehensive study on the impact of UB in full-scale applications using modern

¹Historically, developers learned to declare loop induction variables as `int`. Unfortunately, that results in a mismatch between the size of pointers and common array indexing types. C++ tries to avoid this situation with iterators.

hardware. Given that modern CPUs execute instructions out of order and have wide pipelines, the overhead of executing a few additional assembly instructions is often negligible.

In this paper, we present an exhaustive analysis of the various classes of UB exploited by the LLVM compiler, which is known for its extensive use of UB for optimization. We modified LLVM to selectively disable exploitation of each UB class, allowing us to measure the impact of each class on performance and code size across a variety of real-world applications.

In summary, the contributions of this paper are as follows:

- (1) A comprehensive study of the classes of UB exploited for optimization by LLVM, which is widely recognized for its extensive use of UB.
- (2) An extension to the Alive2 translation validation tool [34] to detect optimizations that rely on UB. This ensures we have thorough coverage of all UB classes exploited by LLVM.
- (3) A detailed analysis of the run-time performance and code size impact associated with exploiting each class of UB for optimization.

2 Undefined Behavior

The C and C++ standards leave many aspects of program behavior undefined. However, most of these cases are benign in practice, as compilers typically provide reasonable, consistent behavior that developers can rely on. In some situations, compilers go beyond the standard's requirements by offering multiple levels of undefined behavior (UB) with stronger guarantees.

We focus on how UB is handled by LLVM, since it is particularly well-specified and because LLVM is widely known for leveraging UB for optimization. LLVM has two main classes of UB:

- Deferred UB, which is used to define corner cases for which LLVM does not wish to specify a concrete value, while allowing the operation to be executed anywhere. For example, a signed addition can be executed speculatively, even though the result in cases of overflow is not concretely defined.
- Immediate UB, which is reserved for operations that trigger hardware exceptions and thus cannot be executed speculatively, e.g., division by zero and dereferencing a null pointer.

The rationale for having these two classes of UB is to enable optimizations that would not be possible if all UB was immediate. For example, deferred UB allows LLVM to hoist arithmetic operations out of loops easily. Furthermore, LLVM has two values for representing deferred UB: `undef` and `poison`, with the former being weaker than the latter. Passing these values to certain operations triggers immediate UB.

The way that Clang (LLVM's C/C++ frontend) maps input programs into LLVM's intermediate representation (IR) is a refinement of the semantics given in the C and C++ standards. Clang offers stronger guarantees and leaves fewer behaviors as undefined. There are three reasons for this: (1) cases that offer no potential for optimization, (2) cases where LLVM tries to offer "nicer" semantics (e.g., not forcing the input pointers to `memcpy` to be non-null even if the size argument is zero), and (3) exploiting some UB leads to many programs misbehaving due to a common coding pattern triggering UB frequently in practice (e.g., forcing a value range for enums is opt-in via `-fstrict-enums`). The difference between the last two cases is that for the last case the compiler offers a set of flags to exploit more UB.

In this study, we wanted to understand the impact of each class of UB that LLVM exploits in practice for optimization. In order to do this, we modified LLVM to allow us to disable each class of UB individually. Note that we do not attempt to detect UB either statically or at run time; we merely want to disable the exploitation of UB. This contrasts with countermeasures, such as initializing all stack values to zero, which have a higher cost.

Table 1. Flags that disable UB exploitation in Clang/LLVM. The ‘New?’ column indicates the flags that we implemented. The last column indicates whether the LLVM IR is expressive enough to implement the flag.

Category	Flags	Acronym	New?	Frontend?
Arithmetic	-fcheck-div-rem-overflow	AO1	✓	✓
Operations	-fconstrain-shift-value	AO2	✓	✓
	-fwrapv	AO3		✓
Type Ranges	-fno-constrain-bool-value	TR1	✓	✓
	-fno-strict-enums	TR2		✓
Function	-fignore-pure-const-attrs	FD1	✓	✓
Domains	-fdrop-ub-builtins	FD2	✓	✓
	-fno-finite-loops	FD3		✓
Pointers and Memory	-fdrop-align-attr	PM1	✓	✓
	-fdrop-deref-attr	PM2	✓	✓
	-fno-delete-null-pointer-checks	PM3		✓
	-fdrop-noalias-restrict-attr	PM4	✓	✓
	-fno-strict-aliasing	PM5		✓
	-fno-use-default-alignment	PM6	✓	✓
	-Xclang -no-enable-noundef-analysis	PM7		✓
	-mllvm -zero-uninit-loads	PM8	✓	
Alias Analysis	-fdrop-inbounds-from-gep	AA1	✓	
	-mllvm -disable-oob-analysis			
	-mllvm -disable-object-based-analysis	AA2	✓	

2.1 Disabling Exploitation of Undefined Behavior

Table 1 lists the five categories of UB that Clang/LLVM currently exploit for optimization. We further subdivide each category, for a total of 18 individual aspects that are exploited. We implemented 12 new flags in Clang/LLVM and use 6 existing ones to selectively disable UB.

We attempted to implement as many flags on the frontend side (Clang) as possible by changing the generated IR to be free from UB. The main reasons why we chose this approach are as follows:

- (1) It is easier to ensure the implementation does not miss some case.
- (2) LLVM is used by many languages. Any change to the IR or the optimizers can negatively impact the performance of these other languages.
- (3) It allows us to benchmark the optimizer that is common across multiple languages and thus some of the results we obtain for C/C++ may transfer to other languages.

Unfortunately, LLVM’s IR is not sufficiently expressive to implement all flags. We had to implement 3 flags directly in LLVM by changing the code of multiple static analyses and optimizations. We now describe the changes implemented by each flag. In the appendix, we show example programs affected by each flag, as well as the IR differences.

2.1.1 Arithmetic Operations. Flag AO1 instruments the IR to trap (a well-defined exit) in the cases where division and remainder would trigger UB (i.e., the divisor is zero, or the operation overflows).

Flag AO2 ensures that shift operations are well-defined by masking the shift amount so it is always smaller than the bitwidth. This matches the semantics of x86.

Flag AO3 makes signed overflow in arithmetic operations defined in term of two's complement arithmetic. In practice, this amounts to not adding the `nsw` attribute to arithmetic operations.

2.1.2 Type Ranges. Flag TR1 prevents the compiler from assuming that `bool` variables only have the integer value 0 or 1. This amounts to not emitting LLVM's `!range` metadata. Flag TR2 is similar, but for `enum`-typed variables.

2.1.3 Function Domains. Flag FD1 ignores the `__attribute__((const/pure))` annotations in the input programs. While these are not part of the C/C++ standards, many compilers support them.

Flag FD2 converts calls to `__builtin_unreachable()` into well-defined traps, and ignores calls to assume builtins. Again, these functions are not part of the C/C++ standards.

Flag FD3 prevents the compiler from assuming that all loops without side-effects terminate. This amounts to removing LLVM's `mustprogress` attribute from functions.

2.1.4 Pointers and Memory. Flags PM1 and PM2 prevent Clang from emitting, respectively, `align` and `dereferenceable` function argument attributes. For example, these are added to the "this" argument of C++ methods, so the optimizer can assume that the whole object is dereferenceable.

Flag PM3 prevents the compiler from assuming that null pointers are not dereferenceable. In particular, it prevents certain null pointer checks from being optimized away. The attribute `null_pointer_is_valid` is added to functions, the `dereferenceable` argument attributes are changed to `dereferenceable_or_null`, and the `nonnull` argument attribute is not emitted.

Flag PM4 makes Clang ignore the `restrict` keyword on pointers, i.e., it makes Clang no longer emit LLVM's `noalias` function argument attribute.

Flag PM5 prevents the compiler from using type-based alias analysis by having Clang skipping the emission of TBAA metadata.

Flag PM6 prevents the compiler from making assumptions about the alignment of data, i.e., all memory operations are emitted with alignment of 1 (i.e., they are possibly completely unaligned).

Flag PM7 prevents the compiler from assuming that function arguments and return values are well-defined values (i.e., they are not `undef` nor `poison`). In practice, this makes Clang skip emitting the `noundef` attribute.

Flag PM8 changes the value of uninitialized loads from `undef` to zero. Note that this is not the same as initializing all variables to zero! Instead, LLVM's optimizer is changed so that when it replaces such a load, it replaces it with zero instead of `undef`. This is to prevent subsequent optimizations that would take advantage of `undef` to prove that the code triggers UB, at a much smaller cost than explicitly initializing all variables. While this flag disables exploitation of UB, the semantics offered is mostly unchanged: loading uninitialized data yields a non-deterministic value, but it never triggers immediate UB.

2.1.5 Alias Analysis. These flags change the behavior of the alias analysis (AA) algorithms.

Flag AA1 prevents the compiler from assuming that the result of pointer arithmetic operations are always within bounds of the input object, as well as prevents AA from concluding no-alias for out-of-bounds accesses. This flag consists of two modifications: change Clang to emit LLVM's pointer arithmetic operations (`getelementptr`) without the `inbounds` attribute, and change the AA algorithm itself.

Flag AA2 prevents AA from using object provenance information for inference (e.g., `p + i` and `q + j` cannot alias if `p` and `q` are initialized with distinct calls to `malloc`).

Together, these flags offer a flat memory model.

2.2 Validating Coverage with Alive2

Finding all places in LLVM/Clang that exploit UB is not an easy task. It is very easy to miss some cases since these are not documented and sometimes the UB exploitation reasoning is not obvious.

To help us ensure that our UB-disabling flags cover most of the UB exploitation in LLVM/Clang, we extended Alive2 [34] with a new mode to detect what we call “guardable UB”. Alive2 is a translation validation tool that integrates with LLVM and verifies whether the optimizations done when compiling a program were correct.

Our extension to Alive2 detects IR that contains UB that could have been removed by a frontend, i.e., cases where the IR is expressive enough to allow certain UB to be switched into well-defined code. For example, division by zero is UB, but we can guard the operation like $d \neq 0 \ ? \ 0 : x / d$ to make it well-defined. Hence, we call this class of UB “guardable”.

