A Profile for Runtime-Checkable Core-Language Undefined Behavior: std::core_ub

Document Number:	P4317R0
Date:	2026-07-14
Intent:	propose
Audience:	EWG, SG22

Reply-to:	Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

The runtime-checkable cases of core-language undefined behavior can be guarded by a single standard profile, with none of the changes to the definitional machinery of the standard that a Contracts-based routing would require.

The C++ standard specifies a finite, enumerable set of core-language operations whose misuse has undefined behavior, and most of them can be checked at run time. This paper proposes `std::core_ub`, a profile under the framework of P3589R2 that guards those cases: when it is enforced, a checkable operation whose precondition is violated ends the program rather than proceeding into undefined behavior. The profile owns its guarantee, its enumeration, and its response to a violation directly, so it needs no foundational wording changes, it leaves the meaning of the `noexcept` operator untouched, and it follows every design principle in the committee's standing document SD-10. The form it standardizes - a named set of checks selected per build, terminating on a violation - is what production hardening ships across eight systems today, with measured cost as low as a third of a percent. The paper sets out four candidate responses to a violation, each drawn from a shipping deployment, and leaves the choice among them to the profile author.

Revision History

R0: July 2026

- Initial version.

---

1. Introduction

The C++ standard specifies a finite, enumerable set of core-language operations whose misuse has undefined behavior, and most of those operations can be checked at run time. This paper proposes to guard them with a single standard profile, `std::core_ub`, under the framework of P3589R2 [3]. When the profile is enforced, a checkable operation whose precondition is violated ends the program rather than proceeding into undefined behavior.

The enumeration that makes this possible is the work of Doumler and Berne. P3100R8 [1] identifies every case of explicit core-language undefined behavior, classifies each by how it can be diagnosed, and determines which cases admit a well-defined replacement. Of its cases, 77 are checkable at run time (as enumerated in P3100R8's Appendix A), and those 77 are what this profile guards. The enumeration is reproduced, with credit, in Appendix A.

This profile takes that enumeration and specifies it as a profile rather than as an extension of the C++26 Contracts machinery. The relationship between the two approaches, and where they differ, is the subject of the companion papers P4297R0 [4] and P4306R0 [2]; this paper does not restate their arguments, and cites them where they apply.

The contributions are four:

1. A profile specification, `std::core_ub`, covering the 77 runtime-checkable cases of core-language undefined behavior (as enumerated by P3100R8) under the P3589R2 framework (Section 2).
2. A demonstration that the profile provides this coverage with none of the six foundational wording changes P3100R8 requires (Section 3).
3. A comparison of four candidate responses to a violation, each drawn from a production deployment, presented for the profile author to weigh (Section 2.3).
4. An evaluation of the profile against the committee's adopted design principles in SD-10 (Section 5).

The paper assumes one thing, stated plainly: that a safety feature is stronger when it standardizes a form already shipping in production than when it standardizes a form that has not shipped. Section 6 gives the deployment record.

---

2. Design

A profile specification should state the guarantee it offers before the list of places it touches. P4222R2 [5] puts the principle plainly: "it is important that we specify the guarantee offered, rather than just long lists of places in the language affected." The guarantee comes first here; the enumeration that backs it is Appendix A.

2.1 The guarantee

When `std::core_ub` is enforced over a region of code, no core-language operation in that region has undefined behavior at run time. Every runtime-checkable precondition among the cases identified by the analysis of Doumler and Berne in P3100R8 [1] is verified before the operation it guards, and a violated precondition ends the program rather than proceeding into undefined behavior.

The guarantee is scoped to correct programs in the ordinary way. A program with no undefined behavior means exactly what it meant without the profile; the checks pass silently and the observable behavior is unchanged. Only a program that would otherwise have executed one of the enumerated operations under a violated precondition sees any difference, and the difference is termination in place of undefined behavior. This is the constraint P3589R2 [3] places on every profile: a profile does not change the meaning of a well-formed program that has no undefined behavior.

2.2 Activation

The profile is activated through the framework syntax of P3589R2 [3]. A translation unit opts in with a profile attribute on its first declaration:

```
[[profiles::enforce(std::core_ub)]];
```

The dominion of the profile runs from that attribute to the end of the translation unit. A declaration or statement may opt out with `[[profiles::suppress(std::core_ub)]]`, the framework's local escape for code that must use an unchecked construct where the programmer has established correctness by other means. No annotation appears in ordinary user code; enforcement is a build-level choice, and suppression is the rare exception.

2.3 The response to a violation

The profile guards 77 cases (the runtime-checkable cases enumerated in P3100R8's Appendix A). What happens when a guard fails is a design choice with a deployed record behind it, and it is left open here. Three candidate responses follow, drawn from what production systems actually ship, for the profile author to select among. All three share one property: none continues past the violation into the state the language leaves undefined.

The three candidates:

1. **Trap.** The violation is a trap instruction. Diagnostics are recovered out of process by a crash reporter that maps the trap address to source. This is the smallest possible codegen and the form Apple's `-fbounds-safety` and libc++ hardening ship. It generates no in-process diagnostic.
2. **Diagnostic, then abort.** The program prints the failed check and its source location, then calls `abort()`. libstdc++ ships this; the diagnostic is triage material, produced by the terminating response itself. It costs the code size of the diagnostic strings.
3. **Non-returning handler.** A replaceable, profile-specific function receives the violation, may log or report it, and must not return; if it returns, the program terminates. This is the shape of Bloomberg's `bsls_assert` where the post-violation state is undefined: the handler is invoked, and if it returns, the program terminates. Bloomberg's log-and-continue facility, `bsls_review`, operates at the library level, where the post-violation state remains defined, and is not applied to the core-language-undefined cases this profile guards. It gives the deployment one hook for logging and telemetry, at the cost of a customization point to specify.

Interoperation with the C++26 contract-violation handler is possible but is not a peer of the three. A deployment that wants a profile violation to construct a `std::contracts::contract_violation` and invoke the replaceable handler with a terminating semantic can have it, reusing a customization point already in the working draft. It is set apart from the three above because it is the only response that routes through the C++26 Contracts runtime, reintroducing the dependency the rest of this design avoids (Table 2), and because routing every checking facility through a single handler slot has no deployed precedent (see P4306R0 Section 9 [2]). It is available as an interop path for a deployment already invested in that handler, not as one of the responses the profile stands on.

No response is preferred here; the three are presented for the profile author to weigh. The full response space for a throwing check on a core-language operation, and the reason a terminating response leaves the meaning of the `noexcept` operator untouched, are analyzed in P4308R0 [21]. This choice governs the 62 cases whose violation the profile answers by ending the program. The other 15 cases have a well-defined replacement (Appendix A.4): for those the profile defines the operation's meaning directly (Section 2.4), so the operation yields that defined value on every conforming implementation rather than terminating. Because that value is defined and is the same on every conforming implementation, the no-continuation-into-undefined-behavior property holds and the meaning does not vary by build.

A profile of exactly this shape has been proposed before. P3608R0 [12] (Dos Reis, Voutilainen, Wakely) asked C++26 to ship "a concrete profile that switches on the standard library hardening, and makes the violations of hardened preconditions just terminate the program, without any additional flexibility for C++26," with vendors "encouraged not to close the door for other violation handling strategies... in the future." That is the arrangement here: a profile that enforces checking, terminates on a violation, and defers the richer response designs. The one difference is scope. P3608R0's profile switches on standard-library hardening, while `std::core_ub` applies the same shape to the core-language cases enumerated in Appendix A. The terminating-profile shape is therefore not novel; it is the shape a framework author already proposed for the library domain.

