Exploiting Undefined Behavior in C/C++ Programs for Optimization

A Study on the Performance Impact

Lucian Popescu, Nuno P. Lopes

Politehnica University Of Bucharest & University of Lisbon

CVE-2009-1897 Linux kernel

```
static unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    ...
    if (!tun)
      return POLLERR;
```

Implies tun != NULL

Always false

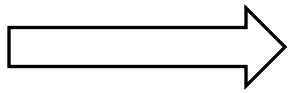
Compilers are designed to assume that programs do not contain UB!

What is the motivation for Undefined Behavior (UB)?

1. Performance: operations don’t have consistent semantics across CPUs
   - Signed integer overflow
   - Shift overflow
   - Unaligned memory access
2. Memory safety: pointer validity at dereference is undecidable or too expensive to verify
   - arr[idx]: Is arr allocated? Is idx inbounds? Is arr NULL?

Leveraging UB for optimizations

```
for (int i = 0; i < n; i++)
   p[i] = 42;
```



```
for (int i = 0; i < n; i++)
   *(p + sign_ext64(i)) = 42;
```

2x faster on my machine

32-bit machine:

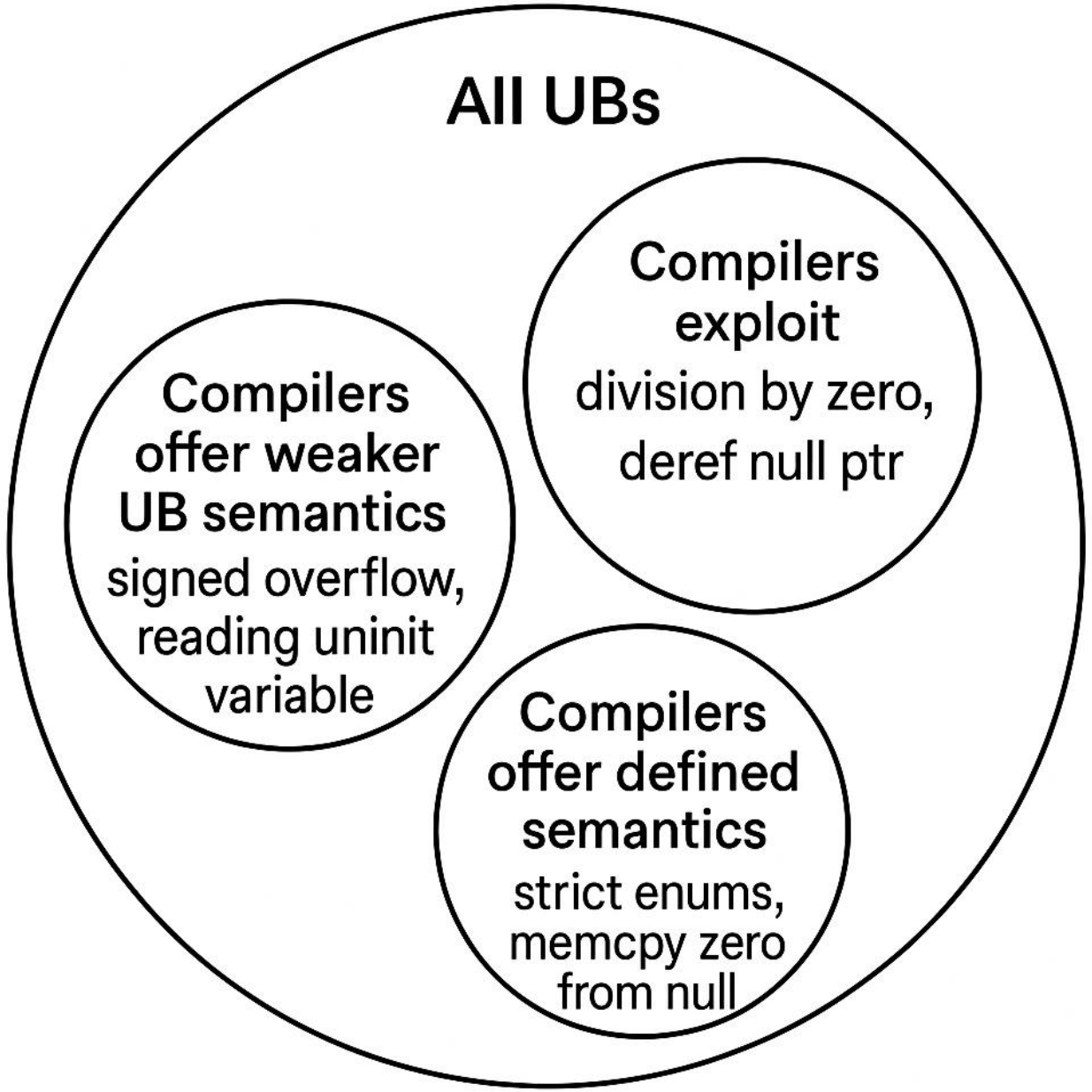
sizeof(i) = 4
sizeof(p) = 8

```
32 bits      / 64 bits

i = 1          1
i = 2          2
...
i = INT_MAX    INT_MAX
i = INT_MIN    INT_MAX+1
...
```

