

PRACTICAL REFERENCE FOR C++ ENGINEERS

UBSan Ultimate C++ Quality Guide

Undefined Behavior Sanitizer for Production C++

PDF Specialist · 28 October 2026

Contents

Executive Summary	3
What UB Is	3
The taxonomy that matters	3
Canonical UB categories	4
Check Groups Catalog	4
Clang checks	4
GCC suboptions	5
Clang Flags	6
GCC Flags	6
MSVC Support	6
Usage Examples	6
Combining with ASan TSan MSan	7
Output Interpretation	7
Suppressions Recover Ignore-lists	8
CI Integration	9
Strengths Limitations Overhead	9
Best Practices	9
Conclusions Roadmap	10
Known Limitations	10
Sources	10

Executive Summary

UndefinedBehaviorSanitizer (UBSan) is a compiler-instrumented checker that turns silent undefined behavior (UB) in C and C++ into loud, file-and-line runtime reports. It is cheap, typically a few percent overhead, needs no special hardware, and combines with AddressSanitizer. Every C++ team shipping production code should run UBSan routinely.

The essentials in 30 seconds

- Build with `clang++ -fsanitize=undefined -fno-sanitize-recover=all -g test.cc` and run the test suite.
- Add `address` to the same flag set for the standard ASan plus UBSan CI job.
- Use `-fno-omit-frame-pointer`, keep debug info, and treat new UBSan reports as build failures.
- Suppress only known third-party issues with ignore-lists or `__attribute__((no_sanitize("undefined")))`.

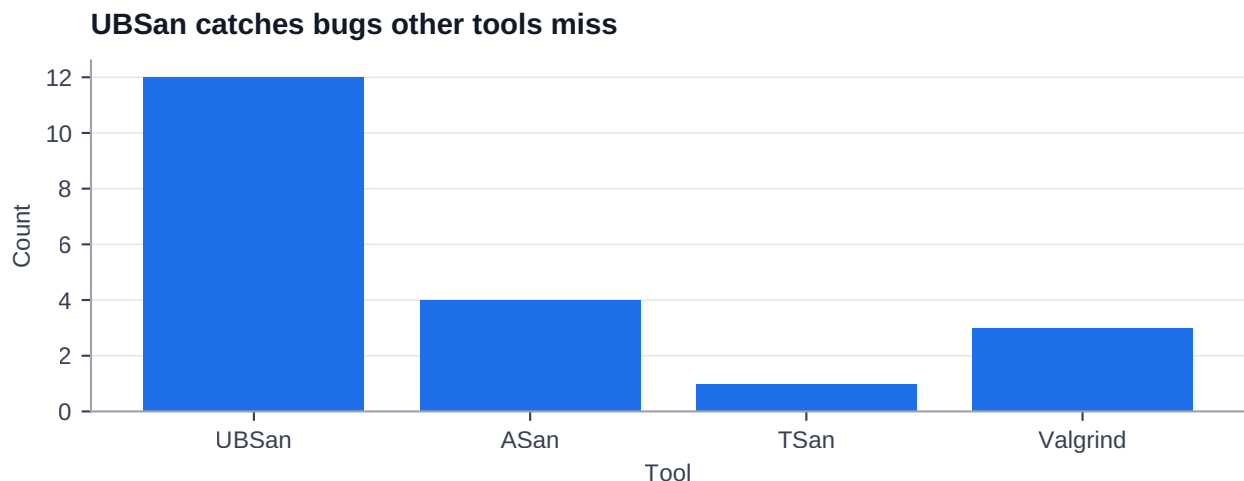


Figure 1. UBSan covers more UB classes than memory or thread tools.

What UB Is

The taxonomy that matters

C++ has four related but distinct concepts. Only the first is caught by UBSan:

- **Undefined behavior (UB):** the standard imposes no requirements at all. Examples: signed integer overflow, null dereference, use-after-lifetime. The compiler may assume it never happens.
- **Unspecified behavior:** several valid outcomes, the implementation need not document which. Example: order of evaluation of function arguments.
- **Implementation-defined behavior:** the implementation must document its choice. Example: size of `int`.