2.4 Replacement behavior

For the 15 guarded cases with a well-defined replacement (12 unconditional, 3 for built-in types only; enumerated in Appendix A.4), the profile defines the meaning of the operation directly, and that meaning is fixed for every conforming implementation rather than left to a per-build choice between terminating and continuing. This is the authorship P3984R0 [6] grants a profile: "A profile cannot change the semantics of a program beyond defining the meaning of some forms of undefined behavior." Signed overflow is defined as wraparound, a conversion out of range as an erroneous value, and so on, per Appendix A.4. Fixing the meaning is deliberate: a construct whose result depended on the build configuration would be the semantic instability P2834R1 [24] warns against, so the profile defines these operations once, the same way everywhere. P3100R8 reports 17 such cases over its full 80-case enumeration; the count is 15 here because two of those 17 fall among the three cases excluded as not runtime-checkable: the assumptions case, whose replacement is to ignore the assumption, and the non-terminating-loop case, whose replacement is to do nothing. The remaining 15 are the guarded cases listed in Appendix A.4.

2.5 Checking tiers

The profile's guarantee is the full set: over an enforced region, none of the 77 cases has undefined behavior at run time. That guarantee is fixed and does not vary with the build. Of the 77, 19 are locally diagnosable and can be checked at any optimization level for negligible cost; the remaining 58 require instrumentation of the kind sanitizers provide, so the full guarantee carries the cost of that instrumentation.

An implementation may still ship the locally diagnosable subset as a cheaper build mode, the way libc++ ships `fast`, `extensive`, and `debug`. Such a mode is an adoption aid, not a weaker enforcement of the profile: a build that checks only the 19 is not `std::core_ub` partially enforced, it is a diagnostic tool below the profile, in the same sense that selecting a subset of `-fsanitize=` checks is a tool rather than a distinct guarantee. Enforcing the profile means all 77. This keeps the profile a single named guarantee that means one thing everywhere - the semantic stability P2834R1 [24] requires (Section 2.4) - rather than a family whose meaning shifts with the build, and it adds no tiering machinery to the framework: the subsets are quality of implementation, not profile structure the user must assemble.

2.6 Composition

`std::core_ub` composes with the other standard profiles under P3589R2's rules; all standard profiles are compatible with each other. Its closest neighbor is the initialization profile of P4222R2 [5], which is purely compile-time and carries no run-time cost. The two divide the work cleanly: `std::init` proves initialization safety statically and rejects what it cannot prove, while `std::core_ub` catches at run time the undefined behavior that no static analysis can rule out in the general case. A program may enforce both, taking the static guarantee where it is available and the runtime guarantee everywhere else.

2.7 Composition across translation units

Enforcement is a per-region choice, so a program may enforce `std::core_ub` over some translation units and not others. The boundary between an enforced translation unit and an unenforced one is well defined, and the guarantee degrades gracefully across it.

The two classes of check behave differently at the boundary. A locally diagnosable case (Appendix A.1) is checked at the operation itself, inside the enforced region, so it holds regardless of how any other translation unit was compiled: a null check, a division-by-zero check, or an alignment check inserted before the operation needs nothing from the rest of the program. The cases that track state across the program (the lifetime, type, and provenance cases of Appendix A.3) are as complete as the instrumentation's coverage of the objects they touch. An object that enters an enforced region from an unenforced translation unit may carry no tracking state, and the implementation then cannot diagnose a violation on it. This is the graceful degradation every sanitizer already exhibits under partial instrumentation: partial coverage yields partial diagnosis, never a false guarantee, and never a change to the meaning of the unenforced translation unit.

That the boundary is harmless at all is a property of the profile's design. Because it introduces no new language construct and changes no type - not `noexcept`, not the ABI of any operation (Section 3) - an enforced translation unit and an unenforced one link and run together with no ODR or ABI hazard. The unenforced unit has today's behavior (Section 2.1); the enforced unit has its checks. Nothing the profile does is part of any interface, so nothing about it has to match across the boundary. A

guarantee that had to be established program-wide before it held anywhere would be far harder to adopt; this one holds over exactly the regions that enforce it, and a deployment can widen those regions one translation unit at a time.

---

3. Relationship to P3100R8

This profile is built on the work of P3100R8. The enumeration of every case of explicit core-language undefined behavior, the classification of each by whether and how it can be diagnosed, and the identification of the cases that admit a well-defined replacement: that is the analysis of Doumler and Berne, and it is a contribution to every safety effort regardless of which mechanism carries it. This profile could not have been specified without it. Appendix A is their enumeration, used with gratitude and cited as theirs throughout.

What this section examines is narrower than the enumeration and separate from it: the claim that the enumerated cases must be guarded through implicit contract assertions, with a profile defined as a preset over that machinery. The enumeration is the data. The routing is the architecture. The data is portable to either architecture, and this profile carries it under the other one, where the profile owns the guarantee directly and the implementation strategy (contract assertions, compiler intrinsics, sanitizer instrumentation, or anything else that catches the case) is a quality-of-implementation matter.

One case shows the portability concretely. For `{expr.mul.div.by.zero}` ([expr.mul]/4), P3100R8's checking strategy is to check that the divisor is nonzero (Appendix A.1). That check is a predicate on a value, not a construct of any one framework: an implementation can carry it as an implicit contract assertion, as a compiler-inserted branch before the division, or as sanitizer instrumentation, and the predicate tested is identical in each. The enumeration names the predicate to check; the architecture names what runs it. Where P3100R8 phrases a strategy in Contracts terms - "insert `pre(false)`" into the pure-virtual stub for `{class.abstract.pure.virtual}` (Appendix A.1) - the obligation beneath the phrasing is the same neutral check, that the call does not reach the pure-virtual stub, which any architecture can insert. This is why the same enumeration serves both proposals (Section 3.3).

3.1 The six foundational clauses are not needed

P3100R8's wording rests on six foundational changes to the definitional machinery of the standard, catalogued in P4297R0 [4] Table 2. Each exists to create, in the Contracts space, a capability that the Profiles framework of P3589R2 already provides in the Profiles space. Under the profile, none of the six is required.

Table 1. P3100R8's six foundational clauses and their status under the profile.

P3100R8 clause	Purpose	Under `std::core_ub`
[defns.undefined]	Redefine UB as an implicit contract assertion	Not needed. UB stays as-is in the standard; the profile adds rules on top of the existing language (P3589R2), so no redefinition is required.
[defns.unconstrained]	New term for the residual state	Not needed. Nothing takes the existing term, so no replacement term is required.
[basic.contract.general]	Split assertions into explicit and implicit	Not needed. No "implicit assertion" concept exists under the profile; a checked operation is a checked operation, and the mechanism is quality of implementation.
[basic.contract.eval]	Add the assume semantic for implicit assertions	Not needed. With the profile inactive the program has today's behavior, so the problem the assume semantic exists to solve does not arise.
[intro.abstract] 3+a	A guarding assertion for every UB operation	Supplied by the profile. Its enumeration (Appendix A) names the 77 guarded operations directly and `[[profiles::enforce(...)]]` attaches checking to the dominion, so no blanket core-language clause is required.
[basic.contract.implicit]	Define implicit assertions normatively	Supplied by the framework. P3589R2 makes a failure to satisfy a profile constraint a diagnosable rule, so the normative weight comes from the framework, not a new core-language section.