Not all UB in LLVM IR is guardable. For example, memory operations trigger UB when dereferencing out-of-bounds pointers. While it is technically possible to guard these operation using, e.g., fat pointers, it would be very expensive to do so. Hence, we declare these as “non-guardable UB”.

Armed with this new Alive2 extension, we compiled our benchmarks to check the completeness of our flags regarding “guardable UB”. It proved to be very useful as we found several problems:

- Found several cases of UB exploitation not disabled by our flags, including missing handling of `__builtin_assume`, `__builtin_unreachable`, `assume_aligned`, and the `align` and `dereferenceable` parameter attributes. Although we did an exhaustive search for UB in the LLVM IR’s manual and in the LLVM/Clang source code, we had still missed these cases.
- Found several bugs in the benchmarks: draco uses `__attribute__((pure))` on non-pure functions. Also, mnoise uses `*(int*)0 = 0`; to generate traps, but that is just UB.
- Found two bugs in LLVM: one in the loop vectorizer, in which it would create store operations that did not respect the original program’s alignment constraints,² and another where Clang was emitting some RTTI data with the wrong size.³

We believe that developing this extension for Alive2 was very valuable to improve coverage of our work. We developed it in parallel with the flags to disable UB exploitation, so we could build confidence in both artifacts. This extension has been incorporated in Alive2 already.

3 Benchmarks

We collected a benchmark suite with 24 C/C++ programs along with their performance tests. These programs cover a wide range of domains, code size, and performance characteristics (e.g., CPU vs memory bound). Table 2 shows the list of programs, the corresponding number of lines of code, as well as the measurement scale for the benchmarks.

In total, we compiled 7.3 million lines of code (LoC), with LLVM being the largest program with over 2M LoC. Also, since many programs have more than one performance test, overall we run 129 performance tests.

All the benchmarks we used were from the Phoronix Test Suite,⁴ with the exception of Z3 for which we created new performance tests using files from the SMT library.⁵ We contributed back the Z3 benchmark to the Phoronix suite. We did not change the source code or the build systems of any program.

²<https://github.com/llvm/llvm-project/issues/65212>

³<https://github.com/llvm/llvm-project/pull/65596>

⁴<https://github.com/phoronix-test-suite/phoronix-test-suite/>

⁵<https://smt-lib.org/benchmarks.shtml>

Table 2. Benchmarks used in our experiments. The version number refers to the Phoronix benchmark version.

No	Benchmark	Category	Measurement Scale	kLoC
1	aom-av1-3.7.0	Video Encoding	frames/second	508
2	encode-flac-1.8.1	Audio Encoding	seconds	59
3	espeak-1.7.0	Speech Synthesizer	seconds	44
4	botan-1.6.0	Security	MB/second	148
5	john-the-ripper-1.8.0	Security	checks/second	315
6	openssl-3.1.0	Security	bytes/s	472
7	aircrack-ng-1.3.0	Security	seconds	496
8	build-llvm-1.5.0	Compiler	seconds	2,139
9	luajit-1.1.0	Compiler	Mflops	69
10	compress-pbzip2-1.6.0	Compression	seconds	6
11	compress-zstd-1.6.0	Compression	MB/second	85
12	draco-1.6.0	Texture Processing	seconds	50
13	graphics-magick-2.1.0	Image Processing	iterations/second	263
14	jpegxl-1.5.0	Image Processing	megapixels/second	106
15	fftw-1.2.0	HPC	Mflops	255
16	primesieve-1.9.0	HPC	seconds	9
17	mafft-1.6.2	HPC	seconds	496
18	simdjson-2.0.1	Parallel Processing	MB/s	74
19	tjbench-1.2.0	Parallel Processing	megapixels/second	57
20	rnnoise-1.0.2	Audio Processing	seconds	14
21	ngspice-1.0.0	Circuit Simulator	seconds	514
22	quantlib-1.2.0	Quantitative Finance	Mflops	395
23	z3-1.0.0	SMT Solver	seconds	496
24	sqlite-speedtest-1.0.1	Database	seconds	249

Most benchmarks in our suite measure the performance by the time spent in running the underlying application. Eleven benchmarks use time as the measurement scale. Next, Mflops, MB/s and megapixels/second are the second most popular categories with 4 benchmarks per category.

4 Setup

We used LLVM 16 (released on 17/March/2023) as the baseline and for implementing the new flags. Our code is available on a fork of LLVM’s repository.⁶

We used three servers for the experiments:

- 2x Intel Xeon CPU E5-2680 v2 @ 2.80GHz (IvyBridge), 64GB DDR3 RAM (@ 1600 MHz), running Debian 11
- 2x ARM64 Neoverse-N1 @ 3.00GHz (Ampere Altra) server, 1024GB DDR4 RAM (@ 3200 MHz), running Ubuntu 22.04
- 2x AMD EPYC 9J14 @ 4.00GHz (Zen) server, 2304GB DDR5 RAM (@ 4800 MHz), running Ubuntu 22.04

We used various techniques to reduce the noise generated by the operating system, other applications running on the system, and hardware interactions. We devised a benchmark environment where the relative standard deviation between the results of the same benchmark is at most 3%.

⁶<https://github.com/lucic71/llvm-project/tree/release/16.x>

Furthermore, we ignore results in the interval $[-2\%, +2\%]$ since real-world applications exhibit noise that must be ignored.

In summary, we applied the following techniques to reduce the environment noise:

- Disable unnecessary system services;
- Run benchmarks in single-threaded mode;
- Use the taskset and nice Linux utilities to reduce CPU and scheduler noise by pinning programs to a single CPU core and assigning maximum scheduler priority;
- Disable address space layout randomization (ASLR);
- Set the CPU to 80% of the total frequency to reduce thermal throttling and dynamic frequency scaling;
- Disable Turbo Boost, Hyper Threading, and similar technologies to reduce CPU noise;
- Use statistical methods provided by the Phoronix Test Suite (PTS) to keep the relative standard deviation between results under a threshold of 3%.

All benchmarks were ran in a freshly installed environment to avoid noise from unessential services and applications. Also, we modified PTS to run benchmarks in single-threaded mode since by default PTS runs benchmarks in multi-threaded mode.⁷

Clang enables exploitation of all UB in LLVM with one exception: by default, Clang does not enforce strict ranges for `enum` types. The reason is historical: many programs do bit-wise operations between values of `enum` types and often the result is not a value of the `enum`. This led to several miscompilations in practice, and thus enforcing `enum` ranges is currently opt-in through the flag `-fstring-enums`. Since our goal was to measure the impact of exploiting UB, we enable this flag for the baseline result, and measure the impact of disabling it through the flag `TR2`.

5 Results

In this section, we give an overview of the impact of exploiting UB in terms of performance and code size. We analyze selected cases in the next section.

5.1 Performance

Fig. 1 shows the overall performance impact of each flag across all benchmarks and CPU architectures. We include results for compilation with and without link-time optimizations (LTO).

Unsurprisingly, the performance impact (average and ranges) on AMD and Intel CPUs is much smaller than on ARM. These are wide out-of-order CPUs that can easily execute for free the few extra instructions due to fewer optimizations. We also note that disabling more UB has a cumulative effect, as can be seen in the ‘all’ column, which has the worst result in all cases.

More surprising are the results for LTO, where for ARM we get performance improvements on average, in some cases over 10%, while without LTO we get performance losses on average. Even for AMD and Intel, the overall performance ranges are a bit wider.

Fig. 2 shows the number of benchmarks that improve performance ($\geq 2\%$), and with moderate ($\geq 2\%$ and $< 5\%$) and severe ($\geq 5\%$) performance losses, for each flag and compilation mode. It is clear that LTO is very effective in recovering the losses observed with non-LTO builds, reducing both the total number of slowdowns and the number of severe performance losses. Again, we see that ARM benefits immensely from LTO, and that disabling exploitation of UB often improves the performance. The cumulative effect of disabling UB can also be observed in this figure, with the ‘all’ column having the most benchmarks that exhibit slowdowns.

⁷Our fork of PTS is available on GitHub: <https://github.com/lucic71/test-profiles/tree/ub>, <https://github.com/lucic71/phoronix-test-suite>.

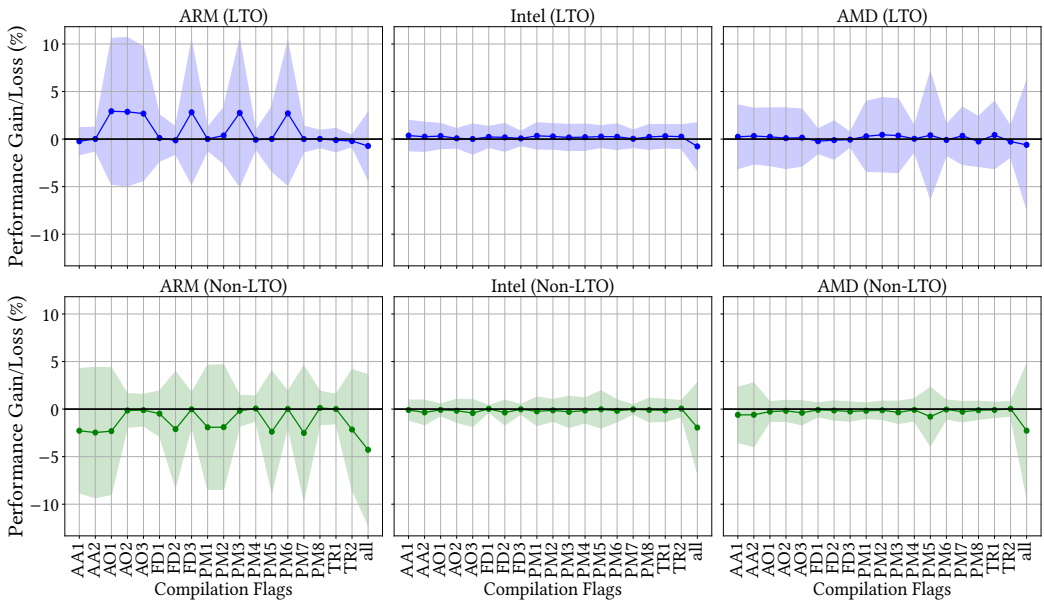


Fig. 1. Performance impact (%) for each flag across all benchmarks for all CPU architectures (ARM, Intel, AMD) and compilation modes (non-LTO and LTO). The lines indicate the average impact, and the solid boxes the min/max values. Higher values are better (meaning the performance improved by disabling some UB).