- **Ill-formed, no diagnostic required (IFNDR):** a broken program the compiler is not required to diagnose.

Canonical UB categories

The core language UB that UBSan targets includes signed overflow, division by zero, out-of-range shifts, null and misaligned pointers, out-of-bounds access, object lifetime violations, invalid casts, and missing returns. Annex and tooling references count well over one hundred distinct UB cases in the standard, which is why a sanitizer beats manual review.

Important

Optimizers exploit UB. Code that "works" at `-O0` can misbehave at `-O2` because the compiler assumed the UB path was unreachable. UBSan finds these assumptions before the optimizer does.

Check Groups Catalog

Clang checks

Check	What it catches	In undefined group
alignment	misaligned load or store	Yes
bool	load of invalid bool value	Yes
bounds	out-of-bounds array index	Yes
builtin	invalid builtin call	Yes
enum	load of invalid enum value	Yes
float-cast-overflow	float to int out of range	Yes
float-divide-by-zero	floating point divide by zero	No
integer-divide-by-zero	integer divide by zero	Yes
nonnull-attribute	null passed to nonnull parameter	Yes
null	null pointer use	Yes
nullability-arg / assign / return	nullability annotation violation	No
object-size	access past object size	Yes
pointer-overflow	pointer arithmetic overflow	Yes
return	missing return in value function	Yes

Check	What it catches	In undefined group
returns-nonnull-attribute	null returned from nonnull function	Yes
shift-base / shift-exponent	bad shift base or count	Yes
signed-integer-overflow	signed overflow	Yes
unreachable	supposedly unreachable code reached	Yes
unsigned-integer-overflow	unsigned overflow (nonstandard check)	No
unsigned-shift-base	unsigned shift UB variant	No
vptra	wrong dynamic type access	No
function	call through wrong function type	Yes
implicit-conversion	dangerous implicit conversions	No
invalid-objc-cast	bad ObjC cast	No

Meta-groups: `undefined` enables the standard set above; `undefined-trap` lowers them to traps.

GCC suboptions

GCC supports `-fsanitize=undefined` plus: alignment, bool, bounds, builtin, enum, float-cast-overflow, float-divide-by-zero, integer-divide-by-zero, nonnull-attribute, null, object-size, pointer-overflow, return, returns-nonnull-attribute, shift, signed-integer-overflow, unreachable, vptra. GCC lacks nullability, function, implicit-conversion, and unsigned variants.

Clang Flags

```
clang++ -fsanitize=undefined -g -fno-sanitize-recover=all app.cc -o app
clang++ -fsanitize=undefined -fno-sanitize=alignment app.cc
clang++ -fsanitize=signed-integer-overflow,null,alignment app.cc
clang++ -fsanitize=undefined -fsanitize-trap=all app.cc # no runtime, SIGILL on hit
clang++ -fsanitize=vptr -g app.cc # vptr needs RTTI, link with clang++
clang++ -fsanitize=undefined -fno-sanitize-recover=null -fsanitize-trap=alignment app.cc
```

Key options: `-fno-sanitize=` excludes checks; `-fno-sanitize-recover=` makes hits fatal per check; `-fsanitize-trap=` compiles checks to traps with no runtime; `-fsanitize-undefined-trap-on-error` is the trap shorthand; `-fno-sanitize-address-use-after-scope` tunes scope checks.

GCC Flags

```
g++ -fsanitize=undefined -g -fno-sanitize-recover=all app.cc -o app
g++ -fsanitize=undefined -fno-sanitize=alignment app.cc
g++ -fsanitize=undefined -fsanitize-undefined-trap-on-error app.cc
```

Version notes: GCC 5 introduced UBSan; GCC 8 added `-fsanitize=pointer-compare` and `pointer-subtract` style checks; newer GCC 12 to 14 improved diagnostics and `object-size` handling. Flag spelling matches Clang for the common subset. GCC accepts `-fsanitize-recover=` as an alias for the Clang recover spelling.