The pattern across the table is one fact stated six times: the Profiles framework already did this work once. P3100R8 redoes it for a different substrate. A profile that reuses the framework inherits the result and adds nothing to the definitional machinery of the standard.

This also answers the concern that a systematic UB framework leaves a profile with little of its own to standardize. The profile standardizes the guarantee, the enumeration of what it guards, and the response to a violation. That is a complete feature, specified here, owing no foundational wording to any other proposal.

3.2 A property comparison

The two approaches address the same 77 cases (the runtime-checkable cases enumerated in P3100R8). They differ on nearly everything else. Table 2 sets the properties side by side.

Table 2. Property comparison for the same guarded cases. The "meanings one operation can carry" row counts the distinct meanings an implementation may assign to a single checked operation, not whether checking is enabled.

Property	`std::core_ub`	P3100R8
Foundational wording changes	0	6
Runtime-checkable cases covered	77	77
Meaning of `noexcept(expr)`	Unchanged	Conceptual meaning changed: `true` means "cannot throw unless there is a contract violation" (P3100R8 Section 5.5)
Meanings one operation can carry	1 (within an enforced region, one meaning; outside it, unchanged)	Up to 5 (ignore, observe, enforce, quick-enforce, assume), selected implementation-defined per case
Normative effect on today's implementations	Enforcement catches the case	"All existing implementations of C++ are already conforming"
Dependency chain	P3589R2 (framework)	P2900R14 [17] + P3400R3 [18] + 6 new clauses
Distinctive machinery, implementation status	Framework implemented in Clang (C++ Alliance, public); the profile's UB checks not yet implemented	Implicit contract assertions and Labels not implemented
Production deployment of the standardized form	8 systems (Section 6)	None
Response in production hardening	Trap or abort (deployed, measured)	Replaceable handler + violation object (undeployed)

Two entries carry the section. The zero-versus-six on foundational wording is the structural fact, and the redefinition of `noexcept` is the one that reaches ordinary code: under P3100R8's Section 5.5, `noexcept(expr)` "changes its conceptual meaning" so that `true` "now effectively means 'evaluating this expression cannot throw unless there is a contract violation'" [1]. Under the profile, with a terminating response, `noexcept` means what it has always meant, because a trap does not throw; P4308R0 [21] analyzes the full space of responses to a throwing check and why the terminating ones avoid that shift. The remaining rows are documented in Section 6 (deployment) and Section 7 (the record).

3.3 What the profile specifies for a guarded case

The claim that the profile leaves each of the 58 instrumented cases underspecified is worth answering on a concrete case, because the line between what the profile fixes and what it leaves to the implementation is the same line P3100R8 draws, and the same one every sanitizer draws.

Take `{lifetime.outside.glvalue.access}` ([basic.life]/8), an access to a glvalue outside its object's lifetime. The profile fixes one thing normatively: over an enforced region, this access does not proceed into undefined behavior - the violation is detected and the response of Section 2.3 applies. What the profile does not fix is how the access is detected, and Appendix A.3 records the strategy without mandating it: track the lifetime and type of the storage. Whether an implementation does that with sanitizer-style shadow memory, with pointer capabilities, or with any other mechanism is quality of implementation. Take `{expr.call.different.type}` ([expr.call]/5), a call through a function pointer of the wrong type. Again the profile fixes the guarantee - the mistyped call does not proceed into undefined behavior - and leaves the mechanism, tracking the function type by address, to the implementation.

The point of the two cases is that the strategy column of Appendix A is not this profile's invention; it is the analysis of Doumler and Berne, and P3100R8 leaves the same mechanism to the implementation that this profile does. Neither proposal specifies, for `{lifetime.outside.glvalue.access}`, the representation of the lifetime metadata or the instructions that consult it, because neither can without foreclosing valid implementations. So "underspecified" cuts the same way for both, or for neither: both name the operation, both name the guarantee, both name a checking strategy, and both leave the instrumentation to the implementer. What differs is everything in Table 2 - the wording changes, the `noexcept` meaning, the number of meanings one operation can carry - not the specificity of the per-case checking obligation, which is identical between the two because it is the same enumeration.

---

4. Coexistence with Legacy Assertion Facilities

A safety mechanism does not arrive in an empty field. The standard `assert` macro and the many project-specific assertion facilities deployed across the C++ ecosystem already check preconditions and invariants, and a new mechanism should say how it sits alongside them. This section records that for completeness; it draws no comparison in the profile's favor.

P3290R4 [19] (Berne, Doumler, Lakos) identifies this need and proposes a path for the Contracts routing: a library API that lets legacy macros invoke the C++26 contract-violation handler, and an opt-in macro (`ASSERT_USES_CONTRACTS`) that routes the standard `assert` macro through that handler. The need is real and the paper names it well. Under the Profiles routing the same need is met through the profile's own response rather than the contract-violation handler; what follows is how, and

where the profile deliberately stops short.

4.1 How existing facilities respond to a failed check

Deployed assertion facilities already share a dominant response to a failed check: they stop the program. Table 3 surveys representative facilities across the ecosystem.

Table 3. Response to a failed check across deployed assertion facilities.

Facility	Default response	Continues past violation?	Replaceable handler?
C `assert` / `<cassert>`	diagnostic, then `abort()`	No	No (on/off via `NDEBUG`)
glibc `_FORTIFY_SOURCE`	`__chk_fail()`, then `abort()`	No	No
Bloomberg `bsls_assert`	log, then `abort()` (default)	No	Yes (handler must not return)
Bloomberg `bsls_review`	log	Yes	Yes (handler may return)
Abseil `ABSL_HARDENING_ASSERT`	immediate `abort()`	No	No
GSL `Expects` / `Ensures`	`std::terminate()`	No	No (throw option removed)
C++26 Contracts, `enforce`	handler, then terminate	No	Yes

Termination is the deployed norm for a detected violation. Log-and-continue - Bloomberg's `bsls_review` and the C++26 `observe` semantic - is deployed too, and is a first-class capability where it applies: staged rollout at the library level, where the post-violation state remains defined. It is not the shipping response to core-language undefined behavior, where the surveyed deployments terminate. The profile's terminating response (Section 2.3) is the posture these facilities take for the class it guards.

4.2 Integration and the `assert` macro

A legacy facility integrates with the profile by routing a failed check to whichever response the profile author selects in Section 2.3, in place of its own ad hoc termination. The integration point is the profile's response rather than the program-wide contract-violation handler, so ownership of the response stays with the safety feature. Granularity follows the framework: the profile's dominion runs to the end of the translation unit, and `[[profiles::suppress(std::core_ub)]]` gives a scope that needs a different response a local escape, without a separate preprocessor macro per header.

The standard `assert` macro fits the same pattern. When `std::core_ub` is enforced and `assert(expr)` fails, the profile's selected response applies and the program ends rather than proceeding. An `assert` failure is not itself undefined behavior, so it is not covered by the profile's core guarantee (Section 2.1); the profile's treatment of `assert` is an extension of its response to the C library's own checking facility, on the same reasoning P3290R4 gives - that the standard's safety infrastructure should cover the checking facilities users already have.

