Configuring Runtime Checking: Profiles and Implicit Contract Assertions

Document Number:	P4306R0
Date:	2026-07-14
Intent:	Inform
Audience:	EWG

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

Table of Contents

Abstract

Measured against the committee's own criteria, none of them awards ownership of runtime-checking configuration to either proposal.

Two proposals answer one question - how a program configures the runtime checking of core-language undefined behavior - and, as P3100R8's own Section 7.2 states, if both are kept then one must be specified in terms of the other. This paper assembles the public record and measures both candidate owners against criteria already in the committee's record - existing practice, deployment and field experience, systematic coverage of undefined behavior, and freedom from dialects - and taken one by one those criteria settle configuration ownership for neither. The P3100 model leads on systematic coverage, the one criterion carrying a poll, but that lead does not resolve ownership, because both owners consume the enumeration identically; existing practice reads both ways and names different owners. Applied evenly, the deployment criterion does not by itself assign the base role, because both proposals pair a deployed lineage with an unshipped specification; what the record establishes is narrower - the shipping practice terminates, and P3100's added machinery of implicit assertions, Labels, and a replaceable handler has no implementation. The record also finds no deployment of the layering's single-architecture premise - one handler slot, one menu of evaluation semantics, one configuration base - together with removed or condemned precedents for its nearest standardized relatives and a first integration already departing from it. On the guarantee the two models differ in both author and reach: P3100's menu makes what an expression means configuration-selected across every checkable operation and, only in the configurations where a violation can throw, makes what the noexcept operator may assume about it configuration-selected as well, while a Profile raises the question only where its author defines a non-terminating meaning. This comparison is supplied for the explicit decision its companion

P4297R0 asks EWG to take, on the evidence.

Revision History

R0: July 2026

- Initial version. Companion to P4297, which asks EWG to decide the relationship between the two proposals by an explicit poll on a dedicated paper. This is that dedicated comparison.

---

1. Introduction

Two bodies of work now answer the same question: how a program configures the runtime checking of core-language undefined behavior. Kept together, as P3100R8's Section 7.2 states, one of them must be defined in the other's terms. The comparison below measures the two candidate owners of that configuration against criteria already in the committee's record, and it is the dedicated comparison its companion P4297R0 [8] asks EWG to weigh: P4297R0 asks for the polls, and this comparison is the record those polls would weigh.

The related work is two-sided. One side is the implicit-contract-assertion machinery of P3100R8 [1] (Doumler and Berne), which configures runtime checks through the C++26 Contracts evaluation semantics and the Labels of P3400R3 [2]. On the other side is the Profiles framework of P3589R2 [3] (Dos Reis) with the individual profiles of P3984R0 [4] (Stroustrup), under which a profile owns the guarantee directly. Section 2 sets the two side by side and states why they cannot both be primary. The criteria the comparison applies are drawn from the Direction Group's P2000R5 [5], the deployment-experience standard applied in P3608R0 [6] and proposed in the companion P4297, the polled Hagenberg mandate, and P3874R1 [7], and Section 3 states the provenance of each.

Four contributions follow:

1. It assembles the two-sided public record for the comparison, including the deployment ledger for both forms of checking (Section 5).
2. It states the criteria already in the committee's record, with the provenance of each (Section 3).
3. It separates the deployed form from the unshipped specification on the deployment criterion, so the experience is weighed against the right object (Section 5).
4. It tests the layering's single-architecture premise - one handler slot, one menu of evaluation semantics, one configuration mechanism as the base every checking facility is specified in terms of - against the deployed record of failure-response mechanisms (Section 8).

One assumption underlies the comparison: that the two configuration mechanisms cannot both be primary. Where the two features collide - granular, in-source control (Section 7.2) - the proposal asserts the dependency, and the partition analysis in Section 2 extends it to the general case: wherever both features speak to the same check, one must be specified in terms of the other.

---

2. Two Proposals Compete to Own One Configuration

The two bodies of work can coexist. But P3100R8's Section 7.2, quoted below, states that if both are kept, one must be specified in terms of the other. In its Section 4.4, the same paper states that "we must make it clear which feature is responsible for providing the user-facing configuration mechanism for which tool", and its Section 7.2 restates that requirement as "we need to make it clear" [1]. Read together with the revision record in Section 9, that requirement assigns the base role to one feature, and defines the other in its terms. That arrangement is called configuration ownership here. Which feature holds it is the question under comparison.