Transformed code:

sizeof(i) = 8
sizeof(p) = 8



The UB landscape

The cost of making UB defined

- C++26 is expected to add erroneous behavior which is stricter than UB (**uninitialized read**)
- C23 made the transition from UB to defined behavior (**signed overflow**)
- More effort is invested in defining some UBs or marking them as errors

- What's the performance cost of defining UB?
- We focus only on performance! This study is not about security implications or developer productivity.

Experimental setup

- 18 compiler flags to control UB
- 24 real-world benchmarks
- 3 hardware arches (AMD, ARM, Intel)

Category	Flags	Acronym	New?	Frontend?
Arithmetic Operations	-fcheck-div-rem-overflow	AO1	✓	✓
	-fconstrain-shift-value	AO2	✓	✓
	-fwrapv	AO3		✓
Type Ranges	-fno-constrain-bool-value	TR1	✓	✓
	-fno-strict-enums	TR2		✓
Function Domains	-fignore-pure-const-attrs	FD1	✓	✓
	-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 -mllvm -disable-oob-analysis	AA1	✓	
	-mllvm -disable-object-based-analysis	AA2	✓	

18 compiler flags

- Clang/LLVM 16
- Already existing (6 flags)
- New Clang flags (9 flags)
- New LLVM flags (3 flags)

```c
// C code
unsigned f(unsigned a, unsigned b) {
  return a << b;
}
```

```
; Without -fconstrain-shift-value (default)
define i32 @f(i32 %a, i32 %b) {
  %r = shl i32 %a, %b
  ret i32 %r
}
```

```
; With -fconstrain-default-behavior
define i32 @f(i32 %a, i32 %b) {
  %m = and i32 %b, 31
  %r = shl i32 %a, %m
  ret i32 %r
}
```

```
shl eax, cl

ret
```

Clang flags

- Generate IR that is free of UB
- -fconstrain-shift-value adds an extra and instruction before the shift

```c
// C code
int f() {
  int a;
  return a;
}
```

```
; Without -zero-uninit-loads (default)
define i32 @f() {
  %a = alloca i32
  %r = load i32, ptr %a
  ; %r will be undef since %a is not initialized
  ret i32 %r
}
```

```
; With -zero-uninit-loads
define i32 @f() {
  %a = alloca i32
  %r = load i32, ptr %a
  ; optimizer is forced to assume %r is 0
  ret i32 %r
}
```

LLVM flags

- IR cannot be modified to encode that reading %a is zero instead of undef
- Solution: add custom logic inside the optimizer to load zero from %a instead of undef