4.3 What the profile does not do

The profile does not provide log-and-continue past a violated precondition for the core-language-undefined class, and it does not claim to. P3290R4 offers `handle_observed_contract_violation()`, which continues after the handler returns; the profile stops instead. This follows the deployed boundary: Bloomberg's `bsls_review` [20] logs and continues at the library level, where the post-violation state is defined, while `bsls_assert` [20] terminates where the post-violation state is language-undefined. The case for terminating on the core-language-undefined class, and the reading of that Bloomberg boundary it rests on, is set out in full in the companion P4310R0 [26]. For the 15 guarded cases with a well-defined replacement (Appendix A.4), the operation has the profile-defined value and does not reach this boundary; for the remaining 62, termination is what every surveyed production deployment ships (Section 6). The handler still runs for logging and telemetry before termination; only continuation past language-undefined state is omitted.

---

5. SD-10 Section 4.1 Describes a Safe-by-Default Feature with an In-Source Opt-Out

EWG adopted SD-10 [7] in December 2024 as the standing document governing language-evolution design. Its Section 3 reaffirms the key principles of Stroustrup's *The Design and Evolution of C++* [8] and its Section 4 adds more. Its Section 4.1, "Make features safe by default, with full performance and control always available via opt-out," describes a feature that is safe by default and carries an in-source opt-out for the hot path. That is the profile model: enforce by default, `[[profiles::suppress(...)]]` where the programmer takes control.

Tables 4a and 4b score both approaches, the first against SD-10's own principles and the second against the further D&E principles SD-10 builds on. Each cell carries its reasoning, per the even-handed-comparison standard the Direction Group states in P2000R5 [9] Section 5.4: it is "not acceptable" to present "only advantages for a 'favored proposal' and only 'disadvantages' for an unfavored alternative." A reader who disputes a verdict can weigh the reasoning in the cell against the cited section.

Table 4a. The approaches measured against SD-10's principles.

Principle	`std::core_ub`	P3100R8
4.1 Safe by default, opt-out for control	Yes. Enforcement makes the dominion safe by default; `[[profiles::suppress(std::core_ub)]]` is the in-source opt-out.	No. Both require enabling checking, so the distinction is what enabling buys. Once enforced, the profile's dominion has one guaranteed behavior. Under P3100R8 the evaluation semantic is implementation-defined per case even when checking is enabled (Section 5.2), and the ignore semantic is a conforming choice, so enabling does not by itself yield a safe result.
4.3 Express intent: "what, not how"	Yes. `[[profiles::enforce(std::core_ub)]]` names the guarantee, not the checking method.	No. The profile attribute names the guarantee - no core-language undefined behavior over the dominion. A Label selects the evaluation semantic for an operation or a group, from the five P3100R8 provides; that selection is the checking method expressed per case, not the guarantee.
4.4 Avoid viral annotation	Yes. One attribute at the top of the translation unit; no annotation in user code.	No. Labels are in-source, per-assertion directives.
4.5 Avoid heavy annotation	Yes. Enforcement is a build-level choice; no annotation per line of source.	No. In-source per-operation directives are the design (P3100R8 Section 7.2).
3.3 No lower-level language below	Yes. A trap instruction is as low-level as the response gets.	Yes. Quick-enforce is also a trap.
3.4 Zero-overhead	Yes. Inactive: zero cost. Active: a trap, one instruction, measured at about 0.30% in production (Section 6).	Yes for the non-throwing semantics. Quick-enforce is a trap and matches the profile's cost exactly; ignore adds nothing. The throwing handler is the exception: it requires exception-handling scaffolding around every check, so only that configuration carries overhead the profile does not.
3.5 Manual control	Yes. `[[profiles::suppress(std::core_ub)]]` is explicit, local, and in-source.	Yes. Labels and the assume semantic provide in-source control, at per-assertion granularity.

Table 4b. The approaches measured against the further D&E principles SD-10 builds on.

Principle	`std::core_ub`	P3100R8
Field-tested (D&E 4.2)	Yes. Standardizes the named-guarantee form shipping in eight production systems (Section 6).	No. Its distinctive machinery has no deployment experience (recorded at Croydon).
Useful now (D&E 4.2)	Yes. Implementable today with existing sanitizer and hardening technology.	No. Requires P2900 plus P3400 plus six new clauses; none is implemented.
A facility, not a system (D&E 4.2)	Yes. One attribute, one profile.	No. Six clauses, five semantics, Labels, a handler, and a violation object.
Local inspection (D&E 4.4)	Yes. Enforced or not by the translation unit's first declaration.	No. The semantic, the handler, and the response are each implementation-defined or fixed at link time.
Integrates with existing features (D&E 6.4.4)	Yes. Standard attributes under P3589R2; no new language concept.	No. Redefines "undefined behavior" and adds a novel "implicit assertion" concept.

Across the twelve principles the profile answers yes to all. P3100R8 answers yes to three, all on its non-throwing configurations - no lower-level construct below (3.3), zero-overhead (3.4), and manual control (3.5), where its quick-enforce semantic is itself a trap - and no to the other nine. The reasons in each cell are the designs' own documented properties.

---

6. Deployed Practice

The named-guarantee form (a named set of checks selected per build, with a terminating response) is what production systems ship today. `std::core_ub` standardizes that form. P4306R0 [2] Section 6 assembles the full record with sources; Table 5 summarizes the deployments and adds the column that matters here: whether each matches the profile's design.

Table 5. Production deployments of the form `std::core_ub` standardizes. Here "form" means a named set of checks selected per build with a terminating response; the scope column records what each deployment checks, and the last column records whether that deployment ships in the profile's form.

Implementation	Shipped	Scope	Response	Measured cost	Scale	Matches profile form
libc++ hardening	LLVM 18, 2024	library preconditions	trap	~0.30% (Google)	hundreds of millions of LoC	**Yes**
libstdc++ assertions	GCC 6, 2016	library preconditions	diagnostic, `abort()`	not separately reported	default at `-O0` since GCC 15.1	**Yes**
MSVC STL hardening	VS 2022 17.14, 2025	library preconditions	`__fastfail`	not separately reported	opt-in	**Yes**
WebKit	2024	library preconditions	trap (libc++ extensive)	not separately published	release builds	**Yes**
Firefox	2025	library preconditions	vendor-selected	not separately published	opt macOS default; release pending	**Yes**
Android UBSan	Android 7.0, 2016	core-language arithmetic, bounds	abort	not public	per-component (media, Bluetooth)	**Yes**
Chrome CFI	production	core-language control flow	SIGILL	not public	official builds	**Yes**
Apple `-fbounds-safety`	production	core-language bounds	deterministic trap	not public	millions of LoC of C	**Yes**

Every row terminates on a violation. None constructs a violation object, and none routes through a replaceable handler. What these systems check falls in the scope the table records: some check the runtime-checkable core-language cases catalogued in P3100R8 directly (Android, Chrome, Apple), and the rest harden the standard-library preconditions built on those cases. What they do on a failure is what the profile does: they end the program in place of undefined behavior. The profile standardizes the form the field already runs; Section 8 draws the scope boundary exactly, including the type-and-lifetime cases not yet a production default anywhere.

