Hasta la Vista, Undefined Behavior: Why Implicit Contract Violations Should Terminate

Document Number:	P4310R0
Date:	2026-07-13
Intent:	Inform
Audience:	EWG

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

Table of Contents

Abstract

For a detected core-language violation, the terminating response - invoke the handler, then terminate - is the default the evidence supports.

P3100R8 would treat the runtime-checkable cases of core-language undefined behaviour as implicit contract assertions evaluated through the C++26 violation handler, leaving open whether execution continues past a detected violation or the program terminates. Separating the handler invocation, preserved by every response and carrying the telemetry a deployment needs, from the contested continue-past-violation response isolates the only disputed part and shows a terminating response loses no telemetry. No compiler yet implements these assertions with any semantic, so the comparison reasons from deployed analogues. Termination or trapping is the steady-state production default of every hardened implementation surveyed; the continue modes that ship, such as UBSan's recover mode and Bloomberg's log-and-continue facility, are documented as adoption aids, the latter confined to library-level checks rather than core-language undefined behaviour; even availability-first systems reach availability by terminating the faulty unit and recovering rather than executing past the fault. Continuation also carries an exception-handling cost the reference implementers decline to incur, runs against P3878R1, the decision C++26 already adopted for the adjacent library-hardening case, and executes on the undefined, possibly corrupted state the security literature treats as the more dangerous failure. Reusing the C++26 enforce semantic and the existing rule that a handler throw at a non-throwing boundary terminates, the terminating response adds no new semantic, leaves the meaning of noexcept unchanged, and keeps the deployer's build-time choice among the remaining semantics. This default holds across both classes of check, while the narrower finding, that a continuing semantic should not be a portable guarantee every implementation carries, is scoped to the class whose continuation is undefined and leaves open the class that continues into a defined result such as a wrapped signed overflow. Where a continuing response is retained, it takes the shape the deployed precedents share: an explicit, non-portable opt-in bounded to an adoption period rather than a default. Because the finding concerns the response and not its configuration, it holds whether the Contracts facility or the Profiles framework owns the selection, and the paper places the record for the committee's use without a request.

Revision History

R0: July 2026

- Initial version.

---

1. Introduction

C++26 Contracts (P2900R14 [1]) define the evaluation semantics and the single violation handler this paper takes as fixed. P3100R8 [2] proposes to extend that machinery to the runtime-checkable cases of core-language undefined behaviour, and P3878R1, adopted into C++26, already settled the parallel question for standard-library hardening. Two companions cover the neighbouring ground: P4306R0 [3] compares the configuration-ownership models, and P4297R0 [4] asks EWG to decide the ownership relationship. None of them resolves what this paper takes up: after the violation handler runs on a detected core-language violation, does execution continue past the violation or does the program terminate?

The contributions are:

1. A separation of the `observe` semantic into two independent parts, the handler invocation (the hook) and the continue-past-violation response (the continuation), showing that the hook and every telemetry need it serves survive a terminating response unchanged (Section 2).
2. A survey of deployed hardened implementations, finding that every one terminates or traps on a detected core-language violation and none makes continuation its production default (Section 3).
3. Two further findings against a continuing default: it carries an exception-handling cost the reference implementers decline to incur, and it runs against P3878R1, the decision C++26 already adopted for the adjacent case (Section 4).
4. A terminating response expressed by reusing the C++26 `enforce` semantic and the existing termination rule, adding no new semantic and leaving the meaning of `noexcept` unchanged (Section 6).
5. The shape a continuing response takes if the committee retains one: an opt-in, non-portable facility bounded to an adoption period (Section 7).
6. A demonstration that the finding is independent of the configuration question, holding whether the Contracts facility or the Profiles framework owns the selection (Section 9).

The analysis rests on three assumptions, each stated where it is used and gathered here for the reader who reads only the surface:

- No compiler yet implements implicit contract assertions with any semantic, so the comparison reasons from deployed analogues rather than from a conforming implementation.
- The strong finding covers the class of core-language checks whose continuation is undefined. The class whose continuation is into a defined result (Section 5) is treated separately, and the continuation question there is left open.
- C++26 as ratified is fixed. What remains open belongs to the C++29 work on implicit assertions.

A note on method. Because no conforming implementation exists, every comparison in this paper is to a deployed analogue in an adjacent domain: library hardening, production sanitizers, security-critical and fault-tolerant systems. The inference these analogues license rests on one principle: the hazard of continuing past a detected violation attaches to the program state, not to the purpose of the tool that detected it. When a check fires on core-language undefined behaviour, the program is in a state the language does not define - the same undefined, possibly corrupted state the hardened libraries, sanitizers, and security mechanisms faced when they chose to terminate. The analogues transfer to the extent that state does, and no further. Where continuation is into a defined result (Section 5), the state is defined and the transfer does not apply; the paper marks that boundary wherever it uses an analogue.