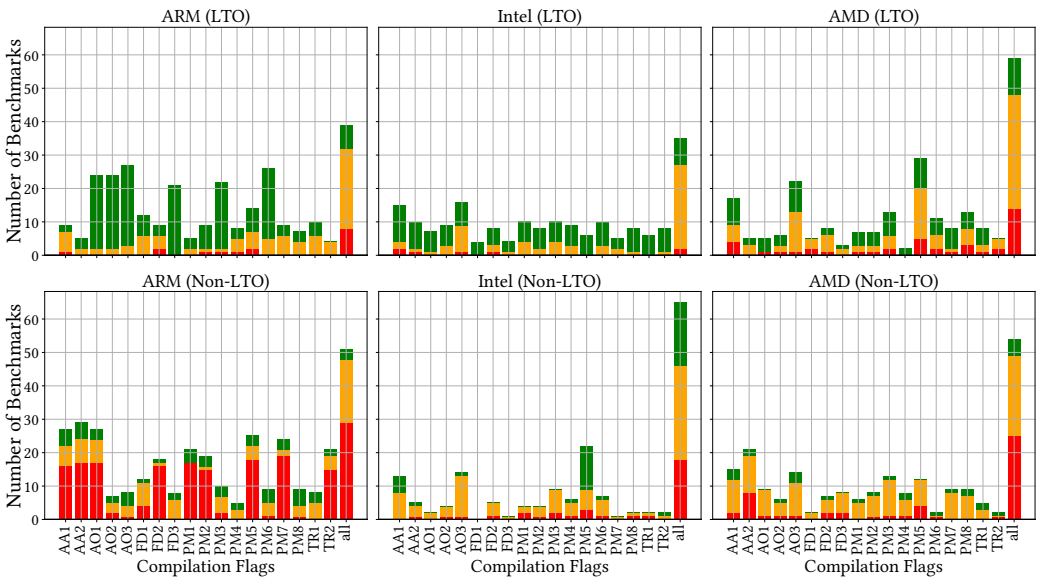


Fig. 2. Number of benchmarks that improve performance (green), and with moderate (orange) and severe (red) performance losses, for each CPU architecture and compilation mode.

Fig. 3 shows the performance results per program and flag category. We indicate the number of benchmark tests that exhibit performance variations greater than 2% (note that many programs have more than one test).

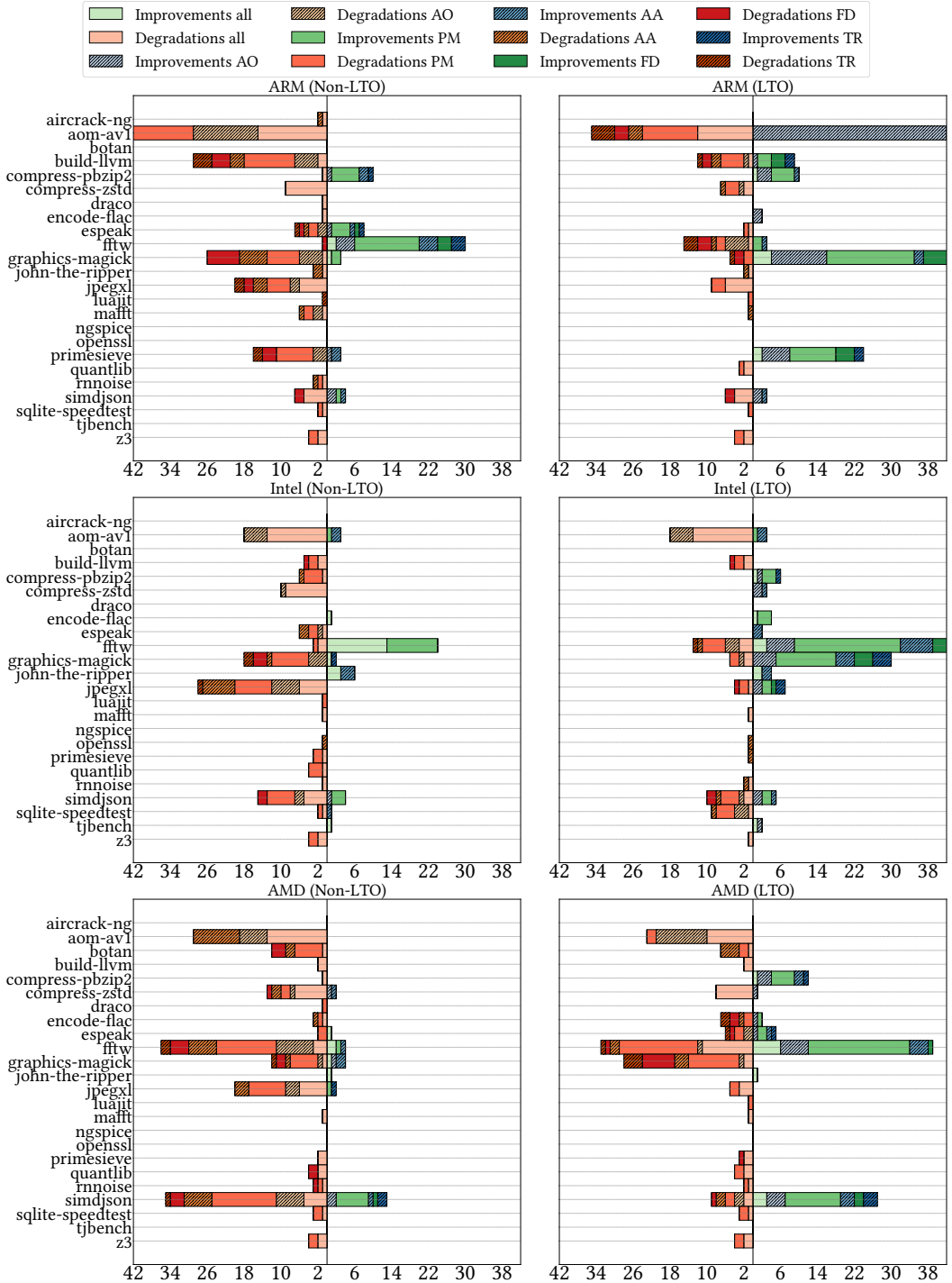


Fig. 3. Number of benchmark tests that degrade (left half of each plot) and improve (right half) the performance per program and per flag category. Results in the range $(-2, 2)$ are omitted, as they are considered noise.

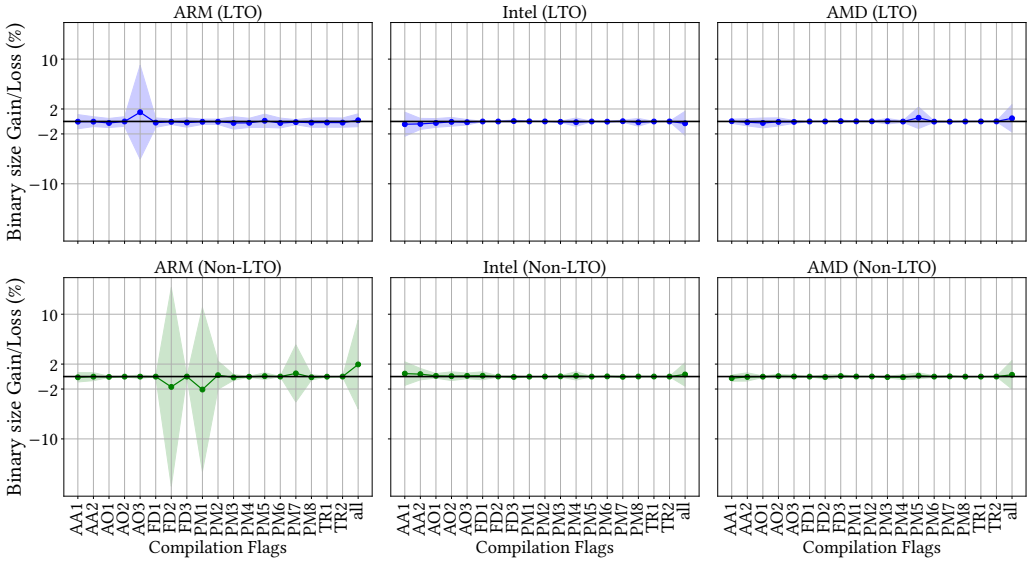


Fig. 4. Binary size impact (%) for each flag across all benchmarks for all CPU architectures (ARM, Intel, AMD) and compilation modes (non-LTO and LTO). The lines indicate the average impact, and the solid boxes the min/max values. Lower values are better (meaning the code size decreased by disabling some UB).

Some programs degrade in all compilation modes and CPU architectures. A notable one is `aom-av1`, where the AO flag group regresses the performance in all platforms, except on ARM LTO. For the `fftw` program, the PM flag group has the strange effect of improving half of the tests and degrading the other half.

Another interesting case is `simdjson`, where AMD is the only architecture with a large number of slowdowns. All flags cause slowdowns in at least one test of this program, but the cumulative effect is not observed: there are few slowdowns when all flags are enabled (no UB exploitation allowed).

5.2 Binary Size

Fig. 4 shows the overall impact on the binary size of each flag across all benchmarks and CPU architectures. The average impact is within our 2% target, with ARM being an exception with three flags showing larger changes: AO3 (LTO), and FD2 and PM1 (non-LTO). We note that the code size changes are often caused by differences in function inlining (due to an extra instruction causing the inlining heuristic crossing a threshold), and less because of the extra instructions due to fewer optimizations.