The one deployment with a published fleet-scale cost figure is Google's. Hardening libc++ across its production services - hundreds of millions of lines of C++ - was measured at an average 0.30% performance overhead, cut the baseline fleet

segmentation-fault rate by roughly 30%, and surfaced more than 1,000 bugs during rollout, with a projected 1,000 to 2,000 prevented each year [22] [23]. Two things about that figure should be stated precisely. It is the cost of standard-library precondition hardening - the bounds and precondition checks on library containers - not a measurement of instrumenting the type-and-lifetime cases that dominate the profile's 58 non-locally-diagnosable entries; those are checked in the instrumented tier (Section 2.5), which costs more, and the profile does not claim its full 77-case guarantee for the price of the 0.30% figure. What the figure does establish is narrower and still load-bearing: a terminating precondition check, deployed at fleet scale, can cost a fraction of a percent, so the locally diagnosable checks a deployment turns on first are cheap at that scale, while the full guarantee is the instrumented tier above them, at instrumentation cost. And the mechanism is the profile's own: on a failed check libc++ "terminates the program with a trap instruction," which its authors identify as "precisely the quick-enforce evaluation semantic" of C++26 Contracts [22]. The terminating response the profile standardizes is thus both deployed and measured for the library-hardening tier, and is the shape the Contracts model already names.

---

7. The Committee's Recorded Direction

The polls in Section 9 ask EWG to record positions it has already taken. This section assembles the record.

The standing document. SD-10 [7], adopted by EWG in December 2024, is the design-principle standard for language evolution. Its Section 4.1 describes a safe-by-default feature with an in-source opt-out, and its Sections 4.4 and 4.5 warn against viral and heavy annotation. P2000R5 [9] Section 5, the Direction Group's direction paper, states the change strategy: "We change the language and standard library by gradually building on previous work or by providing a better alternative to an existing feature."

The Direction Group. P3970R0 [10] (January 2026) designates Profiles as the primary strategy for C++29 safety. Its authors are the full Direction Group.

The poll trail. Thirteen successful polls over four years have supported the Profiles direction and framework. Counts are given as SF/F/N/A/SA (strongly favor / favor / neutral / against / strongly against) where the record preserves the full breakdown, and as for-against totals where only aggregates were recorded:

- SG23, Kona, November 2022: 35-2 and 33-2 (for-against), to pursue the combination of runtime checking, library facilities, and static analysis, and to start from P2687R0 [11].
- EWG, Issaquah, February 2023: 47-2 (for-against), supporting the Profiles direction of P2816R0 [13].
- SG23, St. Louis, June 2024: 12/6/1/3/0, for the attribute syntax of P3447R0 [14].
- SG23, Wroclaw, November 2024: a four-way priority poll of 19/11/6/9 (Profiles / both / neutral / the alternative); 23-1-0 to forward P3081R0 [15] to EWG; 22-2-0 to give more time to the invalidation profile.
- EWG, Sofia, June 2025: 16/14/11/2/0, EWG likes the approach of the P3589R2 [3] framework; 31-2 (for-against), that P3700R0 [16] is correct guidance for adding safety rules.
- SG23, Croydon, March 2026: 20-2 supporting the design principles of P3984R0 [6]; 25-0 to focus on the framework for C++29; 18-0 to volunteer to EWG to drive the work (all for-against).
- SG23, Brno, June 2026: 20/15/4/0/0, to encourage more work on the initialization profile of P4222R2 [5].

The deployment-experience standard, stated in the committee's own voice. At Croydon, Gabriel Dos Reis: "We need real deployment experience, and this is not ready to forward." Timur Doumler has set the same bar for the machinery generally: "real deployment experience across different domains and companies." P3608R0 [12] (Rationale), co-authored by Voutilainen, Wakely, and Dos Reis, applied it in this exact domain: "the standard library hardening is existing practice, and comes with very positive field experience reports."

Taken together, the record holds three positions: SD-10 governs evolution design, deployment experience is the standard for a safety feature, and Profiles is the endorsed direction with a four-year poll trail. The profile proposed here satisfies all three. The polls in Section 9 ask EWG to say so.

---

8. Potential Concerns

Each heading below states a concern in its strongest form; each answer draws only on evidence already presented.

The first concern: the profile has no implementation

True, and stated plainly: `std::core_ub` is specified here, not shipped. Three facts bound the concern. First, the checking each guarded case requires is deployed technology today; the sanitizers and hardened libraries of Section 6 perform checks of exactly these kinds. Second, the checking instrumentation is the same work under either routing: an inserted bounds check or lifetime check serves a profile and an implicit contract assertion alike, so an implementation of the checks is not duplicated effort between the two proposals. Third, the framework the profile is specified on has a public Clang implementation. Applied evenly, the concern weighs the other way: the named-guarantee form the profile standardizes ships across the eight systems of Section 6, while the routing it declines ships nowhere.

The second concern: the deployed systems check library preconditions, not core-language cases

Partly true, and the boundary is worth drawing exactly. Not all of Section 6 is library hardening: core-language undefined behavior is checked in production today by more than the hardened libraries. Apple's `-fbounds-safety` turns out-of-bounds access into a deterministic trap across millions of lines of production C; Android's integer-overflow sanitizer (IntSan) and bounds sanitizer (BoundSan) abort on signed and unsigned overflow and on array-bounds violations across its media stack; and Chrome's control-flow integrity traps on indirect-call type errors in shipping builds (P4306R0 [2] Section 6 assembles these with sources). So the arithmetic, bounds, and indirect-call subsets of the 77 ship as core-language checks, not only as library preconditions. What remains genuinely at the frontier is the type-and-lifetime subset that dominates the 58 instrumented cases, which is not yet a shipped production default anywhere. The profile's response is the one every deployed hardened library gives: it standardizes the guarantee and leaves the checking mechanism to quality of implementation, in tiers (Section 2.5). The claim is not that all 77 checks ship today; it is that the profile's form, response, and per-build activation are the deployed shape, and the enumeration says exactly what an implementation must eventually check.

The third concern: the SD-10 scorecard is scored by the paper's own author

It is, and so the criteria are stated with their sources so a reader can re-score. The principles in Tables 4a and 4b are SD-10 and the D&E principles it builds on, not this paper's invention, and each cell carries its reasoning against the cited section, per the even-handed standard of P2000R5 [9] Section 5.4. A delegate who reads a verdict as unfair can change that one row and see whether the comparison's shape survives; the three rows P3100R8 wins are recorded in Table 4a for exactly that reason.

The fourth concern: "zero foundational wording changes" only relocates the work, it does not remove it

A fair challenge, and the distinction it turns on should be made exactly. The profile does specify what it guards - the guarantee (Section 2.1) and the enumeration of guarded operations (Appendix A) - and a conforming implementation must still perform the checks. None of that is denied. But "foundational wording changes" is a claim about the definitional machinery of the standard, and that is the specific thing the profile does not touch. The six clauses in Table 1 redefine undefined behavior as an implicit contract assertion, add an assume semantic for implicit assertions, and introduce a new "implicit assertion" concept into the core language; those are changes to how the standard defines behavior. The profile introduces no new core-language concept and redefines nothing - not undefined behavior, not `noexcept`. Its specification is additive: it names a set of operations, a guarantee over them, and a response. As for the checking itself, that is quality of implementation and is the same instrumentation work under either routing (the first concern above). So the checking is not "removed" by either proposal; what the profile removes is the six definitional-machinery changes that the Contracts routing adds on top of the checks - and those are removed, not relocated.