The first is the implicit-contract-assertion machinery of P3100R8 [1] ("A framework for systematically addressing undefined behaviour in the C++ Standard", Doumler and Berne - the current revision of the framework). It guards the runtime-checkable cases of core-language undefined behavior with implicit contract assertions, configured through the evaluation semantics of C++26 Contracts and, for in-source granularity, through the Labels proposed in P3400R3 [2]. On how Profiles relate to that machinery, its Section 4.4, under "Configuration", states the open question and then the suggested answer:

> *There is no consensus yet on how Profiles relate to and compose with other proposals such as the holistic strategy for removing UB from C++ proposed here.*

> *It therefore seems logical to define Profiles as a higher-level feature building on top of these three basic tools (see Figure 4). Given that these three features are configurable, a concrete profile could be defined as being a named configuration preset for these features.*

The modality is the proposal's own: "seems logical", "could be defined". Those sentences are not read here as settled design, and the comparison does not depend on them. It rests on the requirement the proposal attaches in the next breath - Section 4.4 continues: "If we pursue such a 'multi-level' strategy, we must make it clear which feature is responsible for providing the user-facing configuration mechanism for which tool (see also Section 7)" [1] - and on its Section 7.2, which restates that requirement where the two features meet, with the failure condition named:

> *For granular, in-source control of the evaluation semantics of implicit contract assertions, we need to agree whether this happens via directives such as the ones proposed in [P3400R3] and shown here, or by using the syntax proposed in the Profiles framework as proposed in [P3589R2]. If we want to have both, we need to specify one in terms of the other to avoid an incoherent and messy design.*

Section 7.2 also states what must be avoided: a situation "where the same functionality is provided simultaneously by different features in incompatible ways" [1]. "We need to agree whether" is an invitation to decide. The agreement the proposal asks for is the decision its companion P4297 proposes EWG schedule. Thus the proposal states the ownership question in its own voice, and this paper's term for the answer - configuration ownership - names what its responsibility

sentence asks.

The second is the Profiles framework of P3589R2 [3] (Dos Reis) together with the individual profiles of P3984R0 [4] (Stroustrup). The framework provides per-translation-unit activation and local suppression attributes, supports standard, implementation-defined, and third-party profiles, and constrains a profile to only reject code, never to change the meaning of code that compiles [3]. In that model the profile owns the guarantee directly. P3984R0:

> *A profile cannot change the semantics of a program beyond defining the meaning of some forms of undefined behavior.*

The two arrange the same pieces in opposite orders. Under the P3100 model, the implicit-contract-assertion machinery is the base and a profile is a named preset that selects a configuration of it. Under the Profiles model, the framework is the base that owns the guarantee and the response to a failed check, and the lower-level checking tools sit beneath it.

P3100R8's Section 7.2 offers a second arrangement of its own, which varies the profile's role without varying the owner. Immediately after the requirement quoted above, the proposal writes that a profile controlling runtime checks "can be defined as essentially a declaration that expands to [P3400R3] directives", or:

> *Alternatively, we could design Profiles as an auditing feature rather than a configuration feature: instead of actively enabling certain configuration options, the effect of a Profile would be that the program is ill-formed if the configuration options chosen via [P3400R3] directives or other mechanisms are not compatible with the guarantees that that Profile ensures.*

Under the first branch a profile is a preset that expands to the contract machinery's directives. Under the second a profile validates a configuration it does not own: the P3400 directives select the configuration, and the profile is reduced to a conformance checker over choices made elsewhere. Both branches assign configuration ownership to the contract machinery. What varies is only whether the profile selects the configuration or polices it. The auditing branch is also not what "profile" has meant where one has been standardized: Ada's Ravenscar profile, in the language's own rationale, "is a mode of operation", and "The general idea is that a profile is equivalent to a set of configuration pragmas" [64] - active configuration, standardized since Ada 2005. The audit shape does have deployed successes: Fedora's annocheck exists "to examine how a binary was built and to check that it has all of the appropriate security hardening features enabled" [65], and Debian's blhc "checks build logs for missing hardening flags" [66]. But in both, the distributions' default build flags inject the guarantees, and the audit polices that they survived the build system. Deployed audit rides on active configuration rather than replacing it. The alternative therefore confirms the comparison's premise twice: it is a second arrangement that is itself unimplemented, and it assigns ownership to the same owner while varying only the profile's role - under a proposal whose own text says there is no consensus yet on how the two features compose.