Fig. 5 shows a breakdown of the results per program and flag category. The impact on binary size is significantly smaller than on performance, with some programs not having any impact. The impact on x86 platforms is also smaller since the instruction set is larger and thus there are more opportunities for the compiler to be able to pack extra IR instructions into one assembly instruction, unlike ARM. Also, although AMD and Intel are both x86 architectures, we used different Linux distributions, explaining the different results.

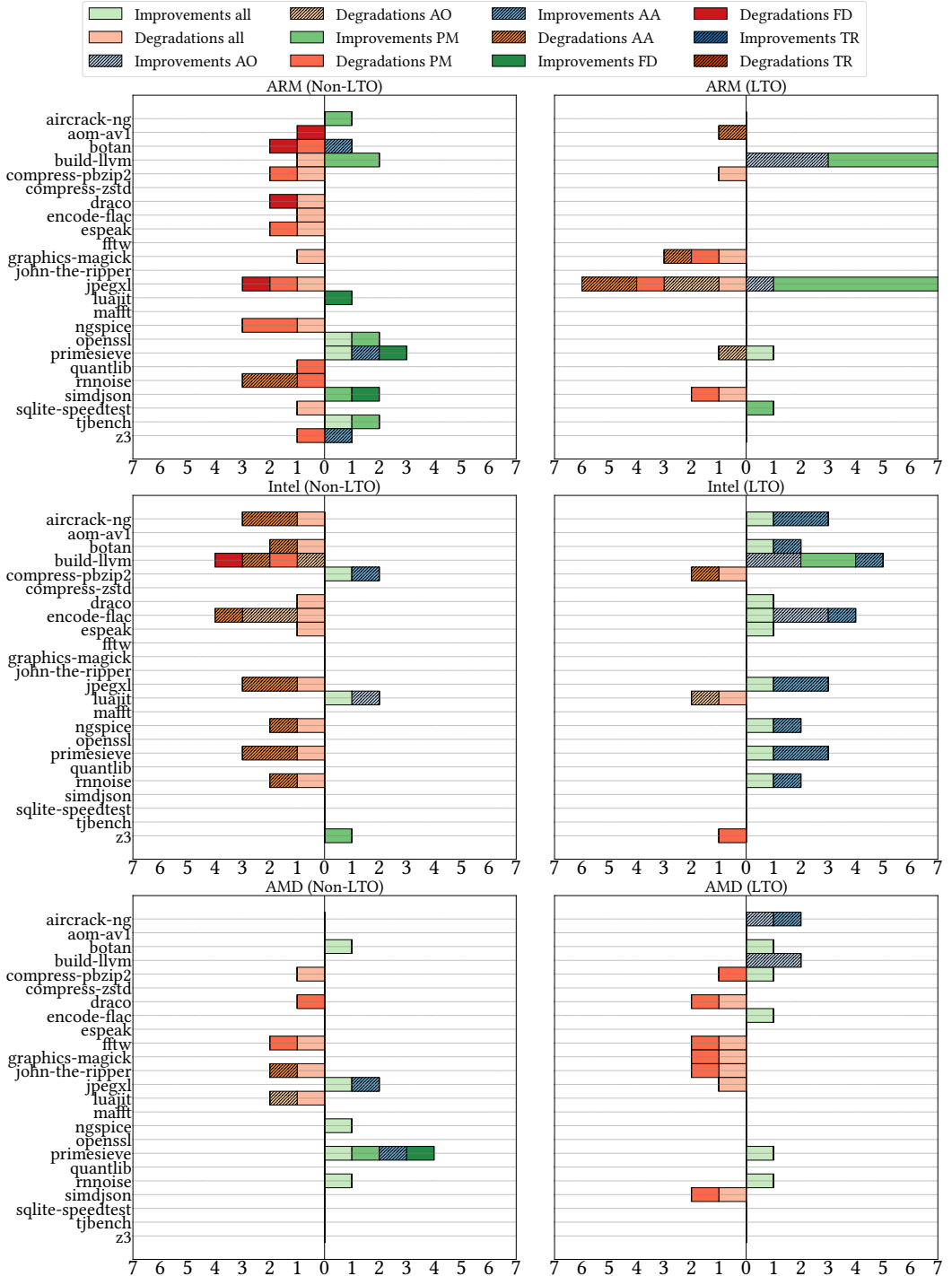


Fig. 5. Number of benchmark tests for which the binary size increases (left half of each plot) and decreases (right half) per program and flag category. We ignore size variations smaller than 1%.

5.3 Threats to Validity

Although we took extra care to ensure the validity of the results here presented, there are several threats to our study, namely:

- (1) **Choice of benchmarks:** compilers are tuned for certain benchmarks, and therefore the results obtained for one program do not necessarily carry over to another. We tried to select a large benchmark suite with programs of a wide range of domains. Nevertheless, our set of benchmarks does not cover all application domains.
- (2) **Missed some UB:** although we did an extensive search for UB in the LLVM IR’s manual and through the source code and consulted with some LLVM developers, it is likely that we missed some case since exploitation of UB is often subtle and hidden in complex preconditions. Therefore, the results shown for all UB disabled may be incorrect.
- (3) **Bugs in the implementation of the flags:** for the flags implemented in the frontend (the majority) this is unlikely as the implementation is simple and easy to test, but the flags implemented in the LLVM optimizer are much more complex and thus the probability of having a bug there is non-zero, albeit small.
- (4) **Bugs in LLVM:** we changed Clang to produce IR with some of the UB replaced with well-defined constructs, but then we rely on LLVM being correct to respect the semantics of the produced IR. As mentioned in Section 2.2, we found one bug in LLVM where it failed to respect one of our IR changes. We may have missed other similar bugs in LLVM that could impact the results.
- (5) **Limitations of Alive2:** Although we used Alive2 to detect missing UB flags and implementation bugs, Alive2 has key limitations that prevents it from giving us full confidence. In particular, Alive2 can only detect uses of “guardable UB” (will miss bugs around “non-guardable UB”), only supports intra-procedural optimizations (will miss bugs and UB exploitation done in inter-procedural optimizations), performs only bounded verification (potentially missing bugs involving complex loop optimizations), and it can timeout for large functions.
- (6) **Bugs in Alive2:** Our implementation of “guardable UB” may miss some cases, as it is difficult to test Alive2 for false negatives.
- (7) **Bugs in the benchmarks:** We caught several bugs in the benchmark programs, with one causing a significant performance drop with one of the flags (more details in the next section). We may have missed other bugs that impact the results.

6 Performance impact analysis

In this section, we analyze selected benchmarks that exhibit large performance variations when compiled with one or more UB-disabling flags. Moreover, we categorize benchmarks with respect to recoverability of the performance regressions according to three categories:

- Recoverable using link-time optimizations (LTO)
- Recoverable with small to moderate changes to the compiler
- Theoretically impossible to fix or requires a very complex algorithm

Table 3 summarizes the results. Out of the presented 12 cases, only 2 have performance drops that are not recoverable using standard compiler optimization algorithms. The remaining cases are either trivially recoverable by compiling the program with LTO or by doing small changes to LLVM.

There are five main root causes for the performance variations: lack of precision of the pointer/alias analysis (which is a known deficiency of LLVM), limitations in the heuristic for alignment of loops, limitations in the loop vectorizer algorithm in handling some loops with unknown trip counts (partially fixed already in LLVM 19), limitations in the register allocator’s heuristics, and the inlining

Table 3. Recoverability and root causes for selected performance variations in benchmarks.

Recoverability	Root Cause	Flag	Benchmark	Impact
Recoverable with LTO	Pointer analysis	PM1	simdjson	-13%
	Pointer analysis	PM2	simdjson	-13%
Recoverable with moderate work	Loop vectorizer	AA1	jpegxl	-4%
	Alias analysis	AA2	espeak	-4.2%
	Inliner	AO2	zstd	-2.1%
	Pointer analysis	PM3	pbzip2	-3.2%
	Loop misalignment	PM4	jpegxl	-2.2%
Unrecoverable	Pointer analysis	FD2	simdjson	-4%
	Pointer analysis	PM6	sqlite	-3%
Bug in the program	Use of uninitialized data	PM8	john-the-ripper	-1%
Improvement	Loop misalignment	AO1	jpegxl	+7%
	Register allocator	PM5	fftw	+3%

heuristic not understanding a free pattern. Next, we explore each category of performance changes and give a detailed root cause analysis and methods to recover the lost performance whenever possible.

6.1 Recoverable with Link-Time Optimizations (LTO)

Both `-fdrop-align-attr` (PM1) and `-fdrop-deref-attr` (PM2) degrade the performance of the `simdjson` benchmark by 13%. These flags disable the emission of, respectively, alignment and dereferenciability information for the “this” pointer of C++ methods. By default, Clang emits IR that assumes that methods are called on objects that are fully dereferenceable. This allows, for example, the compiler to execute loads from the object speculatively.

For this benchmark, dropping either piece of information prevents the loop invariant code motion (LICM) optimization from hoisting a load out of the loop. A simplified version of the code is shown in Fig. 6. The blue line indicates the load instruction that is hoisted when not using either of the flags.

To move the load instruction out of the loop, the compiler needs to prove one of the following conditions: either the loop is guaranteed to execute at least once, or the load cannot trigger UB in any circumstance and yields the same result in all iterations. If the loop executes at least once, then executing the load earlier would be sound even if it triggers UB since UB in LLVM has “time-travel” semantics. Since the loop guard is a function argument, the compiler cannot prove whether the loop executes or not (using intra-procedural reasoning).

For a load instruction to be well-defined, the following conditions must hold: (1) the input pointer must point to an object in memory that is live and larger than the access size, and (2) the input pointer must be at least as aligned as the alignment argument.

In the code above, both conditions are met: the `dereferenceable(48)` attribute on the `%this` pointer guarantees condition (1) since the load does an 8-byte access (loads a pointer from memory) which is less than the 48 bytes pointed to by `%this`. The `align 8` attribute guarantees condition (2) since the load requires the pointer to be at least 8-byte aligned and the attribute guarantees that.