The recurring terms carry one meaning throughout:

Term	Meaning
hook	The handler invocation: the contract-violation handler is called and logs the violation. Preserved by every response discussed here.
continuation	The continue-past-violation response: after the handler returns, execution proceeds past the violation.
`ignore`	Evaluation semantic under which the assertion has no effect.
`observe`	Evaluation semantic under which the handler is invoked and, on a normal return, execution continues.
`enforce`	Evaluation semantic under which the handler is invoked and the program is then contract-terminated.
`quick-enforce`	Evaluation semantic under which the program is contract-terminated without invoking the handler.
`assume`	The fifth semantic P3100R8 adds, preserving today's undefined behaviour as an escape hatch.
terminating response	Invoke the handler for its telemetry, then terminate: the `enforce` semantic together with the rule that a handler throw at a non-throwing boundary terminates.

---

2. Two questions inside the one word 'observe'

Because the stated semantic treats two decisions as one, this section separates the parts of the `observe` semantic. A reader who knows the C++26 Contracts model can skip to the last paragraph.

P3100R8 [2] ("A framework for systematically addressing undefined behaviour in the C++ Standard") proposes to guard the 77 runtime-checkable cases of core-language undefined behaviour with implicit contract assertions, checks the language itself inserts at each point of undefined behaviour, configured through the same evaluation semantics C++26 Contracts provide for explicit assertions.

C++26 Contracts (P2900R14 [1], `[basic.contract.eval]`) define four evaluation semantics for a contract assertion. Under `ignore`, the assertion has no effect. Under `observe`, the contract-violation handler is invoked and, if it returns normally, control continues past the point of evaluation. Under `enforce`, the handler is invoked and the program is then contract-terminated. Under `quick-enforce`, the program is contract-terminated without invoking the handler. The handler is a single, program-wide function, `::handle_contract_violation`, and P3100R8 [2] Section 5.6 proposes to keep it single for implicit assertions rather than introducing a second one.

P3100R8 [2] proposes to respecify the runtime-checkable cases of core-language undefined behaviour as implicit contract assertions evaluated with five semantics: the four from C++26 plus a fifth, `assume`, which preserves today's undefined behaviour as an escape hatch. That extension raises a question: on a detected core-language violation, such as a signed-integer overflow or an out-of-bounds access, what happens after the check fails?

The word `observe` bundles two things that can be separated, the first being the handler invocation (termed "hook" henceforth): the handler is invoked, and it logs the violation, giving a deployment one place to record and report it. The second is the continue-past-violation response (termed "continuation" henceforth): after the handler returns, execution proceeds past the violation. While the hook is uncontested and is preserved by every response this paper discusses, `enforce` included, the continuation is the contested part. The single difference between `enforce` and `observe` is what happens on a normal return from the handler: `enforce` terminates, `observe` continues. What follows credits the hook and examines only the continuation.

Scope: this paper concerns implicit assertions on core-language undefined behaviour, where a detected violation means the program has already entered a state the language does not define. Explicit, author-written contract assertions, where a precondition can encode a recoverable condition and continuation can be meaningful, are outside this scope. Whether core-language checks are best specified as implicit contract assertions or as profile-governed checks is the architecture question P4297R0 [4] addresses. The terminology here follows P3100R8, because the response question arises within its proposal.

The terminating response preserves the hook. Under `enforce`, the handler is invoked and logs the violation before the program terminates. Every telemetry need the handler serves - recording the violation site, its kind, the predicate text - survives the terminating response unchanged. Recording a violation does not depend on the contract-violation handler at all: a sanitizer logs the same violation site and kind with no contract handler in the program, as Android IntSan's log mode [10] and UBSan's diagnostics [11] do through the tool rather than through a handler. The handler is one capability that can carry the telemetry, not the mechanism the telemetry requires. What `enforce` removes is the continuation alone: execution past a state the language does not define. The deployer's ability to observe a violation is not at stake, only the program's ability to keep running after one is.

---

3. What ships, terminates

This section reports what deployed hardening does on a detected core-language violation. The pattern is uniform.

Up front, the survey states its population and selection rule. The population is the implementations that detect a core-language violation in production, and the selection rule is every such implementation the authors could identify, recorded in its default or production configuration. That frame deliberately excludes the availability-first domains - long-running services and fault-tolerant systems that might prefer bounded degradation to a hard stop - which are not sampled in Table 1 and are engaged on their own terms in Section 3.1. Table 1 is therefore evidence about the sampled population.

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]

The survey covers standard-library hardening modes, production sanitizer configurations, and critical-assertion facilities. Table 1 records the default response; the non-default continue modes are examined in Section 7.

Every surveyed implementation that detected a core-language violation made termination its steady-state production default. The continue modes that ship - libc++ `observe`, Bloomberg `bsls_review`, UBSan's recover mode - are documented as adoption or testing aids and are used during rollout rather than as the standing production configuration; none is a steady-state default. The absence of a conforming implementation of implicit contract assertions is symmetrical - no compiler has one with any semantic. The deployment analogues are not symmetrical: termination is the steady-state production default across the surveyed implementations, and continuation is the steady-state default in none of them. The precedent is asymmetrical.

The implementations that log before terminating - libc++ debug mode, glibc `_FORTIFY_SOURCE`, Android IntSan in its production configuration - perform exactly the operation `enforce` specifies: invoke a reporting facility, then terminate. The handler is present in every entry that reports before it stops.