A third arrangement divides the territory instead of assigning a base: the framework owns per-translation-unit activation and the naming of guarantees, the contract machinery owns per-assertion semantics, and neither is specified in the other's terms.

The texts already quoted foreclose it. Only where the spheres never meet does a partition hold, and both sides put both features on the same lever: P3984R0 gives the profile author the choice of the violating case's meaning [4], and P3100R8's Section 4.4 has the profile selecting configurations of the evaluation semantics [1]. Wherever an active profile and an in-source directive speak to the same check, something must say which wins, and Section 7.2 names the alternative: "an incoherent and messy design" [1]. At the collision points, a precedence rule is an ownership assignment under another name. Partition relocates the question. It does not answer it.

Terms used in this comparison

This comparison uses terms from the Contracts and Profiles proposals, collected here for a reader who does not track SG21, the study group responsible for Contracts.

- Implicit contract assertion: a check the compiler inserts at a core-language operation that can have undefined behavior - a pointer dereference, an array index, a signed addition - evaluated like a C++26 contract assertion on the operation's precondition.
- Evaluation semantic: the mode that decides what happens when a check's predicate is false. The P3100 model uses five. Ignore performs no check and runs the operation's existing behavior. Observe calls the violation handler and then continues. Enforce calls the handler and then terminates. Quick-enforce terminates immediately, without constructing a violation object or calling a handler. Assume performs no check and lets the compiler optimize on the predicate being true, as C++ does today.
- Quick-enforce: the evaluation semantic that terminates on a failed check with no handler and no violation object. It is what a trap instruction implements, and it is the semantic production hardening uses today.
- Violation handler: the replaceable function C++26 Contracts calls when a checked assertion fails. Depending on the semantic it may report and continue, report and terminate, or throw.
- Const-ification: the rule that a contract predicate treats the objects it reads as const, so evaluating the check cannot itself modify program state.
- Labels: the in-source directives proposed in P3400R3 for choosing and constraining the evaluation semantic of assertions at a chosen granularity.
- Named-guarantee form: a vendor-defined set of named safety guarantees that a build turns on as a unit - hardened standard libraries and sanitizer build modes are examples - which is the shape the Profiles framework standardizes and the P3100 model also expresses as a named configuration preset.

---

3. The Criteria and Their Provenance

The comparison measures both proposals against criteria already in the committee's record, and it states where each criterion comes from, because the criteria have different provenance and the reader is entitled to weigh them accordingly. One point governs the weighing before any criterion is applied: a criterion that answers what to check does not decide who owns the configuration of the checking, because both candidate owners can guard the same cases. A second point follows from applying the deployment criterion evenly to both proposals. On a strict reading - has this proposal's own specification shipped - neither has: the P3100 machinery and the P3589/P3984 framework syntax are both unshipped. On a loose reading - does a deployed lineage match the proposal's shape - both match one, the named-guarantee set for Profiles and the compiler-inserted check for P3100, which P3100R8 maps into its semantics. Read either way, the deployment record does not by itself assign the configuration base role to one proposal over the other. What it does establish is narrower and holds under both readings: the shipping practice terminates, and P3100's added machinery - implicit assertions, Labels, the replaceable handler - has no implementation, as distinct from its deployed lineage. Four criteria recur in the record.

The first is existing practice. P2000R5 [5] ("Direction for ISO C++"), Section 5:

> *We change the language and standard library by gradually building on previous work or by providing a better alternative to an existing feature.*

Provenance: an advisory paper of the Direction Group, without a poll behind it. Its authors include Bjarne Stroustrup, an author on the Profiles side of this comparison, so this criterion is one a disputant helped write. If the reader rejects it on that ground, the comparison in Sections 5 and 6 rests on the vendor deployment record directly.