Now it is obvious that removing either of the parameter attributes blocks LICM from hoisting the load. However, it is possible to recover this optimization in two ways. First, some CPU architectures

```

define i1 @run(ptr align 8 dereferenceable(48) %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

```

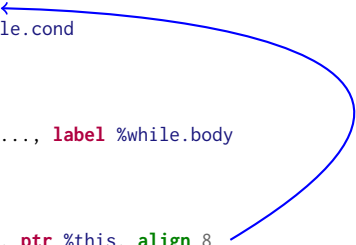


Fig. 6. Simplified code of a function of `simdjson`. The blue arrow indicates the load that is hoisted when not using the `PM1` or `PM2` flags.

(notably `x86`) can execute unaligned memory operations. Therefore, in order to work around the loss of alignment information on the `%this` pointer, we could change LICM to still hoist the load for CPU architectures that support unaligned loads, but dropping the alignment requirements of the load instruction (i.e., use `'align 1'` on the load).

A simpler alternative is to compile the program with LTO. We confirmed that LLVM's inter-procedural analyses can propagate both alignment and dereferenceability information for this function, which allows the LTO build to recover the performance loss.

6.2 Recoverable with Moderate Changes to LLVM

We now discuss benchmarks whose performance regressions could be recovered with easy to moderate changes to LLVM.

6.2.1 `-fdrop-inbounds-from-gep -mllvm -disable-oob-analysis (AA1)`. These flags cause a 4% degradation on `jpegxl` because the missing assumption that pointer arithmetic cannot overflow prevents the vectorization of a hot loop in the function `jxl::FindTextLikePatches`.

The missing `inbounds` attribute affects vectorization of loops where the induction variable is a pointer, e.g.:

```

preheader:
  ...
  br label %loop

loop:
  %ptr = phi ptr [%ptr2, %loop], [%init, %preheader]
  ...
  %ptr2 = getelementptr inbounds i8, ptr %ptr, i64 %inc
  %c = icmp ne ptr %ptr2, %end
  br i1 %c, label %loop, label %exit

exit:
  ...

```

When both `%init` and `%end` are a multiple of the increment (`%inc`), the loop is guaranteed to terminate. We get the sequence of `%ptr = %init, %init + %inc, %init + 2 × %inc, . . . , %end`. Otherwise, it may happen that the sequence wraps around and continues forever.

For disequalities (i.e., `icmp ne`), the `inbounds` attribute guarantees that the loop is terminating because since objects cannot wrap around the address space and since the result of the pointer arithmetic operation must be in-bounds of some object, the induction variable must reach `%end`, otherwise it triggers UB.

Loop vectorization algorithms generate vectorized loops that iterate, e.g., a quarter of the iterations that the original loops did. Therefore, computing the loop trip count (even if in a symbolic form) is crucial for these algorithms. As we have seen, in some cases we cannot statically decide if a loop terminates without the help of UB reasoning. An alternative is to push some of the reasoning to run time. In fact, LLVM 19 can already vectorize some loops similar to the one above by generating extra code to check that the start/end pointers are multiples of the increment.

6.2.2 -disable-object-based-analysis (AA2). This flag causes a 4.2% degradation on `espeak` because the precision loss in the alias analysis prevents vectorization of a hot loop.

Concretely, in order to vectorize the loop below, the compiler needs to prove that the array accesses on the right side of the assignment do not alias with the array access on the left. This is because the goal is to perform multiple loop iterations at once and if there is the possibility that, say, `harm_inc[0]` and `htab[1]` point to the same location, we could not read `htab[0..3]` at once and do four iterations of the loop in parallel because of the inter-iteration dependency.

```
static int harm_inc[N_LOWHARM];
static int hspect[2][MAX_HARMONIC];
static int *harmspect;

int PeaksToHarmspect(wavegen_peaks_t *peaks, int pitch, int *htab, int control) {
    for (h = 1; h < N_LOWHARM; h++)
        harm_inc[h] = (htab[h] - harmspect[h]) >> 3;
}

int Wavegen(...) {
    harmspect = hspect[0];
    maxh2 = PeaksToHarmspect(peaks, wdata.pitch<<4, hspect[0], 0);
}
```

We first note that `PeaksToHarmspect` gets inlined in the `Wavegen` function, and thus it becomes obvious that `htab` and `harmspect` point to a global variable. Without the flag, LLVM proves that `htab` and `harm_inc` do not alias using object-based reasoning: since pointer arithmetic cannot overflow (the program would trigger UB otherwise), it determines that `'harm_inc + h'` and `'hspect[0] + h'` cannot alias as these pointers are based on different objects.

To prove non-aliasing without resorting to UB, the alias analysis would need to take into consideration the loop bounds to prove that the accesses do not overflow the arrays. We tried LLVM's SCEV AA algorithm since it implements such kind of reasoning, but unfortunately it was not able to prove no-aliasing. We believe that extending SCEV AA to support the case above can be done with moderate amount of work.

6.2.3 -fconstrain-shift-value (AO2). This flag caused a 2.1% degradation on `zstd` because the extra 'and' instruction added before each shift prevents one function (`ZSTD_getOffsetInfo`) from being inlined. However, this extra instruction is free on x86 because the shift assembly instruction implements this mask+shift semantics, so there is no change in the assembly emitted. The fix is simple: LLVM's inlining heuristic should learn about this pattern to not count the 'and' instruction.

6.2.4 *-fno-delete-null-pointer-checks (PM3)*. This flag causes a 3.2% degradation on pbzip2 because the SimplifyCFG optimization fails to merge two basic blocks. In the `StreamScanner::getNextStream` function, there is the following check (where all variables are object fields):

```
if (_InBuffCurrent == _InBuffEnd && _eof)
    return ...;
```

C++ evaluates the `&&` operator using short-circuiting, i.e., it only evaluates `_eof` if the comparison evaluates to true. This creates two basic blocks, but LLVM without the flag merges them by speculatively evaluating both sides of the `&&` operator.

To merge the two basic blocks without breaking the short-circuiting semantics, the compiler must prove that it is safe to execute the right-hand side (RHS) speculatively, i.e., the RHS cannot trigger UB nor have side-effects.

The PM3 flag changes the attribute of the “this” pointer parameter from `dereferenceable`(204) to `dereferenceable_or_null`(204). On the surface, this means that the compiler can no longer prove that accessing the `_eof` field is safe since the “this” pointer may be null. However, since we had already accessed the other two fields, accessing `_eof` is safe. The culprit is that LLVM’s pointer analysis needs to learn about this pattern, which amounts to modest amount of implementation work.

6.2.5 *-fdrop-noalias-restrict-attr (PM4)*. This flag causes a 2.2% degradation on jpegxl due to a loop becoming misaligned. With the flag, the size of function `jpegxl::N_SSE4::L2DiffAsymmetric` increases by 16 bytes due to loss of precision in the alias analysis. Although this function is not called by the benchmark, this size increase has a ripple effect and changes the alignment of a loop in another function placed below in the binary.

This performance degradation is recoverable with a better loop alignment heuristic.

6.3 Unrecoverable

6.3.1 *-fdrop-ub-builtins (FD2)*. This flag causes a 4% degradation on simdjson. When this flag is enabled, calls to unreachable builtins (UB) are converted into traps (well-defined exits). Unreachable builtins are used only for optimization and do not generate assembly instructions, while traps do.

The program uses unreachable builtins as follows:

```
#define SIMDJSON_ASSUME(COND) do { if (!(COND)) __builtin_unreachable(); } while (0)
```

LLVM recognizes this pattern and transforms it into a call to `@llvm.assume`, which is used only by optimizations; the code for the condition is not emitted. However, when the unreachable is converted into a trap, LLVM translates the code into an if statement, which gets emitted into assembly and checked at run time.

Out of the 30 uses of the macro, 13 traps are deleted as LLVM can statically prove that the condition holds. The remaining 17 cases are evaluated at run time and we argue that no standard compiler algorithm could remove those. Many of these cases originate in the following function:

```
bool value_iterator::find_field_unordered_raw(const std::string_view key) {
    // validate input string
    while (has_value) {
        if ((error = field_value()) ) { abandon(); return error; }
        ....
    }

    // consume the string
    while (true) {
        error = field_value(); SIMDJSON_ASSUME(!error);
    }
    ...
}
```

The function traverses the input string twice: first it checks if the string is well-formed, and then it consumes the string assuming it is well-formed. Proving that the assumptions of the second loop hold statically would require proving that the first loop only lets well-formed strings pass through, which is well beyond the reasoning power of compiler algorithms.

6.3.2 *-fno-use-default-alignment (PM6)*. This flag causes a 3% degradation on `sqlite` because the loss of alignment information in atomic operations is very expensive. The `sqlite3VdbeExec` function has the following code:

```
if( AtomicLoad(&db->u1.isInterrupted) ) goto abort_due_to_interrupt;
```

The size of the `isInterrupted` field is 4 bytes. If the compiler knows the field is 4-byte aligned (or more), it can compile the load into a single assembly instruction (on x86). But without knowing the alignment, the compiler inserts a call to a run-time library function (`__atomic_load`), since it may need to acquire a lock. Unaligned atomic operations are thus very expensive.

Inferring the alignment of a pointer is trivial when we have “line-of-sight” to the allocation site (i.e., we have a simple data-flow path until a `malloc`, for instance). In the case above, the field is accessed through multiple pointer indirections, which is beyond the reasoning power of compiler algorithms.

6.4 Bug in the source code

The flag `-zero-uninit-loads (PM8)` degrades the performance of `john-the-ripper` by 1% because the program accesses uninitialized data. The flag changes that access to load zero instead, incurring in the cost of materializing the constant in terms of both run time and binary size.