The fifth concern: EWG already declined core-language safety profiles at Hagenberg

Stated plainly, because the record is public. At Hagenberg (February 2025) EWG declined to forward the Profiles framework to CWG for C++26 (P3589R1, the prior revision of P3589R2 [3]: 18/16/4/14/20, not consensus) and reached consensus against forwarding P3081R2 [25] core safety profiles for C++26 (10/10/2/25/29). This paper does not read those votes as settling the question it raises, for three reasons. First, timeframe and scope: both were C++26-forwarding votes, taken at the meeting that also resolved to restrict the runtime-checking component of Profiles v1 to standard-library hardening (27/33/9/1/0); `std::core_ub` is proposed for C++29, not for that C++26 v1. Second, design: the profile specified here - a terminating response backed by the deployment record of Section 6 - is a different proposition from the 2025 P3081R2 design those votes addressed. Third, the direction recovered on the C++29 track after Hagenberg, as Section 7 records: Sofia (June 2025) liked the approach of the P3589R2 framework (16/14/11/2/0), and Croydon (March 2026) resolved to focus on the framework for C++29 (25-0) and volunteered to EWG to drive the work (18-0). The Hagenberg votes are evidence that a C++26 core-language profile did not carry then; they are not evidence against continued C++29 work of the kind the committee has since encouraged.

The sixth concern: the full 77-case guarantee costs sanitizer overhead no production default runs today

Fair, and the profile's own Section 2.5 states it: of the 77 cases, 19 are locally diagnosable and cheap at any optimization level, while the remaining 58 require the instrumentation sanitizers provide, and no production default runs all 58. Four facts bound the concern. First, the 19 locally diagnosable cases are deployable now at negligible cost, and they are the tier a deployment turns on first (Section 2.5). Second, the full 77 is the guarantee's definition, not its day-one deployment: the tiers are adoption aids on the path libc++ took from `fast` to `extensive` to `debug`, and a build that runs only the cheaper tier is a diagnostic tool below the profile, not a weaker enforcement of it. Third, the instrumented cost is not a cost of the profile over its alternative, because P3100R8's enforce or quick-enforce on the same 58 cases inserts the same instrumentation; the overhead belongs to the checking, not to the routing. Fourth, a single fixed guarantee keeps the 77-case target the same everywhere, where the alternative - each vendor enforcing its own subset as the meaning of the guarantee - is the per-build semantic variation P2834R1 [24] names as the hazard (Section 2.4), so defining the full set once is what prevents that variation.

The seventh concern: the profile terminates, so it cannot be adopted into working legacy code

The strongest form is that a terminating response crashes code that runs correctly today, so no deployment with legacy code can adopt the profile - the position argued from Bloomberg's experience, that adding checks to working production code requires a log-and-continue response (P3290R4 [19], and the observe semantic of P2900R14 [17]). Five facts bound it. First, the 15 defined-replacement cases (Appendix A.4) do not terminate: they take the profile-defined value, and they include signed overflow to wraparound, the canonical latent condition in working legacy code. Second, `[[profiles::suppress(std::core_ub)]]` is the in-source escape for a region whose correctness is established by other means, and enforcement widens one translation unit at a time (Section 2.7), so adoption is incremental rather than all-or-nothing. Third, the terminating response follows a boundary already drawn in deployment: `bsls_review` [20] logs and continues at the library level, where the post-violation state is defined, while `bsls_assert` [20] terminates where the state is language-undefined - the class this profile guards - and the profile takes the same line (Section 4.3). Fourth, continuation past a language-undefined state is the one response the profile declines, and the case for declining it is set out in the companion P4310R0 [26]. Fifth, the handler still runs: the non-returning handler of Section 2.3 receives the violation and may log or report it before the program ends, so termination does not cost the telemetry.

---

9. Suggested Straw Polls

Three polls follow. Each records a position already in the committee's stated direction (Section 7), so each is straightforward to affirm, and the three read in order.

> *Poll 1. EWG holds that proposals for the runtime checking of core-language undefined behavior should follow the design principles in SD-10.*

SD-10 is EWG's own standing document, adopted December 2024, and Section 2 already provides that a proposal deviating from it should document the tradeoff rationale. Poll 1 records that the standard applies to this domain.

> *Poll 2. EWG holds that proposals for the runtime checking of core-language undefined behavior should be informed by implementation and deployment experience.*

This is the standard already stated by Dos Reis, Doumler, and P3608R0, and consistent with P2000R5's change strategy and the Hagenberg mandate. Poll 2 records that the standard applies here. The named-guarantee form has the experience Table 5 records; both proposals' specifications remain unshipped.

> *Poll 3. EWG supports further work on a standard profile `std::core_ub` that guards the runtime-checkable cases of core-language undefined behavior (as enumerated by P3100R8) under the P3589R2 Profiles framework.*

The profile follows the principles of Poll 1 (Section 5), it standardizes the form with the deployment experience of Poll 2 (Section 6), and Profiles is the direction already endorsed across thirteen polls and the Direction Group's P3970R0 (Section 7). The parenthetical credits P3100R8 in the poll's own text, because the cases it guards are the enumeration of Doumler and Berne.

A delegate who affirms Poll 1 has recorded that SD-10 governs this domain; a delegate who affirms Poll 2 has recorded that deployment experience is the standard. Poll 3 asks for further work on a profile that meets both, in the direction the committee has already chosen. If the three pass, C++ gains a runtime safety profile whose specification is complete, whose form ships today, and whose enumeration the committee already possesses. Whoever designs the response and the replacement behaviors builds on this work next.

---

10. Conclusion

`std::core_ub` guards the runtime-checkable cases of core-language undefined behavior (the 77 cases enumerated by Doumler and Berne in P3100R8) with a single profile under the P3589R2 framework. It provides that coverage with zero foundational changes to the definitional machinery of the standard, where the alternative routing requires six. It follows every principle in SD-10, and it standardizes the named-guarantee form that ships in production across eight systems today. The profile owns its guarantee, its enumeration, and its response, and it leaves the meaning of `noexcept` untouched.

The enumeration belongs to P3100R8, and this profile is built on it. What remains is a design choice on the response to a violation and the replacement behaviors, to be settled in R1 with the committee's guidance. That work builds on this paper next.

---

11. Disclosure

The author provides information and serves at the pleasure of the committee.

Vinnie Falco is the founder of the C++ Alliance, which funds a Clang implementation and a GCC implementation of the Profiles framework; the Clang implementation is public, with regularly released experimental builds that implement the framework attributes and an initial slice of the `std::init` profile.

This paper proposes a profile specification. It does not propose wording; the guarding cases, the response to a violation, and the replacement behaviors are set out here for the profile author to develop into wording. This is a companion to P4297R0 [4] and P4306R0 [2] in the July 2026 mailing, and it works from the published record; where an argument is made in one of those companions, this paper cites it rather than repeating it. It uses machine-assisted drafting.

Acknowledgments

Timur Doumler and Joshua Berne performed the exhaustive enumeration and classification of core-language undefined behavior in P3100R8. Their systematic identification of the runtime-checkable cases, the checking strategies, and the replacement behaviors is the foundation this profile stands on, and Appendix A is their work.

John Lakos's decades of work on Bloomberg's assertion facilities, and the evolution of `bsls_assert` and `bsls_review`, inform the violation-response options in Section 2.3.

Herb Sutter's P3081R0 first demonstrated the application of profiles to core-language undefined behavior, and its deployment-experience framing informs Section 6.

Andrzej Krzemieński contributed to the Contracts design that P3100R8 builds on.

Gabriel Dos Reis designed the Profiles framework of P3589R2 on which this profile is specified.

This paper is indebted to Bjarne Stroustrup, whose design of the Profiles concept, whose D&E principles inform the evaluation in Section 5, whose P3984R0 establishes the authority for a profile to define the meaning of some forms of undefined behavior, and whose decades of advocacy for type-safe C++ created the space in which this work exists. R1 will settle the violation response and the replacement behaviors, and on those choices the author would welcome his direction and the committee's.