MSVC Support

MSVC supports a subset under `/fsanitize=address` plus runtime checks (`/RTC1`, `/GS`, `/SDL`) and static analysis (`/analyze`). There is no full UBSan port. Strategy: run UBSan on Clang or GCC CI jobs for portable code, and use MSVC `/RTC`, `/analyze`, and ASan on the Windows job. Keep code warning-clean at `/W4`.

Usage Examples

Minimal signed overflow reproducer:

```
int main(int argc, char **argv) {
    int k = 0x7fffffff;
    k += argc;
    return 0;
}
```

```
test.cc:3:5: runtime error: signed integer overflow:
2147483647 + 1 cannot be represented in type 'int'
```

More repros (build with `-fsanitize=undefined -g -fno-sanitize-recover=all`):

Check	Repro	Diagnostic
null	<code>int *p = nullptr; *p = 1;</code>	load of null pointer
alignment	<code>misaligned int load via buf+1</code>	misaligned address
shift	<code>1 << 32, 1 >> -1</code>	bad shift
divide-by-zero	<code>x / 0</code>	division by zero
bounds	<code>a[10] on int a[10]</code>	out of bounds
return	fall off non-void function	missing return
vptr	invalid downcast then member use	wrong dynamic type

Combining with ASan TSan MSan

Combination	Verdict
ASan + UBSan	Recommended default: <code>-fsanitize=address,undefined -fno-omit-frame-pointer -g -O1</code>
TSan + UBSan	Combinable (<code>thread,undefined</code>) but run separately; TSan dominates cost
MSan + UBSan	Combinable only on fully instrumented builds; keep separate jobs
ASan + TSan + UBSan	Not supported together; split jobs

Run two jobs: (1) ASan plus UBSan for memory and UB, (2) TSan plus UBSan for races. OSS-Fuzz uses ASan plus UBSan by default.

Output Interpretation

A UBSan report names the file, line, column, check, and values:

```
main.cpp:16:9: runtime error: signed integer overflow:
2147483647 + 1 cannot be represented in type 'int'
```

Read it as: location first, then the violated rule, then the offending values. Recover mode prints and continues; `-fno-sanitize-recover=all` aborts on first hit, which is what CI wants. Trap mode raises `SIGILL / SIGTRAP` with no message, useful for minimal binaries. Always rebuild with `-g` and `-fno-omit-frame-pointer` for usable stacks.

Suppressions Recover Ignore-lists

```
clang++ -fsanitize=undefined -fno-sanitize-recover=all app.cc  
clang++ -fsanitize=undefined -fsanitize-ignorelist=ubsan.txt app.cc
```

```
# ubsan.txt  
src:third_party/*  
fun:*legacy_hash*
```

Options: `-fno-sanitize=` disables a check globally; `-fsanitize-ignorelist=` excludes files or functions; `__attribute__((no_sanitize("undefined")))` excludes one function;

`UBSAN_OPTIONS=print_stacktrace=1:halt_on_error=1` tunes the runtime. Suppress narrowly, expire suppressions, and never suppress first-party code permanently.

CI Integration

```
cmake -DCMAKE_CXX_COMPILER=clang++ \
  -DCMAKE_CXX_FLAGS="-fsanitize=address,undefined -fno-sanitize-recover=all \
    -fno-omit-frame-pointer -g -O1" ..
ctest
```

Guidance: fail the build on any report; keep sanitizer jobs separate from release jobs; pin compiler versions; archive logs with artifacts; add a TSan job for threaded code. Example GitHub Actions job: configure with sanitizer flags, build, run `ctest`, upload logs.