The relevant code is as follows:

```
void scan_central_index(const char *fname) {
    zip_context ctx;
    ...
    if (this_disk != 0 || cd_start_disk != 0) {
        // ctx.archive.zip64 not initialized
    } else {
        ctx.archive.zip64 = zip64;
    }
    ...
    if (ctx.archive.zip64) {
        ...
    }
}
```

On the true branch, the variable is left uninitialized and thus the program triggers UB when that path is followed. The increase in binary size causes the performance degradation due to a ripple effect in misaligning a loop. We have fixed the bug in the program and upstreamed the bug fix.⁸

6.5 Improvements

6.5.1 *-fcheck-div-rem-overflow (AO1)*. This flag causes a 7% improvement on `jpegxl` due to a hot loop becoming better aligned. Enabling this flag increases the binary size because of the extra checks around division/remainder operations, and that causes a ripple effect of increasing the alignment of a hot loop.

The following function reproduces the effect of loop alignment on performance. On the left, a function in C, and on the right the corresponding x86 assembly. Increasing the alignment from 16 to 32 bytes improves the performance of the function significantly on some microarchitectures. It

⁸<https://github.com/openwall/john/pull/5466>

halves the `idq.all_dsb_cycles_any_uops` performance counter, which indicates that the CPU executes fewer cycles.

```

                                .align 16
                                loop:
int loop(int *array, int length, int value) {
                                test  %rsi,%rsi
                                jle  end
                                lea  -0x1(%rsi),%rax
                                cmp  %edx,-0x4(%rdi,%rsi,4)
                                mov  %rax,%rsi
                                jne  loop
                                mov  $0x1,%eax
                                ret
                                end:
                                mov  $0x0,%eax
                                ret
}

```

LLVM aligns small loops with 16 to 31 bytes to a 16-byte boundary. The extra code inserted by the flag had the effect of changing the alignment of one small loop (18 bytes) to 32 bytes. For certain microarchitectures, the increased alignment improves the performance of the instruction decoding mechanism of the CPU. This is because the window of the uops cache becomes aligned with the loop body.

We reproduced this performance improvement on Intel Sandy Bridge, but not on IvyBridge nor on Westmere. This issue has been reported in LLVM’s mailing list before.⁹ LLVM’s loop alignment heuristic could be improved to increase alignment to a 32-byte boundary for the relevant microarchitectures (possibly conditional on profiling data to avoid large binary size increases). Currently, the solution is to specify one of the two compiler flags by hand: `-branches-within-32B-boundaries` or `-x86-experimental-pref-innermost-loop-alignment`.

6.5.2 *-fno-strict-aliasing (PM5)*. This flag causes a 3% improvement on `fftw` due to a limitation in the register allocator’s heuristics. Disabling type-based alias analysis prevents the common sub-expression elimination (CSE) optimization to remove some redundant instructions. Although this leads to an IR with 8% more instructions, the smaller liveness intervals for key registers makes the register allocator produce better code (the final binary is 11% smaller).

In particular, for the `q1_8` function, we observe that the number of `add/lea` instructions increases from 137 to 169 (+32), but the total number of moves decreases from 1,296 to 1,013 (-283), far outstripping the small increase in arithmetic operations.

In principle, the register allocator should be able to tradeoff register spills for rematerializations (repeat arithmetic operations), but since register allocation is an NP-hard problem, it is not surprising that the register allocator can sometimes fail to produce better code.

LLVM has 3 register allocators: greedy (default for `-O2`), fast (default for `-O0`), and PBQP [17]. We tried all the three allocators, and could only reproduce the problem with the greedy allocator.

7 Related work

We are not aware of any other systematic study on the performance benefits of exploiting UB for optimization. We survey related work that contributed to the understanding of UB and that made this work possible, including what UB exactly means, how it can be detected, etc.

Semantics of intermediate representations (IRs). Lee et al. [27, 28] proposed a new semantics for LLVM’s IR to allow LLVM to perform key optimizations in the presence of UB. Previously, LLVM was doing optimizations that were not correct and it had optimizations that were incompatible

⁹<https://lists.llvm.org/pipermail/llvm-dev/2021-January/148177.html>

between each other. Chakraborty and Vafeiadis [8] formalized the semantics of some parts of the LLVM IR related with concurrency, including cases involving UB. Parts of the LLVM IR semantics, including some UB aspects of it, have been formalized in Coq [22, 64] and in K [29]. Dahiya and Bansal [11] formalized parts of the GCC’s IR related with UB. Shen [47] gives a non-exhaustive list of UB exploited by GCC.

Semantics of C/C++. There have been multiple attempts to formalize the UB in the C [18, 24, 37] and C++ [61] standards. There are also proposals to remove some UB from C++, including automatically initializing local variables [5], allowing certain infinite loops [6], and memory safety [41], as well as downgrading certain UB to “erroneous behavior” [25].

Dialects of C/C++. There is a proposal for the creation of C++ profiles to let the users pick the flavor of safety they want [19]. C/C++ compilers already support multiple flags that change the language semantics, e.g., `-fwrapv`, which changes the semantics of signed integer overflows.

Detection of UB. Several tools have been created to help developers find sources of UB and to protect applications from potential security vulnerabilities created by UB. Many tools focus on finding memory safety issues at run time [3, 7, 40, 44, 50, 51], as well as race conditions [45], integer overflows [12], and violations of the strict aliasing rules [16]. Most tools degrade performance significantly and thus cannot be used in production. Some tools opt to run the checks at random only in order to reduce the overhead at the expense of missing bugs [46]. There are also static UB checkers [32]. Other tools focus on cheaper properties than full memory safety, such as control-flow integrity (CFI), where indirect calls/jumps are forced to go to a “safe” location [1, 54].

Baev et al. [4] and Dunaev et al. [15] studied the performance impact of several compiler hardening flags of GCC and Clang, respectively. They report an almost 3× slowdown in some benchmarks when all flags are enabled.

Compiler testing. Fuzzers are now widely used to find bugs in compilers, and they go to great lengths to generate programs without UB [26, 52, 63]. One reason being that they usually compare the output of different compilers and compilers differ in the treatment of UB. Compiler verification tools have also added support for UB in the past decade [34, 35]. LookUB [21] is a tool to find optimizations that remove run-time safety checks. UBFUzz [31] finds bugs in run-time safety checks by generating programs with UB,

Security risks of UB. Taking advantage of UB to perform optimizations can break the security and correctness properties of applications [14]. Wang et al. [55] provide examples of code in PostgreSQL, the Linux kernel, and FreeBSD’s libc that are deleted when compiled with a modern compiler. Later, they developed STACK [57, 58] to detect such vulnerable code automatically. From over 8,000 analyzed Debian packages, 40% of them contained vulnerable code. In addition, Xu et al. [60] collected and analyzed more than 4,000 bug reports related to UB optimizations, underlining the increased negative security impact of UB optimizations in the wild. Further studies [23, 30, 36, 48, 56, 59] focused on understanding the security impact of particular optimizations, such as integer overflow, dead code elimination, or uninitialized reads.

Performance studies. Mytkowicz et al. [39] showed that the execution environment and ripple effects can sometimes explain performance differences in programs compiled with different flags. In our study we encountered several cases of ripple effects degrading the performance that were unrelated with our changes (e.g., adding an instruction to a function made another function slower due to code alignment).

Theodoridis and Su [53] showed that compilers are not monotonic with respect to UB: sometimes adding more UB to a program makes the compiler generate worse code, while in theory that should never happen. Our study confirms this result as well.

Some studies with HPC software indicate an upper bound of 20% in the performance improvements available if the perfect UB information is added [13, 20]. Other studies have focused on loop optimizations such as vectorization [2, 49]. Hydra [38], Minotaur [33], and Souper [43] are superoptimizers that produce optimizations that exploit UB in the LLVM IR.

8 Conclusion

We presented the first comprehensive study on the performance impact of exploiting undefined behavior (UB) in C and C++ programs. First, we cataloged 18 individual UB aspects exploited by LLVM, a compiler that is widely known for its extensive use of UB. We then implemented flags in the compiler that disable exploitation of each UB aspect by strengthening the semantics (e.g., define division by zero as a well-defined trap instead of UB).

The results show that, in the cases we evaluated, the performance gains from exploiting UB are minimal. Furthermore, in the cases where performance regresses, it can often be recovered by either small to moderate changes to the compiler or by using link-time optimizations.

We note that our study makes no attempt to quantify the potential security benefits or improvements in developer productivity from disabling the identified UB classes. Our primary goal was to first understand the performance impact, leaving further exploration for future work. We hope this study serves as a starting point for discussions within language committees and related communities.

Acknowledgments

This work was supported in part by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), and a cloud credits gift from Oracle.