The second is implementation and deployment experience. Provenance, disclosed in full because it is closest to this paper: P3608R0 [6] ("Contracts and profiles: what can we reasonably ship in C++26") applied it in this exact domain - "the standard library hardening is existing practice, and comes with very positive field experience reports" - and is co-authored by an author of this paper (Voutilainen) together with Wakely and Dos Reis, the author of P3589R2. The same criterion is the subject of P4297 [8], a companion paper by this paper's authors, whose Poll 3 proposes that EWG weigh deployment experience when it decides this relationship. That poll has not been taken, and this criterion is therefore a standard proposed by one side of the dispute and not yet adopted. It is used here because it is the criterion a comparison of shipped practice can be run against.

The third is systematic coverage of undefined behavior. Provenance: the one criterion here with a poll behind it. The Hagenberg poll's text, as reproduced in P3656R1 and in P3100R8's own history section, reads in full: "Pursue a language safety white paper in the C++26 timeframe containing systematic treatment of core language Undefined Behavior in C++, covering Erroneous Behavior, Profiles, and Contracts. Appoint Herb and Gašper as editors" - SF 32, F 31, N 6, A 4, SA 4, consensus [25][1]. The poll approves a work item and names both features as covered subject matter. It contains no configuration or ownership language, so it enters this comparison as a mandate for the systematic treatment rather than as an assignment of the base role to either feature. This criterion the comparison can weigh immediately, because the record is not contested: P3100R8's Appendix A enumerates the cases of core-language undefined behavior - 80 cases, 77 of them runtime-checkable [1] - and no Profiles paper offers an equivalent enumeration. On systematic coverage, the P3100 model leads today, and its enumeration is useful to every safety effort regardless of which proposal ships. One property of this

criterion matters for the weighing: it does not discriminate between the candidate owners. The enumeration answers what to check, and either owner consumes that answer identically - a profile can guard exactly the enumerated cases, and so can implicit assertions. Weighting this criterion dominant, as its poll invites, changes the comparison's totals but not its ownership finding, because both candidates inherit the enumeration.

The fourth is the strength of the guarantee and freedom from dialects. Provenance: P3874R1 [7] ("Should C++ be a memory-safe language?") states the guarantee standard, and the no-dialects standard is stated inside P3100R8 itself, quoted in Section 7 - the disputants' own texts, binding on their authors if on anyone.

Criteria the reader might expect and will not find here: wording maturity, integration with the already-adopted C++26 machinery, in-source granularity, and the cost of standardizing two configuration mechanisms. Each presupposes an answer to the ownership question rather than informing it. Wording maturity measures drafting progress, not fitness to own configuration. C++26 integration presupposes that building on the newest adoption is the operative reading of the direction sentence - Section 6's second reading, weighed there. In-source granularity compares a capability Labels propose and nothing has shipped, which the deployment criterion already covers. And the dual-mechanism cost is the cost the ownership decision exists to avoid, whichever owner is chosen - an argument for deciding.

Sections 5 through 7 weigh the remaining three criteria in turn - deployment and field experience, existing practice, the guarantee and dialects - systematic coverage having been weighed above. Section 8 applies the deployment criterion to the layering's single-architecture premise, and Section 9 records what configuration ownership has already cost in practice. From advisory to polled to party-proposed, the criteria differ in provenance, and each section's finding stands on the evidence inside it.

---

4. The Record at a Glance

The tables below collect the evidence Sections 5 through 9 source and defend. Each cell is a cited fact. For readers who want provenance, the exposition follows.

Table A: What Has Shipped

Neither proposal's specification has shipped. The asymmetry is in the deployed forms.

Implementation	Shipped	Default or opt-in	Measured cost	Response
libc++ hardening	LLVM 18, 2024 [13]	Xcode 16 build setting [14]	Google: ~0.30% [15]	trap instruction [33]
libstdc++ assertions	GCC 6, 2016 [16]	default at `-O0` since GCC 15 [16]	not separately reported	diagnostic, `abort()` [16]
MSVC STL hardening	VS 2022 17.14, 2025 [17]	opt-in	not separately reported	`__fastfail` [17]
WebKit	libc++ extensive level [75]	release builds [75]	"zero" end-to-end [75]	trap (libc++) [75]
Firefox 145	cross-vendor flag [76]	opt-in; release default pending [76]	"negligible" [76]	vendor-selected [76]
Android UBSan	Android 7.0, 2016 [50]	per-component [50]	not public	abort [50]
Chrome CFI	production builds [53]	enabled in official builds [53]	not public	process kill [53]
Apple `-fbounds-safety`	millions of lines of C [74]	production [74]	not public	deterministic trap [74]

Table B: Specification Status

Both sides have unshipped specifications and deployed lineage. The objects of standardization differ.