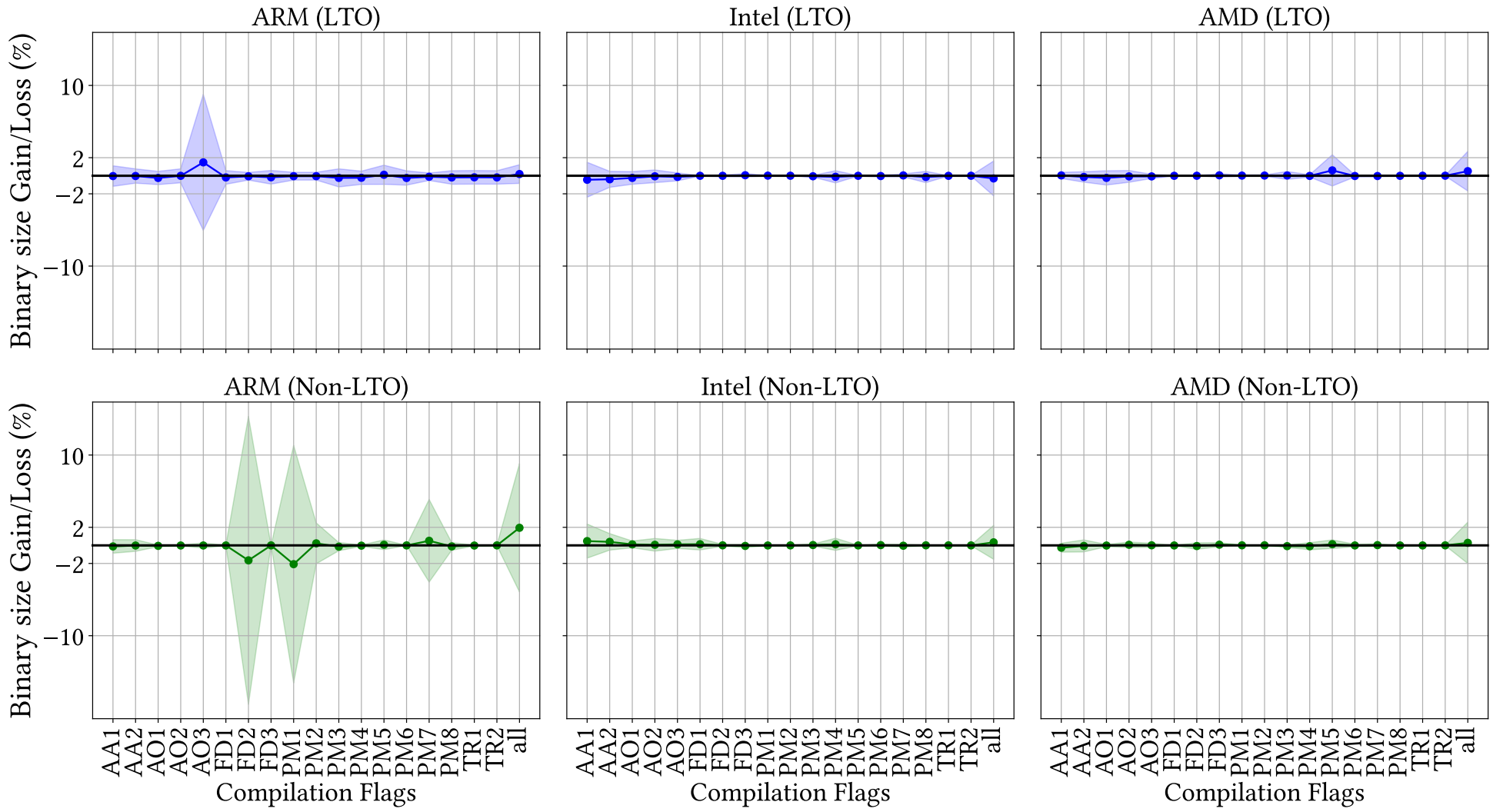
No	Benchmark	Category	Measurement Scale	kLoC
1	aom-av1-3.7.0	Video Encoding	frames/second	508
2	encode-flac-1.8.1	Audio Encoding	seconds	59
3	espeak-1.7.0	Speech Synthesizer	seconds	44
4	botan-1.6.0	Security	MB/second	148
5	john-the-ripper-1.8.0	Security	checks/second	315
6	openssl-3.1.0	Security	bytes/s	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