References

- [1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (nov 2009). <https://doi.org/10.1145/1609956.1609960>
- [2] Neil Adit and Adrian Sampson. 2022. Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V. *IEEE Micro* 42, 5 (2022), 41–48. <https://doi.org/10.1109/MM.2022.3184867>
- [3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security*. https://www.usenix.org/legacy/event/sec09/tech/full_papers/akritidis.pdf
- [4] R. V. Baev, L. V. Skvortsov, E. A. Kudryashov, R.A. Buchatskiy, and R. A. Zhuykov. 2021. Prevention of vulnerabilities arising from optimization of code with Undefined Behavior. *ISP RAS* 33 (2021), Issue 4. [https://doi.org/10.15514/ISPRAS-2021-33\(4\)-14](https://doi.org/10.15514/ISPRAS-2021-33(4)-14)
- [5] JF Bastien. 2023. C++ Proposal P2723R1: Zero-initialize objects of automatic storage duration. <http://wg21.link/P2723>
- [6] JF Bastien. 2024. C++ Proposal P2809R3: Trivial infinite loops are not Undefined Behavior. <http://wg21.link/P2809r3>
- [7] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *CGO*. <https://doi.org/10.1109/CGO.2011.5764689>
- [8] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *CGO*. <https://doi.org/10.1109/CGO.2017.7863732>
- [9] ANSI Technical Committee and ISO/IEC JTC 1 Working Group. 1989. Rationale for International Standard - Programming Language - C. <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n850.pdf>
- [10] CVE. 2009. CVE-2009-1897: Vulnerability in the linux kernel involving a NULL pointer dereference. <https://www.cve.org/CVERecord?id=CVE-2009-1897>
- [11] Manjeet Dahiya and Sorav Bansal. 2017. Modeling Undefined Behaviour Semantics for Checking Equivalence Across Compiler Optimizations. In *HVC*. https://doi.org/10.1007/978-3-319-70389-3_2

- [12] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding Integer Overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 2 (dec 2015). <https://doi.org/10.1145/2743019>
- [13] Johannes Doerfert, Brian Homerding, and Hal Finkel. 2019. Performance exploration through optimistic static program annotations. In *ISC High Performance 2019*. https://doi.org/10.1007/978-3-030-20656-7_13
- [14] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *SPW*. <https://doi.org/10.1109/SPW.2015.33>
- [15] P. D. Dunaev, A. A. Sinkevich, A. M. Granat, I. A. Batraeva, S. V. Mironov, and N. Yu. Shugaley. 2024. Developing a clang-based safe compiler. *ISP RAS* 36 (2024). Issue 4. [https://doi.org/10.15514/ISPRAS-2024-36\(4\)-3](https://doi.org/10.15514/ISPRAS-2024-36(4)-3)
- [16] Hal Finkel. 2017. The Type Sanitizer: Free Yourself from -fno-strict-aliasing. <https://llvm.org/devmtg/2017-10/slides/Finkel-The%20Type%20Sanitizer.pdf>
- [17] Lang Hames and Bernhard Scholz. 2006. Nearly optimal register allocation with PBQP. In *JMLC*. https://doi.org/10.1007/11860990_21
- [18] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *PLDI*. <https://doi.org/10.1145/2737924.2737979>
- [19] H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, and M. Wong. 2023. C++ document P2759R1: DG Opinion on Safety for ISO C++. <https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2023/p2759r1.pdf>
- [20] Jan Hueckelheim and Johannes Doerfert. 2023. ORAQL — Optimistic Responses to Alias Queries in LLVM. In *ICPP*. <https://doi.org/10.1145/3605573.3605644>
- [21] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. 2023. Don't Look UB: Exposing Sanitizer-Eliding Compiler Optimizations. *Proc. ACM Program. Lang.* 7, PLDI, Article 143 (jun 2023). <https://doi.org/10.1145/3591257>
- [22] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: verified credible compilation for LLVM. In *PLDI*. <https://doi.org/10.1145/3192366.3192377>
- [23] Andreas D. Kellas, Alan Cao, Peter Goodman, and Junfeng Yang. 2023. Divergent Representations: When Compiler Optimizations Enable Exploitation. In *SPW*. <https://doi.org/10.1109/SPW59333.2023.00035>
- [24] Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *CPP*. <https://doi.org/10.1145/2676724.2693571>
- [25] Thomas Köppe. 2023. Correct and incorrect code, and erroneous behaviour. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2795r0.html>
- [26] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*. <https://doi.org/10.1145/2814270.2814319>
- [27] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (oct 2018). <https://doi.org/10.1145/3276495>
- [28] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *PLDI*. <https://doi.org/10.1145/3062341.3062343>
- [29] Liyi Li and Elsa L. Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *ECOOP*. <https://doi.org/10.4230/LIPIcs.ECOOP.2020.7>
- [30] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *ASPLOS*. <https://doi.org/10.1145/3582016.3582053>
- [31] Shaohua Li and Zhendong Su. 2024. UBFuzz: Finding Bugs in Sanitizer Implementations. In *ASPLOS*. <https://doi.org/10.1145/3617232.3624874>
- [32] Changming Liu, Yaohui Chen, and Long Lu. 2021. KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-5_24461_paper.pdf
- [33] Zhengyang Liu, Stefan Mada, and John Regehr. 2024. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 326 (Oct. 2024). <https://doi.org/10.1145/3689766>
- [34] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *PLDI*. <https://doi.org/10.1145/3453483.3454030>
- [35] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *PLDI*. <https://doi.org/10.1145/2737924.2737965>
- [36] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *CCS*. <https://doi.org/10.1145/2976749.2978366>
- [37] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *PLDI*. <https://doi.org/10.1145/2908080.2908081>

- [38] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 120 (April 2024). <https://doi.org/10.1145/3649837>
- [39] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *ASPLOS*. <https://doi.org/10.1145/1508244.1508275>
- [40] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*. <https://doi.org/10.1145/1250734.1250746>
- [41] Thomas Neumann. 2023. C++ Proposal R2771R1: Towards memory safety in C++. <https://wg21.link/R2771/1>
- [42] Alexander Potapenko. 2020. Fighting Uninitialized Memory in the Kernel. In *Clang-Built Linux Meetup*. https://clangbuiltlinux.github.io/CBL-meetup-2020-slides/glider/Fighting_uninitialized_memory_%40_CBL_Meetup_2020.pdf
- [43] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2018. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422
- [44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*. <https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf>
- [45] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *WBLA*. <https://doi.org/10.1145/1791194.1791203>
- [46] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2024. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. In *ICSE-SEIP*. <https://doi.org/10.1145/3639477.3640328>
- [47] Zefan Shen. 2022. The Impact of Undefined Behavior on Compiler Optimization. In *ESSE*. <https://doi.org/10.1145/3501774.3501781>
- [48] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *EuroS&P*. <https://doi.org/10.1109/EuroSP.2018.00009>
- [49] Sergi Siso, Wes Armour, and Jeyarajan Thiyagalingam. 2019. Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information. *ACM Trans. Archit. Code Optim.* 16, 4, Article 40 (oct 2019). <https://doi.org/10.1145/3356842>
- [50] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *SP*. <https://doi.org/10.1109/SP.2019.00010>
- [51] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *CGO*. <https://doi.org/10.1109/CGO.2015.7054186>
- [52] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *OOPSLA*. <https://doi.org/10.1145/2983990.2984038>
- [53] Theodoros Theodoridis and Zhendong Su. 2024. Refined Input, Degraded Output: The Counterintuitive World of Compiler Behavior. *Proc. ACM Program. Lang.* 8, PLDI, Article 174 (jun 2024). <https://doi.org/10.1145/3656404>
- [54] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security*. <https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf>
- [55] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *APSYS*. <https://doi.org/10.1145/2349896.2349905>
- [56] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In *OSDI*. <https://www.usenix.org/system/files/conference/osdi12/osdi12-final-88.pdf>
- [57] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*. <https://doi.org/10.1145/2517349.2522728>
- [58] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2015. A Differential Approach to Undefined Behavior Detection. *ACM Trans. Comput. Syst.* 33, 1, Article 1 (mar 2015). <https://doi.org/10.1145/2699678>
- [59] Zekai Wu, Wei Liu, Mingyue Liang, and Kai Song. 2020. Finding Bugs Compiler Knows but Doesn't Tell You: Dissecting Undefined Behavior Optimizations in LLVM. *BlackHat Europe* (2020). <https://i.blackhat.com/eu-20/Wednesday/eu-20-Wu-Finding-Bugs-Compiler-Knows-But-Does-Not-Tell-You-Dissecting-Undefined-Behavior-Optimizations-In-LLVM.pdf>
- [60] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *USENIX Security*. <https://www.usenix.org/system/files/usenixsecurity23-xu-jianhao.pdf>
- [61] Shafik Yaghmour. 2019. C++ Proposal P1705R1: Enumerating Core Undefined Behavior. <https://wg21.link/P1705>
- [62] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. 2011. Why nothing matters: the impact of zeroing. In *OOPSLA*. <https://doi.org/10.1145/2048066.2048092>
- [63] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. <https://doi.org/10.1145/1993498.1993532>

- [64] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021). <https://doi.org/10.1145/3473572>

A Disabling Undefined Behavior: Examples for Each Flag

Here we give an example for each flag we used to disable exploitation of undefined behavior (UB). For each flag, we give an example in C and the corresponding LLVM intermediate representation (IR) generated by Clang without (left) and with (right) the flag. Note that we simplified the IR to remove irrelevant details.

A.1 AO1: -fcheck-div-rem-overflow

```
unsigned f(unsigned a, unsigned b) {
    return a / b;
}
```

```
define i32 @f(i32 %a, i32 %b) {
    %r = udiv i32 %a, %b
    ret i32 %r
}
```

```
define i32 @f(i32 %a, i32 %b) {
    ; with signed division (sdiv) there is
    ; an extra check for overflow (INT_MIN/-1)
    %cmp = icmp eq i32 %b, 0
    br i1 %cmp, label %z, label %nz

z:
    ret i32 0

nz:
    ; never UB
    %r = udiv i32 %a, %b
    ret i32 %r
}
```

A.2 AO2: -fconstraint-shift-value

```
unsigned f(unsigned a, unsigned b) {
    return a << b;
}
```

```
define i32 @f(i32 %a, i32 %b) {
    %r = shl i32 %a, %b
    ret i32 %r
}
```

```
define i32 @f(i32 %a, i32 %b) {
    %m = and i32 %b, 31
    %r = shl i32 %a, %m
    ret i32 %r
}
```

A.3 AO3: -fwrapv

```
int f(int a, int b) {
    return a + b;
}
```

```
define i32 @f(i32 %a, i32 %b) {
    %r = add nsw i32 %a, %b
    ret i32 %r
}
```

```
define i32 @f(i32 %a, i32 %b) {
    %r = add i32 %a, %b
    ret i32 %r
}
```

A.4 TR1: -fno-constrain-bool-value

```
bool f(bool *b) {
    return *b > 1;
}
```

```
define i1 @f(ptr %b) {
  %val = load i8, ptr %b, !range !1
  %r = icmp sgt i8 %val, 1 ; always false
  ret i1 %r
}
```

!1 = !{i8 0, i8 2} ; the value is in [0, 2)

A.5 TR2: -fno-strict-enums

```
enum X { A, B, C, D };
bool f(X *e) {
  return *e > 3;
}
```

```
define i1 @f(ptr %e) {
  %val = load i32, ptr %b, !range !1
  %r = icmp sgt i32 %val, 3 ; always false
  ret i1 %r
}
```