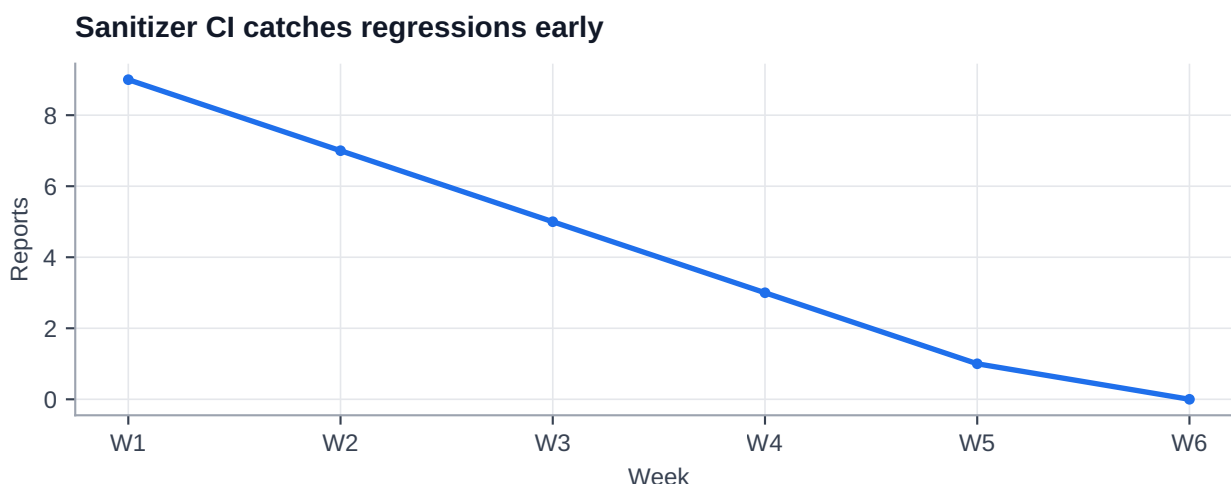


Figure 2. Reports fall to zero as suppressions and fixes land.

Strengths Limitations Overhead

Strengths: tiny overhead (single-digit percent is typical), precise file and line reports, no special hardware, composes with ASan, catches optimizer-visible UB early.

Limitations: dynamic only (unexecuted code is unchecked); no cross-translation-unit type checking beyond instrumentation; `vptr` needs RTTI; trap mode gives no message; some checks (unsigned overflow, implicit conversion) are opt-in and noisy.

Overhead: UBSan alone is cheap enough for unit tests and fuzzing. ASan plus UBSan adds ASan's memory cost. TSan dominates any job it joins. Measure on your own suite rather than trusting rules of thumb.

Best Practices

1. Enable `undefined` plus `address` in CI from day one.
2. Abort on first error (`-fno-sanitize-recover=all`) in CI; recover locally.
3. Always ship `-g` and `-fno-omit-frame-pointer` in sanitizer builds.

4. Fix or narrowly suppress; review suppressions quarterly.
5. Add `vptr`, `float-cast-overflow`, and `implicit-conversion` explicitly where they matter.
6. Fuzz sanitizer builds; OSS-Fuzz pairs ASan with UBSan.
7. Keep compiler versions pinned and logs archived.

Conclusions Roadmap

UBSan is the cheapest high-value quality gate a C++ team can add. Adopt ASan plus UBSan now, add TSan coverage for concurrent code, and track new Clang and GCC checks each release. The roadmap: instrument everything in CI, drive reports to zero, and keep them there.

Known Limitations

1. Dynamic coverage only: code that never runs is never checked.
2. `vptr` requires RTTI and fails under `-fno-rtti`.
3. Trap mode reports no source location, only a signal.
4. No detection of unspecified or implementation-defined behavior.
5. Opt-in checks (unsigned overflow, implicit conversion) are noisy at scale.
6. MSVC has no full UBSan equivalent; cross-compiler CI is required.
7. Suppression ignore-lists need maintenance or they rot.

Sources

- Clang UBSan docs: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
- Android UBSan reference: <https://source.android.com/docs/security/test/ubsan>
- Conan sanitizer guide: <https://docs.conan.io/2/examples/security/sanitizers.html>
- Clang ThreadSanitizer docs: <https://clang.llvm.org/docs/ThreadSanitizer.html>