24 benchmarks

- Phoronix Test Suite
- 7.3 MLoC

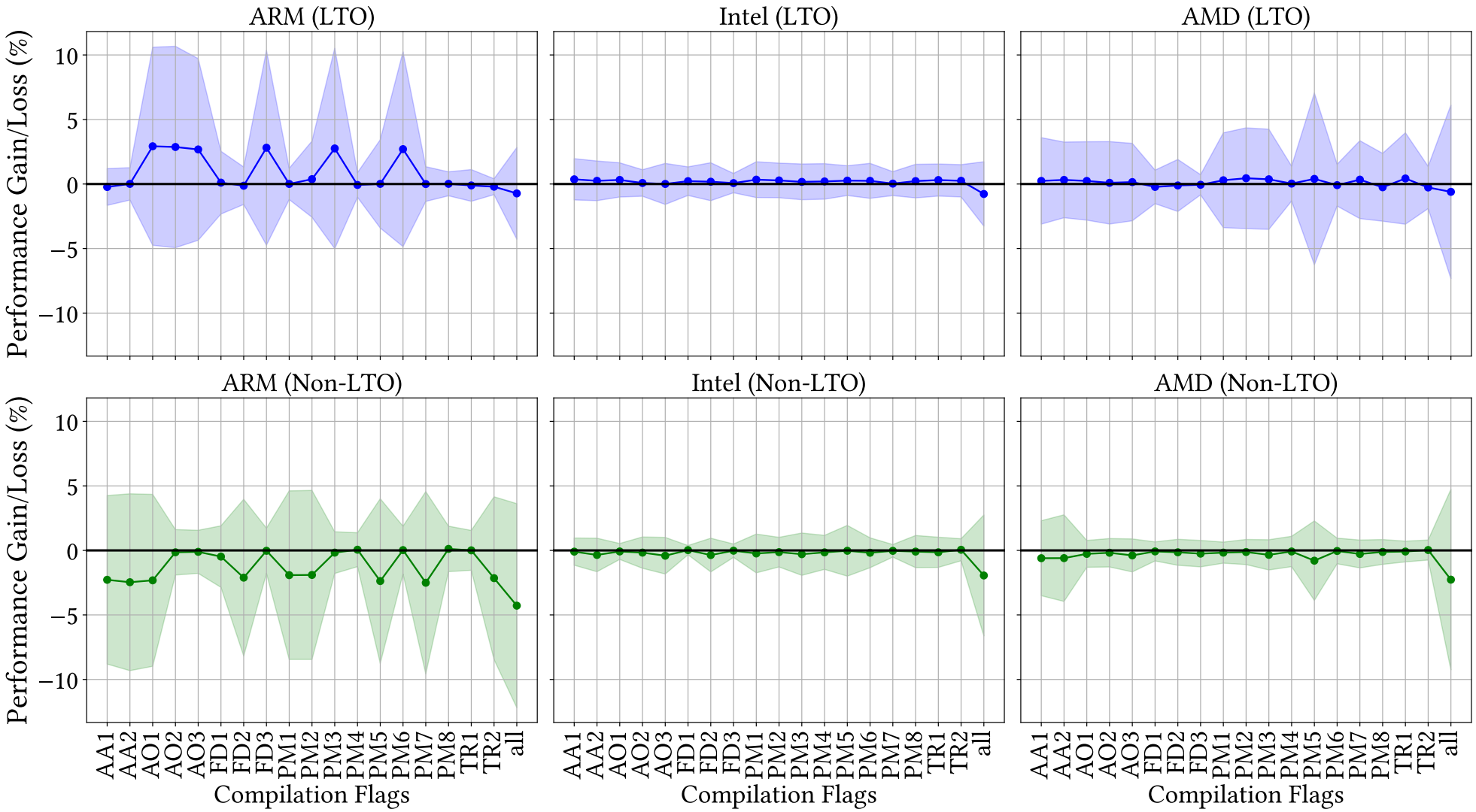
Binary size impact

- 6 setups: (AMD, ARM, Intel) x (Link Time Optimization (LTO), non-LTO)
- 18 individual flags + all flags combined
- Impact computed relative to base (no UB flag)