References

[1] P3100R8 - "A framework for systematically addressing undefined behaviour in the C++ Standard" (Timur Doumler, Joshua Berne, 2026).

[2] P4306R0 - "Configuring Runtime Checking: Profiles and Implicit Contract Assertions" (Vinnie Falco, Ville Voutilainen, 2026).

[3] P3589R2 - "C++ Profiles: The Framework" (Gabriel Dos Reis, 2025).

[4] P4297R0 - "Severing P3100's Profiles Claim from Its Case-by-Case Review" (Vinnie Falco, Ville Voutilainen, 2026).

[5] P4222R2 - "An initialization profile" (Bjarne Stroustrup, 2026).

[6] P3984R0 - "A type-safety profile" (Bjarne Stroustrup, 2026).

[7] SD-10 - "Language Evolution (EWG) Principles" (EWG chairs, 2024-12-02).

[8] B. Stroustrup, *The Design and Evolution of C++* (Addison-Wesley, 1994).

[9] P2000R5 - "Direction for ISO C++" (Jeff Garland, Paul E. McKenney, Roger Orr, Bjarne Stroustrup, David Vandevoorde, Michael Wong, 2026).

[10] P3970R0 - "Profiles and Safety: a call to action" (David Vandevoorde, Jeff Garland, Paul E. McKenney, Roger Orr, Bjarne Stroustrup, Michael Wong, 2026).

[11] P2687R0 - "Design Alternatives for Type-and-Resource Safe C++" (Bjarne Stroustrup, Gabriel Dos Reis, 2022).

[12] P3608R0 - "Contracts and profiles: what can we reasonably ship in C++26" (Ville Voutilainen, Jonathan Wakely, Gabriel Dos Reis, 2025).

[13] P2816R0 - "Safety Profiles: Type-and-resource Safe Programming in ISO Standard C++" (Bjarne Stroustrup, Gabriel Dos Reis, 2023).

[14] P3447R0 - "Profiles syntax" (Bjarne Stroustrup, 2024).

[15] P3081R0 - "Core safety profiles for C++26" (Herb Sutter, 2024).

[16] P3700R0 - "Principles for C++ safety" (Peter Bindels, 2025).

[17] P2900R14 - "Contracts for C++" (Joshua Berne, Timur Doumler, Andrzej Krzemieński, 2025).

[18] P3400R3 - "Controlling Contract-Assertion Properties" (Joshua Berne, 2026).

[19] P3290R4 - "Integrating Existing Assertions with Contracts" (Joshua Berne, Timur Doumler, John Lakos, 2026).

[20] bsls_assert and bsls_review component documentation (Bloomberg BDE, retrieved 2026).

[21] P4308R0 - "Eight Responses to a Throwing Implicit Contract Assertion" (Vinnie Falco, Ville Voutilainen, 2026).

[22] Practical Security in Production - "Practical Security in Production: Hardening the C++ Standard Library at Massive Scale" (Louis Dionne, Alex Rebert, Max Shavrick, Konstantin Varlamov, ACM Queue Vol. 23 Iss. 5, 2025).

[23] Retrofitting spatial safety to hundreds of millions of lines of C++ - Google Security Blog (Alex Rebert, Kinuko Yasuda, Max Shavrick, 2024-11-15).

[24] P2834R1 - "Semantic Stability Across Contract-Checking Build Modes" (Joshua Berne, John Lakos, 2023).

[25] P3081R2 - "Core safety profiles for C++26" (Herb Sutter, 2025).

[26] P4310R0 - "Hasta la Vista, Undefined Behavior: Why Implicit Contract Violations Should Terminate" (Vinnie Falco, Ville Voutilainen, 2026).

Appendix A: Enumeration of Guarded Operations

The enumeration below is the work of Doumler and Berne, reproduced from P3100R8 Appendix A. Their exhaustive identification of every case of explicit core-language undefined behavior, the classification of each by diagnosability, the checking strategies, and the replacement behaviors are the foundation this profile stands on. The 77 runtime-checkable cases are grouped here by whether a check can be performed locally; the three cases P3100R8 identifies as not runtime-checkable are omitted.

A.1 Locally checkable (19 cases)

No cross-program instrumentation is required; these are checkable at any optimization level.

Identifier	Clause	Checking strategy
`{basic.align.object.alignment}`	[basic.align]/1	Insert alignment check
`{expr.mptr.oper.member.func.null}`	[expr.mptr.oper]/6	Insert null pointer check
`{expr.assign.overlap}`	[expr.assign]/7	Check overlap of the two address ranges
`{class.abstract.pure.virtual}`	[class.abstract]/6	Insert `pre(false)` into the pure-virtual stub
`{expr.expr.eval}`	[expr.pre]/4	Check the value is valid
`{conv.double.out.of.range}`	[conv.double]/2	Check the value is valid
`{conv.fpint.float.not.represented}`	[conv.fpint]/1	Check the value is valid
`{conv.fpint.int.not.represented}`	[conv.fpint]/2	Check the value is valid
`{expr.static.cast.enum.outside.range}`	[expr.static.cast]/9	Check the value is valid
`{expr.static.cast.fp.outside.range}`	[expr.static.cast]/10	Check the value is valid
`{expr.mul.div.by.zero}`	[expr.mul]/4	Check the divisor is nonzero
`{expr.mul.representable.type.result}`	[expr.mul]/4	Check the value is valid
`{expr.shift.neg.and.width}`	[expr.shift]/1	Check the right operand is valid
`{intro.execution.unsequenced.modification}`	[conv.rank]/10	Check unsequenced read and write refer to the same address
`{stmt.return.flow.off}`	[stmt.return]/4	`contract_assert(false)` at end of function body
`{dcl.attr.noreturn.eventually.returns}`	[dcl.attr.noreturn]/2	Insert `post(false)`
`{basic.stc.alloc.dealloc.throw}`	[basic.stc.dynamic.deallocation]/4	Assertion in a catch handler
`{expr.new.non.allocating.null}`	[expr.new]/22	Insert `post(r: r)`
`{stmt.return.coroutine.flow.off}`	[stmt.return.coroutine]/3	`contract_assert(false)` at end if no `return_void`

A.2 Locally checkable only in special cases (6 cases)

Checkable locally under the stated condition; otherwise they require instrumentation.

Identifier	Clause	Condition	Checking strategy
`{expr.add.out.of.bounds}`	[expr.add]/4	array bound statically known	Track pointer provenance, insert bounds check
`{expr.add.sub.diff.pointers}`	[expr.add]/4	array bound statically known	Track pointer provenance, insert bounds check
`{conv.ptr.virtual.base}`	[conv.ptr]/3	null pointer case	Track lifetime and type, or ctor-dtor state; null check
`{expr.dynamic.cast.pointer.lifetime}`	[expr.dynamic.cast]/7	null pointer case	Track lifetime and type, or ctor-dtor state; null check
`{expr.static.cast.downcast.wrong.derived.type}`	[expr.static.cast]/11	null pointer case	Track lifetime and type, or ctor-dtor state; null check
`{expr.unary.dereference}`	[expr.unary.op]/1	null pointer case	Track lifetime and type, and function address; null check

A.3 Not locally checkable (52 cases)

These require whole-program instrumentation of the kind sanitizers provide. Grouped by category for reference.