A limit of the table belongs with it. The standard-library and sanitizer entries trap or abort directly and expose no user-replaceable handler that could continue, so the choice between a continuing handler and a terminating one does not arise for them. The table therefore shows what handler-less hardening does, not that a handler-capable facility weighed continuation and rejected it. The one surveyed facility with a replaceable handler that can continue is Bloomberg's `bsls_review`, and it applies that continuation at the library level during adoption.

Bloomberg's own contract-checking machinery pairs a terminating default with a documented, bounded continue mode. Its `bsls_assert` facility terminates by default: its default handler aborts rather than returning, which is the terminating response described here. Its companion `bsls_review` logs and continues, and Bloomberg documents it as the way to add checks to working production code without stopping it: `bsls_review` lets a deployment "increase the number of precondition checks used to catch such bugs without negatively impacting the existing behavior of the software," through a default handler that "logs that a failure has occurred and then allows processing to continue." [13] The review mode in `bsls_assert` is documented in the same terms, as a way to test in production "without terminating the application," and as "an interim step towards lowering the assertion level threshold for an existing application." [37] The availability of this log-and-continue response is something Bloomberg's deployment relies on, and the finding here does not contest that. It contests the continuing response for the core-language-undefined class and only as a default; Section 7 gives the adoption need its own opt-in facility.

Where `bsls_review` continues, it operates at the library level, on a precondition whose violation breaks an invariant the library relies on. Bloomberg's own header calls the resulting state "undefined" and, for this adoption use, continues past it rather than aborting. [13] The distinction the finding here rests on - between a violated library precondition, where the language still assigns the surrounding operations a meaning, and core-language undefined behaviour one level down, where the language assigns the continuation no meaning at all - is this paper's analytical line, not one Bloomberg's documentation draws, and it turns on where the check sits rather than on a guarantee that every continued `bsls_review` state stays defined. What the public record supplies is the shape: a terminating default in `bsls_assert`, and a continue mode `bsls_review` documents as an interim adoption aid rather than a steady-state default. That shape is the one the finding reaches - terminate by default, continue only as a bounded opt-in - and the commit history shows the same default applied to a specific component, changing a check that could "continue past that point" into one that terminates because the "execution path is library undefined behavior and the program would be out of contract anyway." [14]

The deployment record therefore establishes one fact: on a detected core-language violation, the terminating response is the steady-state production default.

3.1 The availability-first case

The population Table 1 excludes is where the case for continuation is strongest, and it deserves its strongest form. Stroustrup's P2698R0 [15] states it: unconditional termination is "a serious problem" for the systems that are not permitted to stop - long-running services, and the fault-tolerant and safety-critical domains where a crash is itself the failure. If any population would rationally continue past a violation it is this one, and it is exactly the population the survey does not sample.

Met on its own terms, though, the availability-first domain reaches the same place for the undefined-continuation subset. Its canonical architecture does not keep a faulty unit running in place. It terminates the unit and recovers from a known-good state. Erlang/OTP, built for telecom availability on the Ericsson AXD301 switch, is the model: its "let it crash" discipline lets a process that detects a fault die and a supervisor restart it, and the high availability it reaches comes from that terminate-and-restart, not from executing past the fault. [16] The transfer is bounded. Erlang reaches this through isolated processes with no shared mutable state, an isolation a C++ program continuing in place does not have, so the analogue supports the terminate-and-recover boundary at the unit level and not a claim that in-process C++ continuation is equivalent. What it carries is only that the availability these systems reach comes from recovering from a known-good state after the

faulty unit stops. Functional-safety practice draws the line the same way: on a detected fault a system transitions to a safe state - a reset, a failover, a deliberately entered degraded mode - rather than continue on a state its own model no longer defines. In these domains availability is a property of the recovery boundary.

Terminate-and-restart is the terminating response at the unit level, and the hook it depends on is the one preserved here: the handler runs and logs before the unit stops, so the supervisor and the operator learn what failed. The subset the objection does not reach is the defined-replacement class of Section 5, where continuation yields a specified result and the question is left open. A team that has weighed this and still wants to continue in-process past an undefined-state violation is served by the explicit, non-portable opt-in of Section 7.

---

4. The cost, and the rule it already breaks

To the deployment record, this section adds two findings: the continuation carries a cost the reference implementers decline to incur, and it runs against a decision the committee has already adopted for C++26.

The cost of continuing is not a matter of a branch. Turning a core-language operation into a checked operation that can invoke a handler and continue requires exception-handling machinery around the check: because whether the program's handler is `noexcept` is a link-time decision, P2900R14 [1] Section 3.6.6 notes that "the compiler ... has to generate the correct instructions for exception handling around every contract assertion." The implementers who ship hardening declined this. P3191R0 [17], from the libc++ team, sets the production requirement that a contract violation "should generate no code at all beyond the equivalent of a branch and a `__builtin_trap()`," with "no exception-handling code being generated around contract predicates," and describes the handler-and-object path as "a lot of code and data being generated for a single assertion." The committee's own response to this cost was to add the `quick-enforce` semantic, which skips the handler entirely, recorded in P3198R0 [18]. The isolated cost of the continuation over a trap has not been published as a measured figure. What the record shows is that the reference implementers decline the continuation path in production and the committee added a semantic to avoid its overhead. The cost falls on the portable guarantee rather than on any one build: an implementation required to offer `observe` for all implicit assertions emits the machinery around every checked operation whether or not a given build selects it (Section 5). The finding it supports is therefore that continuation should not be a semantic every implementation must carry, not that a particular deployment saves the cost by choosing termination.