!1 = !{i8 0, i8 4} ; the value is in [0, 4)

A.6 FD1: -fignore-pure-const-attrs

```
int g() __attribute__((const));
int f() {
  return g() + g();
}
```

```
define i32 @f() {
  %c = call i32 @g()
  %r = mul i32 %c, 2
  ret i32 %r
}
```

A.7 FD2: -fdrop-ub-builtins

```
int f(int a) {
  __builtin_assume(a > 0);
  return a;
}
```

```
define i32 @f(i32 %a) {
  ; Note that no assembly is generated for this
  ; comparison. It is used just for optimizations.
  ; It is discarded afterwards.
  %cmp = icmp sge i32 %a, 0
  ; UB if a <= 0
  call void @llvm.assume(i1 %cmp)
  ret i32 %a
}
```

A.8 FD3: -fno-finite-loops

```
int g();
int f() {
```

```
define i1 @f(ptr %b) {
  %val = load i8, ptr %b
  %r = icmp sgt i8 %val, 1
  ret i1 %r
}
```

```
define i1 @f(ptr %e) {
  %val = load i32, ptr %b
  %r = icmp sgt i32 %val, 3
  ret i1 %r
}
```

```
define i32 @f() {
  %c1 = call i32 @g()
  %c2 = call i32 @g()
  %r = add nsw i32 %c1, %c2
  ret i32 %r
}
```

```
define i32 @f(i32 %a) {
  ; Assembly is generated for the comparison
  %cmp = icmp sge i32 %a, 0
  br i1 %cmp, label %ok, label %nok

ok:
  ret i32 %a

nok:
  ; exits the process
  call void @llvm.trap()
}
```

```

while (1) {
    g();
}
return 0;
}

define i32 @f() {
    br label %loop

loop:
    call i32 @g() ; function must have side effects
    br label %loop, !llvm.loop !1
}
!1 = !{"llvm.loop.mustprogress"}

```

```

define i32 @f() {
    br label %loop

loop:
    call i32 @g()
    br label %loop
}

```

A.9 PM1: -fdrop-align-attr

```

struct X {
    int a, b;
    int f() {
        return a + b;
    }
};

; Assumes that the "this" pointer points to a memory region with size
; at least that of the struct X (8 bytes).
define i32 @_ZN1X1fEv()(ptr align 4 dereferenceable(8) %this) {
    ; These loads can be executed speculatively.
    %ap = getelementptr inbounds %struct.X, ptr %this, i32 0
    %a = load i32, ptr %ap, align 4
    %bp = getelementptr inbounds %struct.X, ptr %this, i32 1
    %b = load i32, ptr %bp, align 4
    %r = add nsw i32 %a, %b
    ret i32 %r
}

define i32 @_ZN1X1fEv()(ptr dereferenceable(8) %this) {
    %ap = getelementptr inbounds %struct.X, ptr %this, i32 0
    %a = load i32, ptr %ap, align 4
    %bp = getelementptr inbounds %struct.X, ptr %this, i32 1
    %b = load i32, ptr %bp, align 4
    %r = add nsw i32 %a, %b
    ret i32 %r
}

```

A.10 PM2: -fdrop-deref-attr

```

struct X {
    int a, b;
    int f() {
        return a + b;
    }
};

define i32 @_ZN1X1fEv()(ptr align 4 dereferenceable(8) %this) {
    %ap = getelementptr inbounds %struct.X, ptr %this, i32 0, i32 0
    %a = load i32, ptr %ap, align 4

```

```

    %bp = getelementptr inbounds %struct.X, ptr %this, i32 0, i32 1
    %b = load i32, ptr %bp, align 4
    %r = add nsw i32 %a, %b
    ret i32 %r
}

```

```

define i32 @_ZN1X1fEv()(ptr align 4 %this) {
    %ap = getelementptr inbounds %struct.X, ptr %this, i32 0, i32 0
    %a = load i32, ptr %ap, align 4
    %bp = getelementptr inbounds %struct.X, ptr %this, i32 0, i32 1
    %b = load i32, ptr %bp, align 4
    %r = add nsw i32 %a, %b
    ret i32 %r
}

```

A.11 PM3: -fno-delete-null-pointer-checks

```

bool f(int *p) {
    *p = 3;
    return p == nullptr;
}

```

```

define i1 @f(ptr %p) {
    store i32 3, ptr %p
    ; always false
    ; otherwise the store would be UB
    %r = icmp eq ptr %p, null
    ret i1 %r
}

```

```

define i1 @f(ptr %p) null_pointer_is_valid {
    store i32 3, ptr %p
    %r = icmp eq ptr %p, null
    ret i1 %r
}

```

A.12 PM4: -fdrop-noalias-restrict-attr

```

int f(int *restrict p, int *restrict q) {
    *p = 3;
    *q = 4;
    return *p;
}

```

```

define i32 @f(ptr noalias %p, ptr noalias %q) {
    ; These two stores can't alias
    store i32 3, ptr %p
    store i32 4, ptr %q
    %r = load i32, ptr %p ; guaranteed to yield 3
    ret i32 %r
}

```

```

define i32 @f(ptr %p, ptr %q) {
    store i32 3, ptr %p
    store i32 4, ptr %q
    %r = load i32, ptr %p ; may yield 3 or 4
    ret i32 %r
}

```

A.13 PM5: -fno-strict-aliasing

```

int f(int *p, float *q) {
    *p = 3;
    *q = 4.0;
    return *p;
}

```

```

define i32 @f(ptr %p, ptr %q) {
    ; These two stores can't alias
    store i32 3, ptr %p, !tbaa !1
    store float 4.0, ptr %q, !tbaa !2
    %r = load i32, ptr %p ; guaranteed to yield 3
    ret i32 %r
}
; C type hierarchy
!0 = !{"char"}
!1 = !{"int", !0}
!2 = !{"float", !0}

```

```

define i32 @f(ptr %p, ptr %q) {
    store i32 3, ptr %p
    store float 4.0, ptr %q
    %r = load i32, ptr %p
    ret i32 %r
}

```

A.14 PM6: -fno-use-default-alignment

```

int f(int *p) {
    return *p;
}

```

```

define i32 @f(ptr %p) {
    ; assume integers are 4-byte aligned
    %r = load i32, ptr %p, align 4
    ret i32 %r
}

```

```

define i32 @f(ptr %p) {
    %r = load i32, ptr %p, align 1
    ret i32 %r
}

```

A.15 PM7: -no-enable-noundef-analysis

```

int f(int a) {
    return a;
}

```

```

; assume that function arguments can't be
; undef or poison
define i32 @f(i32 noundef %a) {
    ret i32 %a
}

```

```

define i32 @f(i32 %a) {
    ret i32 %a
}

```

A.16 PM8: -zero-uninit-loads

```

int f(bool c) {
    int a;
    if (c)
        a = 1;
    return a;
}

```

```

define i32 @f(i1 %c) {
  %a = alloca i32
  br i1 %c, label %true, label %ret

true:
  store i32 1, ptr %a
  br label %ret

ret:
  %r = load i32, ptr %a
  ; %r will be replaced with 1 since %a is
  ; either uninitialized (thus undef)
  ; or it is 1. undef can be replaced with 1.
  ret i32 %r
}

```

```

define i32 @f(i1 %c) {
  %a = alloca i32
  br i1 %c, label %true, label %ret

true:
  store i32 1, ptr %a
  br label %ret

ret:
  %r = load i32, ptr %a
  ; %r will be simplified to
  ; %r = select i1 %c, i32 1, i32 0` since the
  ; optimizer is forced to assume that
  ; uninitialized memory is zero.
  ret i32 %r
}

```

A.17 AA1: -fdrop-inbounds-from-gep -disable-oob-analysis

```

int glb;
int f(int *p) {
  glb = 2;
  p[3] = 4;
  return glb;
}

```

```

@glb = global i32
define i32 @f(ptr %p) {
  ; these two stores can't alias because
  ; if %p can't be inbounds of some offset
  ; of glb (4 bytes) and that offset plus 12
  store i32 2, ptr @glb
  %pp = getelementptr inbounds i32, ptr %p, i32 3
  store i32 4, ptr %pp
  %r = load i32, ptr @glb ; guaranteed to be 2
  ret i32 %r
}

```

```

@glb = global i32
define i32 @f(ptr %p) {
  ; these two stores may alias
  store i32 2, ptr @glb
  %pp = getelementptr i32, ptr %p, i32 3
  store i32 4, ptr %pp
  %r = load i32, ptr @glb
  ret i32 %r
}

```

A.18 AA2: disable-object-based-analysis

```

int f(int i, int j) {
  int *p = (int*)malloc(8);
  int *q = (int*)malloc(8);
  p[i] = 0;
  q[j] = 1;
  return p[0];
}

```

```
define i32 @f(i32 %i, i32 %j) {  
  %p = call ptr @malloc(i64 8)  
  %q = call ptr @malloc(i64 8)  
  ; these stores can't alias because  
  ; p+i and q+j are pointers  
  ; based on different objects  
  %pp = getelementptr inbounds i32, ptr %p, i32 %i  
  store i32 0, ptr %pp  
  %qq = getelementptr inbounds i32, ptr %q, i32 %j  
  store i32 1, ptr %qq  
  %r = load i32, ptr %p ; guaranteed to yield 0  
  ret i32 %r  
}  
}|  
define i32 @f(i32 %i, i32 %j) {  
  %p = call ptr @malloc(i64 8)  
  %q = call ptr @malloc(i64 8)  
  ; these stores may alias  
  %pp = getelementptr inbounds i32, ptr %p, i32 %i  
  store i32 0, ptr %pp  
  %qq = getelementptr inbounds i32, ptr %q, i32 %j  
  store i32 1, ptr %qq  
  %r = load i32, ptr %p ; we can't assume anything  
  ret i32 %r  
}
```

Received 2024-11-04; accepted 2025-03-06