Initialization (1)

Identifier	Clause	Checking strategy
`{basic.indet.value}`	[basic.indet]/2	Track whether storage has been initialised

Bounds (3)

Identifier	Clause	Checking strategy
`{basic.stc.alloc.zero.dereference}`	[basic.stc.dynamic.allocation]/2	Track pointer provenance, insert bounds check
`{expr.delete.mismatch}`	[expr.delete]/2	Track pointer provenance, insert bounds check
`{expr.delete.array.mismatch}`	[expr.delete]/2	Track pointer provenance, insert bounds check

Type and Lifetime, object lifetime and type (18)

Identifier	Clause	Checking strategy
{intro.object.implicit.create}	[intro.object]/11	Track whether storage can hold implicit-lifetime objects
{intro.object.implicit.pointer}	[intro.object]/11	Track whether storage can hold implicit-lifetime objects
{lifetime.outside.pointer.delete}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.pointer.member}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.pointer.virtual}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.pointer.dynamic.cast}	[basic.life]/7	Track lifetime and type of storage
{lifetime.outside.glvalue.access}	[basic.life]/8	Track lifetime and type of storage
{lifetime.outside.glvalue.member}	[basic.life]/8	Track lifetime and type of storage
{lifetime.outside.glvalue.virtual}	[basic.life]/8	Track lifetime and type of storage
{lifetime.outside.glvalue.dynamic.cast}	[basic.life]/8	Track lifetime and type of storage
{original.type.implicit.destructor}	[basic.life]/11	Track lifetime and type of storage
{expr.basic.lvalue.strict.aliasing.violation}	[basic.lval]/11.3	Track lifetime and type of storage
{expr.basic.lvalue.union.initialization}	[basic.lval]/11.3	Track lifetime and type of storage
{expr.ref.member.not.similar}	[expr.ref]/9	Track lifetime and type of storage
{expr.dynamic.cast.glvalue.lifetime}	[expr.dynamic.cast]/7	Track lifetime and type, or ctor-dtor state
{expr.static.cast.base.class}	[expr.static.cast]/2	Track lifetime and type of storage
{expr.add.not.similar}	[expr.add]/6	Track whether storage holds an object of the correct type
{class.dtor.no.longer.exists}	[class.dtor]/18	Track lifetime and type of storage

Type and Lifetime, allocation, const, and volatile (6)

Identifier	Clause	Checking strategy
`{creating.within.const.complete.obj}`	[basic.life]/12	Track whether storage holds a const object
`{basic.compound.invalid.pointer}`	[basic.compound]/4	Track whether storage has been allocated and freed
`{expr.type.reference.lifetime}`	[expr.type]/1	Track whether storage has been allocated and freed
`{conv.lval.valid.representation}`	[conv.lval]/3.4	Track lifetime and type of storage
`{dcl.type.cv.modify.const.obj}`	[dcl.type.cv]/4	Track whether storage holds a const object
`{dcl.type.cv.access.volatile}`	[dcl.type.cv]/5	Track whether storage holds a volatile object

Type and Lifetime, function, member-pointer, and reference types (9)

Identifier	Clause	Checking strategy
`{conv.member.missing.member}`	[conv.mem]/2	Track which type the pointer-to-member originated from
`{expr.call.different.type}`	[expr.call]/5	Track function type by address
`{expr.static.cast.does.not.contain.orignal.member}`	[expr.static.cast]/12	Track which type the pointer-to-member originated from
`{expr.delete.dynamic.type.differ}`	[expr.delete]/3	Track dynamic type of non-polymorphic objects
`{expr.delete.dynamic.array.dynamic.type.differ}`	[expr.delete]/3	Track dynamic type of non-polymorphic objects
`{expr.mptr.oper.not.contain.member}`	[expr.mptr.oper]/4	Track pointer-to-member origin and dynamic type
`{dcl.ref.incompatible.function}`	[dcl.ref]/6	Track function types by address
`{dcl.ref.incompatible.type}`	[dcl.ref]/6	Track whether storage holds an object of the correct type
`{dcl.ref.uninitialized.reference}`	[dcl.ref]/6	Track whether references have been initialised

Type and Lifetime, construction and destruction state (9)

Identifier	Clause	Checking strategy
{class.base.init.mem.fun}	[class.base.init]/16	Track whether objects are being constructed or destroyed
{class.cdtor.before.ctor}	[class.cdtor]/1	Track whether objects are being constructed or destroyed
{class.cdtor.after.dtor}	[class.cdtor]/1	Track whether objects are being constructed or destroyed
{class.cdtor.convert.pointer}	[class.cdtor]/3	Track whether objects are being constructed or destroyed
{class.cdtor.form.pointer}	[class.cdtor]/3	Track whether objects are being constructed or destroyed
{class.cdtor.virtual.not.x}	[class.cdtor]/4	Track whether objects are being constructed or destroyed
{class.cdtor.typeid}	[class.cdtor]/5	Track whether objects are being constructed or destroyed
{class.cdtor.dynamic.cast}	[class.cdtor]/6	Track whether objects are being constructed or destroyed
{except.handle.handler.ctor.dtor}	[except.handle]/11	Track whether objects are being constructed or destroyed

Threading (1)

Identifier	Clause	Checking strategy
{intro.races.data}	[intro.races]/17	Track inter-thread access and synchronization (TSan-style; a subset only)

Control Flow (3)

Identifier	Clause	Checking strategy
{basic.start.main.exit.during.destruction}	[basic.start.main]/4	Track whether static or thread-local objects are being destroyed
{basic.start.term.use.after.destruction}	[basic.start.term]/4	Track the lifetime of static objects
{stmt.dcl.local.static.init.recursive}	[stmt.dcl]/3	Recursion counter in the static and thread-local init guard

Coroutines (2)

Identifier	Clause	Checking strategy
`{dcl.fct.def.coroutine.resume.not.suspended}`	[dcl.fct.def.coroutine]/9	Track the suspension state of each coroutine handle
`{dcl.fct.def.coroutine.destroy.not.suspended}`	[dcl.fct.def.coroutine]/12	Track the suspension state of each coroutine handle

A.4 Cases with well-defined replacement behavior (15 cases)

The other 62 guarded cases have no replacement: a violation ends the program. For these 15 the profile adopts the defined behavior below in place of termination, fixed for every conforming implementation (12 unconditional, 3 for built-in types only; for those 3 the replacement applies to built-in types and the operation terminates otherwise).

Identifier	Replacement behavior
{basic.indet.value}	Erroneous value (built-in types only)
{conv.lval.valid.representation}	Coerce invalid representations to erroneous values
{expr.expr.eval}	Coerce to erroneous value
{conv.double.out.of.range}	Coerce to erroneous value
{conv.fpint.float.not.represented}	Coerce to erroneous value
{conv.fpint.int.not.represented}	Coerce to erroneous value
{expr.static.cast.enum.outside.range}	Coerce to erroneous value
{expr.static.cast.fp.outside.range}	Coerce to erroneous value
{expr.mul.div.by.zero}	Coerce to erroneous value
{expr.mul.representable.type.result}	Coerce to erroneous value
{expr.shift.neg.and.width}	Coerce to erroneous value
{intro.races.data}	Make primitive memory accesses implicitly atomic
{intro.execution.unsequenced.modification}	Sequence the operations in some unspecified order
{stmt.return.flow.off}	Return erroneous value (built-in return types only)
{stmt.return.coroutine.flow.off}	Return erroneous value (built-in return types only)