For the adjacent case, the committee has also already decided this question. P3878R1 [19], adopted into C++26, established that a standard-library hardened precondition may not be evaluated with a non-terminating semantic, on the reasoning that 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 the core-language checks whose continuation is likewise undefined, a detected null dereference or out-of-bounds access, the same reasoning applies one level down. It does not reach the class in Section 5, where continuation is into a defined result. One disclosure belongs here: the lead author of P3878R1 is a co-author of this paper and its companions, though the decision was the whole committee's, and the argument stands on the deployment record and the security literature that follow even if the precedent is set aside.

On this premise, the present analysis and the Contracts proposal agree, though they reach a different conclusion about the default. Doumler and Berne write in P3097R2 [20] that once a program

> *is found to be in a possibly corrupted state, executing any user-defined code could result in a vulnerability.*

They keep the `observe` semantic available nonetheless. The same hazard is the reason a corrupted-state continuation should not be the default. The agreement is on the danger, and the resolution is where the two diverge.

The security literature is consistent with it. Microsoft's fail-fast documentation states that on a detected corruption "no exception handlers are invoked because the program is expected to be in a corrupted state." [21] The glibc maintainers removed even the backtrace from the heap-corruption path, on the reasoning that "doing more work at this point risks ... enabling code execution exploits." [22] The CERT C++ secure-coding rule ERR56-CPP states that "a violated invariant leaves the program in a state where graceful continued execution is likely to introduce security vulnerabilities." [23] Work on exception unwinding as an exploit surface (CHOP, NDSS 2023 [24]) shows that running the unwinder over corrupted state can defeat shadow stacks. Because C++26 Contracts are not yet deployed, no published incident names a C++ contract-violation handler. The evidence here is transfer, on the principle stated in Section 1, from mechanisms that faced the identical choice and chose to terminate: the hazard is a property of the undefined state, which these checks produce, not of the security purpose that first documented it. Where continuation runs on corrupted or undefined state, this argument is strongest. For the defined-replacement class of Section 5, it does not apply. For the checks whose continuation is undefined but whose violation is not memory corruption, the transfer is from the principle rather than from the mechanism: the state is undefined, and executing further on undefined state is what the fail-stop doctrine treats as the hazard.

---

5. Where continuing is defined

One class of core-language checks continues into defined behaviour, and the objection in Section 4 does not reach it.

Not every core-language check continues into undefined behaviour. P3100R8 [2] gives a defined replacement to cases such as signed-integer overflow, which can be specified to produce a wrapped result, so that continuing after the check yields a defined value rather than undefined behaviour. For that class, the objection in Section 4 does not apply, because continuation is into defined behaviour, and `observe` there is coherent.

What the concession does not supply is a deployed user. The field's response to signed overflow is either `-fwrapv`, which defines wraparound with no handler and no log, or `-ftrapv`, which terminates. P3100R8 [2] Section 5.4 maps the first to the `ignore` semantic and the second to `quick-enforce`. Neither is the log-and-continue that `observe` would add. So even in the class where continuation is defined, the deployed responses are silent replacement and termination, and no surveyed deployment logs and continues by default. The question the evidence leaves standing is who requires that response, and no surveyed deployment answers it. Where a deployer selects a continuing semantic for this defined class, that is a defensible choice. The finding here is limited to the class whose continuation is undefined.

Regardless of the safety objection, a second constraint applies to the defined-replacement class: the cost. If any implicit assertion can be evaluated with a continuing semantic, the exception-handling machinery the reference implementers decline in Section 4 must be generated around every checked operation, because whether the deployer selects `observe` or a terminating semantic is a build-time or link-time decision unknown to the compiler at code-generation time. This makes the cost a property of the specification rather than of any deployer's choice: an implementation that offers `observe` must emit the machinery whether or not a given build selects it. The cost is also class-agnostic: it applies to a signed-overflow check that continues into a defined wrapped result just as it applies to an out-of-bounds check that continues into undefined behaviour. In P3191R0 [17], the implementers' objection - "no exception-handling code being generated around contract predicates" - does not distinguish between the two classes, and the generated code cannot.

---

6. A terminating response

This section states the response the evidence supports and shows it in code, reusing the C++26 semantics without adding to them. It separates two claims the evidence supports to different degrees. The first is the default: a terminating response, invoke the handler then terminate, is what the deployment record and the committee's own recommendations support. The second is narrower, and the evidence supports it less strongly: whether a continuing response should remain available as a portable guarantee for the class of checks whose continuation is undefined. Whichever way the second question is resolved, the default holds.