Component	Side	Implementation	Deployment
Implicit contract assertions	P3100	none	none
Labels (P3400R3)	P3100	none	none
C++26 Contracts runtime (P2900)	P3100	GCC 16.1 opt-in experimental [19]; Clang "No" [20]; MSVC "not yet" [17]	none
P3589R2 framework syntax	Profiles	public Clang build: framework + `std::init` slice, compile-time [105]; GCC in development	none
P3984R0 individual profiles	Profiles	none	none
libc++ assertion semantics	vocabulary	LLVM 21 [13]; names from P2900	vocabulary adopted; handler withheld from users [103]

Table C: Failure Mechanisms Deployed

No deployed hardened library constructs a `contract_violation` object or routes through a replaceable handler.

Implementation	Response	Customization	Stated reason
libc++ hardening	trap instruction	vendor only; not user-replaceable [33]	single-instruction codegen; code size [33][34]
libstdc++ assertions	diagnostic, `abort()`	not designed for replacement [16]	diagnostic quality [16]
MSVC STL hardening	`__fastfail`	termination function via macro [17]	corrupted process must not run handlers [17][35]
GSL 3.0	`std::terminate()`	throwing/unenforced modes removed [36]	Core Guidelines: violations terminate [36]

Table D: Precedents for Global Violation Handlers and Hooks

The surviving committee handlers own single-purpose protocols. The removed or condemned mechanisms either adjudicated a violation class program-wide or exposed a writable global slot.

Mechanism	Language	Outcome	Stated reason
`set_unexpected`	C++	removed C++17 [44]	"highly unlikely to Do the Right Thing" [43]
Annex K constraint handler	C	condemned (N1967) [94]	process-global; ill-suited for threads; security risk [94]
glibc malloc hooks	C	removed glibc 2.34 [95]	"eliminated a key exploit primitive" [95]
`set_terminate`	C++	survives	single-purpose protocol
`set_new_handler`	C++	survives	single-purpose protocol

Table E: The First Integration's Departures

The departures include the handling channel itself, not only the predicate evaluation.

Property	`contract_assert`	C `assert` integration (P3290R4) [45]
Predicate evaluation	const-ified	ordinary expression
Exception escaping predicate	translated to violation	propagates unchanged
Termination on handler exception	unwinds (enforce)	terminates

Table F: Criterion-by-Criterion Summary

On systematic coverage the P3100 model leads. On deployment, the shipped form is the named-guarantee set and both proposals' specifications are unshipped.

Criterion	Provenance	Finding
Deployment experience	party-proposed (P4297R0 Poll 3) [8]	named-guarantee form: decade of production; both specifications: none; applied evenly, leaves the base role unassigned
Existing practice	advisory (P2000R5) [5]	supports both proposals on different clauses
Systematic coverage	polled (Hagenberg, consensus) [25]	P3100 leads, but does not resolve ownership: both owners consume the enumeration identically
Guarantee and dialects	disputants' texts [1][7]	P3100's menu makes meaning configuration-selected across checkable operations; the compile-time `noexcept` question arises only in throwing configs; a throw-defining Profile raises the same question through undeployed definitions; P3100 did the design work
Single-architecture premise	deployment criterion applied	no deployment; removed/condemned precedents; first integration departs

5. Deployment and Field Experience: The Record Separates the Forms

This is the criterion P4297 asks EWG to weigh, so it comes first. Because both proposals have an undeployed specification and a deployed lineage, the comparison has to be stated carefully. The two are not the same thing, and this section separates them.

First, the record of what has shipped. For a decade, the named-guarantee form of safety enforcement - a vendor-defined set of named guarantees a build turns on - has shipped, with its cost measured at production scale in recent years. Its rows are tagged by check domain, because the domains differ and the comparison does not turn on blurring them:

- The C++ Core Guidelines checkers (domain: static analysis): clang-tidy has carried the `cppcoreguidelines-*` checks since LLVM 3.8 (March 2016) [11], and the MSVC Core Guidelines checker has been installed by default since Visual Studio 2017 [12].

