Hasta la Vista, Undefined Behavior: Why std::core_ub Should Terminate by Default

Document Number:	P4310R1
Date:	2026-08-02
Intent:	Inform
Audience:	EWG, SG23

Reply-to:	Vinnie Falco vinnie.falco@gmail.com Ville Voutilainen ville.voutilainen@gmail.com

Table of Contents

Abstract

`std::core_ub` (P4317R1) guards the runtime-checkable cases of core-language undefined behaviour and, when enforced, guarantees the check runs. It leaves one question open: after a guarded violation is detected, does the program continue or terminate? This paper answers terminate. Whether contracts are the right substrate for these checks at all is argued against them in P4332R0; this paper takes a check that runs and argues only the response.

The evidence is the deployment record: every hardened implementation surveyed that detects such a violation in production terminates or traps, and none defaults to continuation. Two findings confirm it - continuation runs against P3878R1, the adjacent case C++26 already decided, and it executes on the corrupted state the security literature treats as the more dangerous failure. Terminating keeps the handler invocation, so it loses no telemetry. The finding is narrow: a continuing response, if kept at all, is an explicit non-portable opt-in, and for the class that continues into a defined value such as wrapped overflow the question is left open. The `noexcept` and throwing-cost analysis lives in P4308R1. The paper makes no request.

Revision History

R1: August 2026

- Rewritten as a single-claim argument: the deployment record (Table 1) plus the two load-bearing findings, with the survey detail, the terminology glossary, and the corroborating evidence deferred to the companion papers.
- Reframed onto the `std::core_ub` profile (P4317R1): the response is argued as the profile's response to a detected core-language violation rather than as a restriction on P3100R8's implicit contract assertions.
- Ceded the enforcement axis to P4332R0 and scoped this paper to the response axis alone.
- Handed the `noexcept` and throwing-response mechanics to P4308R1.

R0: July 2026

- Initial version.

---

1. Introduction

P4317R1 [22] proposes `std::core_ub`, a profile under the P3589R2 [21] framework that guards the runtime-checkable cases of core-language undefined behaviour and, when enforced over a region, guarantees the check is performed. It leaves one question open (P4317R1 Section 2.3): after a guarded operation's precondition is detected as violated, does execution continue past the violation or does the program terminate?

This paper answers it: terminate. It does not reopen the prior question - whether the C++26 Contracts machinery is the right substrate for these checks, given that a contract assertion may be compiled `ignore` and so may never run - which P4332R0 [23] argues against. Taking a check guaranteed to run, as the profile's enforcement guarantees, this paper argues only the response. P3878R1 [16], adopted into C++26, already settled the same response question for standard-library hardening; P4306R1 [3] and P4297R1 [4] cover the configuration and ownership questions, and P4308R1 [24] the `noexcept` and throwing-response space.

Two terms recur. The hook is the handler invocation: a violation handler is called and logs the violation. The continuation is what a continuing response adds on top of it: after the handler returns, execution proceeds past the violation. Every response discussed here preserves the hook; only the continuation is contested, and only for the class whose continuation is into a state the language does not define. The class that continues into a defined value (Section 4) is left open, and C++26 as ratified is untouched.

---

2. What ships, terminates

What deployed hardening does on a detected core-language violation is uniform; P4317R1 [22] Section 6 assembles the full record with sources, and Table 1 summarizes it. The sampled population is every implementation the authors could identify that detects such a violation in production, in its default configuration.

Table 1. Response to a detected violation in deployed hardened implementations, in their production configurations. Every entry terminates or traps. None makes continuation its production default.

Implementation	Response on violation	Source
libc++ `fast` / `extensive`	trap (`quick-enforce`)	[5]
libc++ `debug`	log + terminate (`enforce`)	[5]
libstdc++ `_GLIBCXX_ASSERTIONS`	`abort`	[6]
MSVC STL `_MSVC_STL_HARDENING`	`__fastfail`	[7]
glibc `_FORTIFY_SOURCE` 1/2/3	`SIGABRT`	[8]
Google server fleet (libc++)	trap (`quick-enforce`); ~0.3% avg overhead	[9]
Android IntSan	`abort` (log mode is testing only)	[10]
UBSan (production guidance)	trap (`-fsanitize-trap`); recover is "meant for testing purposes"	[11]
Abseil `CHECK`, Folly `XCHECK`, WebKit `RELEASE_ASSERT`	terminate	[12]

Every entry terminates or traps, and none makes continuation its production default; the continue modes that ship - libc++ `observe`, Bloomberg `bsls_review`, UBSan's recover mode - are documented as adoption or testing aids, not standing configuration. Termination is not lossy on telemetry: the entries that log before terminating do exactly what a terminating handler does, and a sanitizer records the same violation site and kind with no handler in the program at all (Android IntSan [10], UBSan [11]).