The response is to invoke the handler for its telemetry and then terminate, and to prevent a handler throw from escaping the checked expression. In C++26, both halves already exist. The first is the `enforce` semantic: the handler is invoked and, on a normal return, the program is contract-terminated (P2900R14 [1], `[basic.contract.eval]`). The second is the existing rule that an escaping exception at a non-throwing boundary results in termination. Applied to an implicit assertion, a handler throw contract-terminates rather than propagating. Expressed as a restriction on P3100R8's proposed implicit assertions, implicit core-language assertions would exclude the observe semantic, and a handler throw from one contract-terminates. Under either architecture the restriction reaches the same checks: whether these implicit checks are configured through Contracts Labels or governed by a profile, the class it covers is the same, implicit checks on core-language undefined behaviour whose continuation is undefined. This adds no new semantic. It reuses `enforce` and the existing termination rule. This is the response the C assert integration already takes: P3290R4 [25] Section 2.2 specifies that `assert` invokes the handler nonthrowing and then terminates. On 2026-07-08, SG22 (C/C++ Liaison) polled whether `assert` should let exceptions thrown from contract-violation handlers propagate. Both bodies reached consensus against, WG21 by 1-0-2-7-2 and WG14 by 0-0-0-5-2 (SF-F-N-A-SA). [26]

The default claim is not a new idea: the co-authors of P3100R8 [2] and P2900R14 [1] recommend the same default. Berne and Lakos, in P3558R1 [27], recommend

> *a default evaluation semantic, when nothing else is specified, of* `enforce` *for all core-language preconditions.*

The enforce default supported here is the one these authors already recommend. The former convener of WG21 records the same recommended practice. For an enforced safety check, P3081R2 [28] advises implementations to

> *use the evaluation semantic* `quick_enforce` *or* `enforce`*, and emit a diagnostic if the evaluation semantic used in execution may be* `observe` *or* `ignore`*.*

Both recommendations state the terminating response as the default, and the default claim rests on the deployment record of Section 3 independently of the narrower question below.

The narrower claim concerns availability, not the default. Removing the continuing semantic as a portable guarantee every implementation must carry rests, for the undefined class, on the corrupted-state hazard of Section 4 and the committee's adopted decision for the adjacent case. The cost of Section 4 is class-agnostic (Section 5): it argues against carrying `observe`-continuation as a portable guarantee at all, which is why the shape that survives is the non-portable opt-in of Section 7, and it does not by itself single out the undefined class. For the defined-replacement class of Section 5, where continuation yields a specified result, the question is open and the committee may reach a different answer. If a continuing response is retained, it takes the shape of the opt-in, non-portable facility of Section 7.

Consider a detected overflow inside a function marked non-throwing, adapted from P3100R8 [2] Section 5.5:

```
int f(int x) noexcept { return x + 1; }
```

Under a throwing response, a detected overflow in `x + 1` invokes a handler that may throw, and the exception attempts to leave a function the `noexcept` operator reports as non-throwing. The exception cannot both leave `f` and honor `noexcept(f)` reporting `true`. Doumler and Berne name this Option A in P3100R8 [2] Section 5.5: the handler may throw and the exception propagates, while the `noexcept` operator keeps its value but changes its conceptual meaning, from "evaluating this expression cannot throw" to "evaluating this expression cannot throw unless there is a contract violation".

P3100R8 reports that SG21 reached strong consensus for Option A and against the non-throwing Option B, but that consensus was taken on P3541R1 [29], the Contracts paper that first proposed the two options. Extending the throwing choice to implicit core-language assertions has not been polled by EWG, which P3100R8 notes "is, of course, entitled to making a different design choice than SG21 did".

Since P0012R1 [30] in C++17, the `noexcept` operator has been part of the function type. The throwing response keeps its value at `true` while changing what that `true` guarantees. The terminating response leaves the operator alone. Under it, the handler for a violation in `x + 1` logs and the program terminates. `noexcept(x + 1)` remains `true` and remains honest, because nothing escapes. The security concern this raises is not hypothetical: N3103 [31], from 2010, records that a `noexcept` violation allowed to continue "can be exploited by a malicious user to bypass security restrictions," and recommends immediate termination.

There is a second reason to contain the throw, visible without leaving the standard library. Turning a detected violation into a thrown exception unwinds the stack through code that did not anticipate a throw at that point, running destructors on objects

whose invariants are momentarily broken, so a frequently benign overflow becomes a double-free or a half-destroyed object. The standard library already does not throw at these moments: `std::vector` reallocation uses `move_if_noexcept` so that a throwing move cannot corrupt the container mid-operation. Bloomberg's test infrastructure records the same collision from the other side: a commit notes that when a destructor "becomes implicitly `noexcept`, so throwing the test exception type out of the assert handler triggers a call to `terminate`," and the workaround is a non-throwing, terminating handler. [32] The terminating response is the one that does not manufacture this defect.

The underlying problem is general, and it runs in four linked steps. First, the exception-safety model of C++ rests on knowing which operations can throw. Code is written to maintain its invariants across those operations and only those. Second, a throwing implicit handler turns every core-language expression into a potential throw point, and no existing code was written to be exception-safe at those points. Third, the result is not a recoverable exception but an unwinding through code that cannot maintain its invariants along the way. Fourth, that unwinding over broken invariants is the condition the CHOP work [24] shows turns stack unwinding into a security vulnerability. The chain resolves the same way each time: writing exception-safe code is possible when the set of throwing operations is known, and effectively impossible when any expression can throw.