ARM (LTO)
Intel (LTO)
AMD (LTO)
ARM (Non-LTO)
Intel (Non-LTO)
AMD (Non-LTO)
Binary size Gain/Loss (%)
10
2
−2
−10
AA1
AA2
AO1
AO2
AO3
FD1
FD2
FD3
PM1
PM2
PM3
PM4
PM5
PM6
PM7
PM8
TR1
TR2
all
Compilation Flags

Performance impact

- 6 setups: (AMD, ARM, Intel) x (Link Time Optimization (LTO), non-LTO)
- 18 individual flags + all flags combined
- Impact computed relative to base (no UB flag)


ARM (LTO)
Intel (LTO)
AMD (LTO)
ARM (Non-LTO)
Intel (Non-LTO)
AMD (Non-LTO)
Performance Gain/Loss (%)
10
5
0
−5
−10
AA1
AA2
AO1
AO2
AO3
FD1
FD2
FD3
PM1
PM2
PM3
PM4
PM5
PM6
PM7
PM8
TR1
TR2
all
Compilation Flags

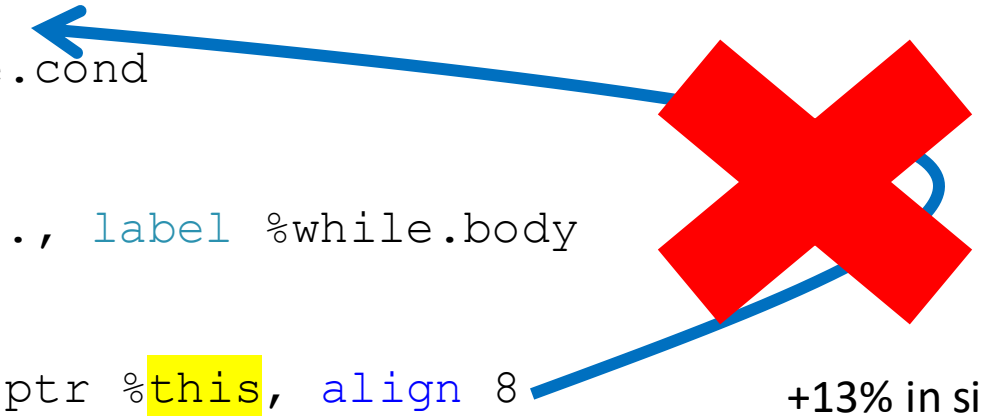
Recoverability	Root Cause	Flag	Benchmark	Impact
Recoverable with LTO	Pointer analysis	PM1	simdjson	-13%
	Pointer analysis	PM2	simdjson	-13%
Recoverable with moderate work	Loop vectorizer	AA1	jpegxl	-4%
	Alias analysis	AA2	espeak	-4.2%
	Inliner	AO2	zstd	-2.1%
	Pointer analysis	PM3	pbzip2	-3.2%
	Loop misalignment	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%

Recovering performance

Recover perf with LTO

-fdrop-align-attr

```
define i1 @run(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
```



+13% in simdjson

Recover perf with moderate work

```
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:
  ...
```

-4% slowdown

$$\%ptr = \%init, \%init + \%inc, \%init + 2 \times \%inc, \ldots, \%end$$

inbounds ensures address is always inside an object => no address space wrap around

Conclusion

- UB has real security impact when exploited by compiler optimizations
- There is growing interest in eliminating certain kinds of UB for safety

- We conducted the first performance study on removing UB from optimizations
- In many cases, removing UB had minimal performance impact
- We also showed that it’s possible to recover performance without relying on UB