The one surveyed facility with a replaceable handler that can continue is Bloomberg's `bsls_review`, and it does so at the library level, where a violated precondition still leaves the surrounding operations a defined meaning; its companion `bsls_assert` terminates where the state is language-undefined. Bloomberg's own header calls the continued state "undefined" [13], so the deployed line is the one this finding reaches: terminate by default, continue only as a bounded library-level adoption aid.

The availability-first domains the survey excludes are where continuation looks strongest; Stroustrup's P2698R0 [14] calls unconditional termination "a serious problem" for systems not permitted to stop. Met on their own terms they reach the same place for the undefined-continuation subset: the canonical architecture terminates the faulty unit and recovers from a

known-good state rather than executing past the fault, and the hook still logs before the unit stops.

---

3. Why continuing is the wrong default

Two findings confirm the default beyond the deployment record.

First, the committee has already decided the adjacent case. P3878R1 [16], adopted into C++26, established that a standard-library hardened precondition may not be evaluated with a non-terminating semantic, because continuing past such a check "can result in violations of hardened preconditions being undefined behaviour, rather than guaranteed to be diagnosed, which defeats the purpose of using a hardened implementation." For a core-language check whose continuation is likewise undefined - a detected null dereference or out-of-bounds access - the same reasoning applies one level down. Its lead author is a co-author here; the decision was the whole committee's, and the argument stands on the deployment record even if the precedent is set aside.

Second, continuing executes user code on a state the language does not define. Doumler and Berne write in P3097R2 [17] that once a program "is found to be in a possibly corrupted state, executing any user-defined code could result in a vulnerability." That danger is why this paper argues for terminate as the default. The security literature draws the same line for any tool: CERT ERR56-CPP [18] holds that "a violated invariant leaves the program in a state where graceful continued execution is likely to introduce security vulnerabilities." The hazard is a property of the undefined state these checks produce, not of any security purpose, so it transfers whether the tool was built for correctness or security.

A continuing response under Contracts-based undefined-behaviour handling also carries an exception-handling cost. The libc++ hardening implementers - the authors of P3191R0 [15] - require "no exception-handling code being generated around contract predicates," and decline to pay that cost in production. In its throwing form, a continuing response also changes what the `noexcept` operator guarantees. Both are analysed in P4308R1 [24]; a terminating response avoids them because nothing escapes.

---

4. Two carve-outs, and the shape of the response

The defined-replacement class. Not every core-language check continues into undefined behaviour: for cases such as signed-integer overflow a defined replacement value can be specified, and P4317R1 [22] Section 2.4 fixes that meaning for the 15 such cases rather than terminating. There continuation is into a defined value, the corrupted-state objection does not apply, and the question is left open. Even there no deployment logs and continues by default - the field's response to signed overflow is `-fwrapv` (silent) or `-ftrapv` (terminate).

If continuation is allowed at all for cases that continue into an undefined state, deployed practice treats it the same way: the user must turn it on explicitly, it is not the production default, and it is meant only for a short period while checks are being adopted. libc++ documents its `observe` semantic in exactly those terms: "Continuing execution after a hardening check fails results in undefined behavior; the `observe` semantic is meant to make adopting hardening easier but should not be used outside of the adoption period." [5] Under libc++ hardening, a terminating response already covers that adoption need. The

failure can be logged before the program stops, so a team can find failures without running past them.

The terminating default is not this paper's alone. Berne and Lakos recommend in P3558R1 [19] "a default evaluation semantic, when nothing else is specified, of `enforce` for all core-language preconditions." Reusing the C++26 `enforce` semantic and the existing rule that an escaping exception at a non-throwing boundary terminates, the response adds no new semantic and leaves `noexcept` unchanged.

---

5. The substrate and ownership questions are separate, and subordinate

The finding is stated as the profile's response, but it is independent of two questions this paper does not take up. Whether these checks belong on the Contracts substrate at all is P4332R0 [23]'s, answered against it; who configures the response, through the Contracts facility (P3400R3 [20]) or the Profiles framework (P3589R2 [21]), is the ownership question of P4297R1 [4] and P4306R1 [3]. Arguing the response on the profile P4317R1 [22] specifies, where enforcement guarantees the check runs, avoids the "check that may not run" hazard entirely. And the finding holds subordinately even if, against P4332R0, the committee routes the response through the contract-violation handler: for the undefined class it should still terminate, for the reasons above, and the hook that precedes termination preserves the deployer's ability to observe the violation.