A distinction bounds this argument. It reaches the continuation whose handler may throw: the escaping exception is what leaves `noexcept`, unwinds through code that did not anticipate a throw, and turns unwinding into the exploit surface the CHOP work describes. A non-throwing log-and-continue, the shape Bloomberg's `bsls_review` handler takes when it logs and returns normally, [13] does not raise the `noexcept` question, because nothing is thrown. That response is reached instead by Section 4: it continues execution on a state the language does not define, which is the hazard the security literature there identifies. The two forms of continuation fail for different reasons, and separating them keeps the `noexcept` argument scoped to the throwing form.

The terminating response leaves build-time configurability intact. Under P3100R8's proposal, a deployer still selects among `enforce` (invoke the handler, then terminate), `quick-enforce` (trap without the handler), and `ignore` (no check) for an implicit core-language assertion, and keeps full control of the handler body. The same three choices remain under a Profiles-first architecture, where a profile selects the evaluation semantic for the checks it governs. For one class of assertion, the finding narrows the menu and leaves untouched the principle that the evaluation semantic is a build-time choice.

The bifurcation this draws, implicit core-language assertions terminate while explicit contracts may still throw, is deliberate and defensible. It is the same line P3878R1 [19] drew for hardening, a decision this paper's co-author led (disclosed in Section 4), and it rests independently on the difference the scope in Section 2 named: an author-written precondition can encode a recoverable condition, whereas a detected core-language violation means the program is already in a state the language does not define. This is also the answer to the objection that a bug is a bug and the semantics should be uniform: the committee has already treated the two differently, one level up.

---

7. If continuing must be possible

This section addresses the case in which the committee concludes a continuing response must be available to someone. Where it is available, the deployed precedents share one shape: an opt-in, non-portable facility, bounded to an adoption period rather than a default or a portable guarantee.

The case for a continuing response deserves its strongest statement, and two forms of it are real. Availability is the first: a long-running service or a fault-tolerant embedded system may prefer bounded degradation to a hard stop, so that for such a system a violation that terminates is itself the failure. The second is migration: a large codebase that adds a new check needs a way to find the violations it surfaces before it enforces them, so that turning the check on does not stop a program that works today. Both are legitimate.

The answer distinguishes them. The migration need is met by the hook without the continuation: the terminating response already invokes the handler and logs, so a team sees every violation before it enforces and moves from find to fix to enforce without ever running past a live violation. The logging that surfaces those violations does not depend on the contract handler; a sanitizer surfaces the same ones (Section 2). Where continuation is into a defined result, the class of Section 5, the availability need is real, and there the objection does not apply. Where continuation is into undefined behaviour, keeping the program running is not availability but execution on a corrupted state, which the security literature in Section 4 identifies as the more dangerous failure. What remains, a team that has weighed this and still chooses to continue past an undefined-state violation, is served by the facility below.

A continuing response has a defensible shape, and the field already uses it. It is the shape of an explicit, non-default opt-in that its own documentation labels as undefined behaviour and disclaims. The libc++ `observe` semantic is documented in these 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] Bloomberg's `bsls_review` is the same idea deployed: an explicit, temporary downgrade from `assert` to log-and-continue while a newly tightened check is rolled out, on library-level checks rather than core-language undefined behaviour. [33] [13] The .NET platform is the closest full-lifecycle precedent: it once let managed code opt into catching corrupted-state exceptions, then discouraged it with an analyzer whose guidance is "the safest option is to allow the process to crash," and ultimately made the opt-in inert. [34]

These precedents share three properties, and a continuing facility for core-language checks would need all three. It is opt-in, so the default remains the terminating response. It is non-portable and implementation-defined, so it does not become a guarantee every implementation must carry, which is what would re-import the cost in Section 4. And it is marked and disclaimed at the point of use, so that continuing past a detected violation is an affirmative choice recorded in the source, owned by whoever makes it, rather than a behaviour hidden in a default. A facility with those three properties answers the migration case that motivates `observe`, because the transitional need, to see violations before enforcing them, is met by the hook, which the terminating response already provides. Continuing during a rollout is a per-project, non-portable concern that a build already expresses today.

Whether such a facility should be built is a separate question from its shape: an explicit opt-in outside the portable semantics, never the default.

---

8. Problems with this analysis

This analysis has problems, and they are worth stating. It reasons from analogy rather than from a conforming implementation. It draws on security literature for a feature that is not a security feature. It proposes to restrict a semantic whose availability matters to teams with large legacy codebases. And the strongest deployed counter-example to its survey comes from the same institution whose co-authors recommend the default this analysis finds the evidence supports.

The first problem: the security literature does not apply to a correctness tool

Contracts are a correctness tool for finding bugs, and citing fail-fast policies, CERT rules, and exploit research can look like importing a category that does not belong. The objection fails on the transfer principle stated in Section 1: the hazard is a property of the program state, not of the tool's purpose. A correctness tool that detects a core-language violation leaves the program in the same undefined, possibly corrupted state the fail-fast mechanisms faced, and executing user code on that state carries the same danger whether the mechanism that detected it was built for security or for correctness. The literature is cited for that state hazard, which is category-independent, not for the claim that contracts are a security feature. Independently, the authors of the feature under examination document the same hazard: Doumler and Berne write in P3097R2 [20] that once a program "is found to be in a possibly corrupted state, executing any user-defined code could result in a vulnerability." Whether to weigh this evidence is a judgment the committee makes; it is placed here to be weighed or set aside.

The second problem: restricting the semantic limits deployer choice

The standard should offer all four semantics and let each deployment decide. Restricting the menu takes a legitimate choice away from teams that need it. Deployer autonomy matters, and the terminating response preserves three of four semantics and the full customizability of the handler body. What it removes is execution past a state the language does not define. P3878R1 [19], adopted into C++26, made the same restriction for standard-library hardened preconditions. The committee adopted it.

The third problem: the migration story requires seeing all violations before enforcing

A large codebase that turns on a new check needs to find every violation it surfaces before it enforces. If the first violation terminates the program, the team sees one signal per deployment. Under the terminating response, the handler fires, logs the violation, and the program terminates. With the first violation fixed, the next deployment reveals the second. Across codebases measured in hundreds of millions of lines - Google's production fleet, libc++ across Apple's platforms, glibc across every Linux distribution - the hardened implementations in Table 1 rolled out checks with the terminating response and without a log-and-continue semantic. The rare violation that only production inputs trigger is handled the way large deployments handle any risky change: a staged rollout surfaces it in a canary population first. There termination stops only the canary, and the logged violation identifies the site to fix before the check reaches the full fleet.

The fourth problem: Bloomberg's `bsls_review` is the deployed counter-example

Bloomberg maintains `bsls_review`, a companion to `bsls_assert` that logs and continues while a newly tightened check is rolled out, and Bloomberg relies on the availability of that log-and-continue response to add checks to working production code. [13] The finding here does not contest that availability. It contests the continuing response for one class, the core-language-undefined class, and only as a default. Section 3 sets out the level distinction and its limits: `bsls_review` continues at the library level, and the library-versus-core-language line is this paper's analytical construct rather than one Bloomberg draws. What the counter-example adds is that the deployed shape - a terminating default with continuation documented as an interim adoption aid - is the shape the finding reaches. The facility Section 7 describes, an explicit, non-portable opt-in bounded to an adoption period, is that shape, so the deployed practice preserves the rollout mode the finding leaves open. Libc++ documents its `observe` semantic in the same terms: "should not be used outside of the adoption period." [5] And the co-authors of Bloomberg's contracts framework recommend, in P3558R1 [27], "a default evaluation semantic, when nothing else is specified, of `enforce` for all core-language preconditions."

The fifth problem: the proposed restriction paternalizes the deployer

Restricting `observe` for implicit assertions tells every deployment that the committee knows their codebase better than they do. A team that has evaluated the trade-offs and chosen log-and-continue at scale, even for years, is making an informed engineering decision, not requesting guardianship. But P3878R1 [19] already makes this restriction for library hardening, on the reasoning that continuing past a diagnosed violation defeats the purpose of diagnosing it. The paternalism charge applies to that adopted C++26 decision equally, or the principle permits the same restriction one level down.

For the class it covers, the restriction also removes no coherent choice. The axiom that makes runtime-checkable undefined behaviour detectable at all, that undefined behaviour carries no specification of what follows, is the axiom that makes its continuation unspecifiable: the handler returns, and what executes next depends on a program state that has no defined meaning. By the standard that a component's test suite is its specification, there is nothing to specify past the violation, because a detected core-language violation falls outside every contract the tests express. Undefined behaviour cannot be documented, and its continuation cannot be specified either. This is the same undefined state that the method note in Section 1 turns on: the state that carries the hazard is the state that cannot be specified past, and both follow from the one axiom.

This unspecifiability argument reaches only the class whose continuation is undefined. For the defined-replacement class of Section 5, where continuing yields a specified result such as a wrapped value, `observe` is coherent. The objection to it there is the cost in Section 5, and the argument here does not apply.

In its throwing form the continuation also inverts a separation the language keeps elsewhere: the evaluation semantic is a deployment property chosen at build or link time, yet a throwing handler leaves `noexcept` reporting `true` while changing what that `true` guarantees (Section 6). That lets a deployment decision govern the meaning of a type-system property, the separation the polymorphic-allocator model was built to preserve.

---

9. The configuration question is separate

The response question and the configuration question are independent, and a boundary between them is worth stating.

The first is what the response to a detected core-language violation should be. The second is who configures that response: whether the selection of a semantic for a core-language check is expressed through the Contracts configuration facility (P3400R3 [35]) or through the Profiles framework (P3589R2 [36]). Routing core-language checks through the contract-violation handler is the layering that P4297R0 [4] examines and would put to an explicit poll.

The layering is described here as the arrangement P3100R8 proposes. It is not resolved. The two questions can be answered independently, and the finding that the response should terminate holds regardless of which facility owns the selection. Under a Profiles-first architecture the finding says the same thing: a profile that covers core-language undefined behaviour defaults to a terminating response for the class whose continuation is undefined. The deployer's ability to observe a violation is preserved by the handler invocation that precedes termination. The comparison of the ownership models is in P4306R0 [3].