---

6. Conclusion

Of the responses to a detected core-language violation, the terminating one - invoke the handler, log, terminate - is the production default across every hardened implementation surveyed, and the continuing response is the default in none. It runs against P3878R1 [16] and executes on a state the language does not define. `std::core_ub` should default to it, reusing the C++26 `enforce` semantic with no new semantic and no change to `noexcept`. A continuing response for the undefined class, if it exists at all, belongs as an explicit non-portable opt-in; for the defined-replacement class the question is left open. The record is placed here for the committee's use; the paper makes no request.

---

7. Disclosure

The authors provide information and serve at the pleasure of the committee.

Vinnie Falco founded the C++ Alliance, which maintains a Clang fork for Profiles work, and co-authors P4317R1 [22] and P4332R0 [23], the profile and substrate findings this paper builds on; he prefers the family of responses in which no exception escapes a checked core-language operation. Ville Voutilainen co-authored the C++26 Contracts facility (P2900R14 [1]) and led P3878R1 [16], the adopted decision reused here. The reader should weigh what follows accordingly.

The intent is `info`: the paper argues a position and requests no poll, and it changes nothing in ratified C++26. In the August 2026 mailing it is one of a set - P4332R0 [23] on the substrate, P4317R1 [22] the profile, P4308R1 [24] the throwing-response space, and P4306R1 [3] and P4297R1 [4] the configuration and ownership questions - and it cross-references those rather than repeating them. The profile's checks are not yet implemented (P4317R1 Section 8), so the paper reasons from deployed

analogues. It was prepared with the assistance of generative tools; the authors are responsible for its content.

Acknowledgements

Timur Doumler and Joshua Berne, whose enumeration and classification of core-language undefined behaviour in P3100R8 [2] made these questions precise, and the authors of P2900R14 [1]. Any errors are the authors' own.

References

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

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

[3] P4306R1 - "Configuring Runtime Checking: Profiles and Implicit Contract Assertions" (Vinnie Falco, Ville Voutilainen, 2026).

[4] P4297R1 - "Severing P3100's Profiles Claim from Its Case-by-Case Review" (Vinnie Falco, Ville Voutilainen, 2026).

[5] libc++ Hardening Modes - "Hardening Modes" (LLVM Project, 2025).

[6] Using libstdc++ Macros - "The GNU C++ Library Manual: Macros" (GNU Project, 2025).

[7] MSVC STL Hardening - "C++ conformance improvements in Visual Studio" (Microsoft, 2025).

[8] Source Fortification - "The GNU C Library: Source Fortification" (GNU Project, 2025).

[9] Practical Security in Production - "Practical Security in Production: Hardening the C++ Standard Library at Massive Scale" (Louis Dionne, Alexander Rebert, Max Shavrick, Konstantin Varlamov, 2025).

[10] Android UBSan - "UndefinedBehaviorSanitizer" (Android Open Source Project, 2025).

[11] Clang UBSan - "UndefinedBehaviorSanitizer" (LLVM Project, 2025).

[12] P3911R2 - "Make Contracts Reliably Non-Ignorable" (Darius Neațu, Andrei Alexandrescu, Lucian Radu Teodorescu, Radu Nichita, Herb Sutter, 2026).

[13] bsls_review - "bsls_review: Provide assertion macros to safely identify contract violations" (Bloomberg BDE, 2019).

[14] P2698R0 - "Unconditional termination is a serious problem" (Bjarne Stroustrup, 2022).

[15] P3191R0 - "Feedback on the scalability of contract violation handlers in P2900" (Louis Dionne, Yeoul Na, Konstantin Varlamov, 2024).

[16] P3878R1 - "Standard library hardening should not use the observe semantic" (Ville Voutilainen, Jonathan Wakely, John Spicer, Stephan T. Lavavej, 2025).

[17] P3097R2 - "Contracts for C++: Virtual functions" (Timur Doumler, Joshua Berne, 2026).

[18] ERR56-CPP - "ERR56-CPP. Guarantee exception safety" (SEI CERT C++ Coding Standard, 2023).

[19] P3558R1 - "Prevent Undefined Behavior By Default" (Joshua Berne, John Lakos, 2025).

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

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

[22] P4317R1 - "A Profile for Runtime-Checkable Core-Language Undefined Behavior: std::core_ub" (Vinnie Falco, 2026).

[23] P4332R0 - "Contracts are inappropriate for undefined behavior checks" (John Spicer, Vinnie Falco, Jose Daniel Garcia Sanchez, Bjarne Stroustrup, Ville Voutilainen, 2026).

[24] P4308R1 - "Eight Responses to a Throwing Implicit Contract Assertion" (Vinnie Falco, Ville Voutilainen, 2026).