---

10. Conclusion

Of the responses to a detected core-language violation, the terminating response - invoke the handler, log, and terminate - is the steady-state production default across every hardened implementation surveyed here. The continuing response is the steady-state default in none of them for core-language undefined behaviour; where it ships, it is an adoption aid. It carries a cost that keeps it out of the portable guarantee, and it runs against P3878R1 [19], the decision C++26 already adopted for the adjacent case. Where continuation is defined, the record still shows no deployment that adopts log-and-continue as its default.

The finding of this paper is that the terminating response is the one the evidence supports as the default, and that it can be had by reusing the C++26 `enforce` semantic and the existing termination rule, without a new semantic and without changing the meaning of `noexcept`. Because no compiler yet implements these assertions, the finding rests on deployed analogues rather than a conforming implementation. Across both classes of check, that default holds. The narrower finding, that a continuing response should not be a portable guarantee every implementation carries, is scoped to the class whose continuation is undefined. For the defined-replacement class of Section 5 the question is left open. If a continuing response is to exist for the undefined class, it belongs as an explicit, per-project opt-in outside the portable feature. Whether to build such a facility is a question for the committee, not this paper. The record is placed here for the committee's use. The paper makes no request.

---

11. Disclosure

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

Vinnie Falco is the founder of the C++ Alliance, which maintains a Clang fork for Profiles work, and prefers the family of responses in which no exception escapes an implicit contract assertion. That is a stake in the outcome. Ville Voutilainen is a longtime WG21 participant, a co-author of the C++26 Contracts facility (P2900R14 [1]) and lead author of P3878R1 [19], the

adopted decision this paper reuses in Sections 4, 7, and 9. The reader should weigh what follows accordingly.

The intent of this paper is `info`. It argues a position, that a terminating response is the one the evidence supports for implicit core-language assertions, but it proposes no wording and requests no poll.

This paper changes nothing in ratified C++26. C++26 Contracts (P2900R14 [1]) are treated as fixed: the four evaluation semantics, the single violation handler, and the deliberate allowance that a handler may throw from an explicit contract assertion all stand unchanged. The question here belongs to the open C++29 work on implicit assertions (P3100R8 [2]), and the paper addresses only that.

Up front, one limitation is disclosed: no compiler yet implements implicit contract assertions with any semantic, so the paper reasons from deployed analogues. As of the July 2026 mailing, P3100R8 [2] reports no implementation or deployment experience with the proposed implicit assertions, and the GCC, Clang, and MSVC C++26 status pages list none.

In the July 2026 mailing, this paper is one of a set on runtime-checking configuration. P4306R0 [3] compares the configuration-ownership models on the public record, and P4297R0 [4] asks EWG to decide the ownership relationship by an explicit poll. This paper is scoped to the response question and cross-references those rather than repeating them.

This paper was prepared with the assistance of generative tools. The authors are responsible for its content.

This paper asks for nothing.

Acknowledgements

The authors of P2900R14 [1] and P3100R8 [2], whose careful statement of the design made the questions in this paper precise. 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] P4306R0 - "Configuring Runtime Checking: Profiles and Implicit Contract Assertions" (Vinnie Falco, Ville Voutilainen, 2026).

[4] P4297R0 - "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 Neățu, 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] BDE commit 03fdb2e1e - "Reduce clang-tidy and coverity warnings when using 'bslmt_once'" (Nathan Burgers, 2019).

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

[16] Making reliable distributed systems - "Making reliable distributed systems in the presence of software errors" (Joe Armstrong, PhD thesis, 2003).

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

[18] P3198R0 - "A takeaway from the Tokyo LEWG meeting on Contracts MVP" (Andrzej Krzemieński, 2024).

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

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

[21] __fastfail intrinsic - "__fastfail" (Microsoft, 2023).

[22] glibc BZ #21754 - "malloc: Abort on heap corruption without a backtrace" (Florian Weimer, 2017).

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

[24] CHOP - "Let Me Unwind That For You: Exceptions to Backward-Edge Protection" (Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, Cristiano Giuffrida, 2023).

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

[26] cplusplus/papers #1943 - WG21 public paper tracker issue for P3290, recording the SG22 2026-07-08 poll on assert exception propagation.

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

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

[29] P3541R1 - "Violation handlers vs noexcept" (Andrzej Krzemieński, 2025).

[30] P0012R1 - "Make exception specifications be part of the type system, version 5" (Jens Maurer, 2015).

[31] N3103 - "Security impact of noexcept" (David Kohlbrenner, David Svoboda, Andrew Wesie, 2010).

[32] BDE commit c73a697a1 - "Fix bsltf::AllocTestType test driver on C++11" (Alisdair Meredith, 2015).

[33] P2877R0 - "Contract Build Modes, Semantics, and Implementation Strategies" (Joshua Berne, Tom Honermann, 2023).

[34] CA2153 - "CA2153: Avoid handling Corrupted State Exceptions" (Microsoft, 2023).

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

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

[37] bsls_assert - "bsls_assert: Provide build-specific, runtime-configurable assertion macros" (Bloomberg BDE, 2019).