- Hardened standard libraries (domain: library-precondition checking): libc++ has shipped hardening since LLVM 18 (March 2024) with four named modes [13], where a failed check is "reliably terminated" and hardening is an Xcode 16 build setting [14], deployed across Google server-side production at an average cost of about 0.30% [15]; libstdc++ has shipped `_GLIBCXX_ASSERTIONS` since GCC 6 (2016), on by default for unoptimized builds since GCC 15 [16]; and the MSVC STL has shipped hardening since Visual Studio 2022 17.14 (May 2025), where a failed check calls `__fastfail()`: "As C++26 Contracts are not yet implemented, this defaults to calling __fastfail() for hardened precondition violations." [17]

Consumers deploy the same form at vendor scale. Apple's WebKit ships libc++ hardening at its extensive level in release builds and reports, after optimization work, that "The end to end performance cost in WebKit has been zero" [75]. Firefox 145 grew a cross-vendor build flag that enables all three vendors' hardening macros "even in non-debug builds", motivated by Mozilla's own release-mode container checks having shipped "with negligible performance impact". The flag ships, and release defaults await performance testing [76]. And the form's failure response has already been exercised in a production fire: when GCC 15's default-on libstdc++ assertions met a live bug in Fedora Rawhide, dnf5 aborted on a `string_view` precondition with a diagnostic naming the exact failed check - triage material produced by the terminating response itself [77].

The rows span three check domains - static analysis, library preconditions, and, below, compiler-inserted core-language checking - so the unit of comparison comes first. The unit is not the domain. It is the configuration form: a named set of guarantees selected per build, the object P3589R2 and P3984R0 standardize. Across all three domains, the decade's deployments cover that form. Restricting the view to the core-language domain sharpens the asymmetry without dissolving it. In this record, the core-language deployments follow the named-guarantee pattern themselves. Android's UBSan checking ships as per-component build selections, and the response is fixed to abort [50]. Chrome's control-flow integrity is a build-enabled guarantee that kills the process [53]. Apple's `-fbounds-safety` is the newest production deployment in the lineage, a language extension "adopted on millions of lines of production C code", and its sole response is "deterministic traps" [74]. Where core-language checking reaches production, it arrives as a named, build-selected, response-fixed guarantee - not as in-source, per-assertion, handler-routed configuration.

That last fact makes the section's finding independent of the unit. Choose the configuration form, the check domain, or the raw mechanism lineage. Reject the discount rule below and count the kernel's reporting deployments as production. The rows move between columns, but one fact does not move: no deployment, under any counting, selects semantics per assertion in source or routes a replaceable violation handler. That fact is a claim about what no deployment does, not about a weighing rule, so it holds under every unit, and a reader who rejects this paper's weighing rules inherits it anyway.

That the named-guarantee form ships without C++26 Contracts is stated on the public SG15 (Tooling study group) mailing list. Gabriel Dos Reis, in the October 2025 discussion of P3835R0 [107]:

> *The point is that hardened standard library implementations are being delivered right now, today, for supported language versions, without being phrased in term [sic] of P2900, by GCC, Clang, MSVC, etc.* [21]

In the same discussion, on checking shipped beyond the standard libraries:

> *Organizations routinely ship running products (not debug, actual retail), with this https://github.com/microsoft/GSL , unsuppressed checking on; e.g. on* `gsl::span<T>::operator[]`. [22]

Second, the P3100 machinery provides capabilities the deployed form does not. It supplies a single vocabulary of evaluation semantics that maps existing mechanisms - `-ftrapv`, `-fwrapv`, and the sanitizers - into one model [1]; it enumerates the cases of core-language undefined behavior systematically; and it routes a failed check through the replaceable `std::contracts` violation handler, a single customization point the model proposes for all of them. These are real capabilities the model offers, and they are properties the deployed named-guarantee sets do not provide.

Against that, the specification record. The C++26 contract-violation runtime the P3100 model builds on (P2900R14 [18]) has one compiler implementation, GCC 16.1 (April 2026), opt-in under GCC's experimental C++26 label [19]. Clang reports "No" [20]. In June 2026, an upstreaming of a P2900 implementation began, gated behind `-fcontracts` and kept out of `-std=c++26` by its author's plan, with early users called "tester[s] in the early days" and the motivation stated directly: "the problems of contracts now is the lack of implementation experience and user experience" [67]. The MSVC STL states Contracts are "not yet implemented" [17]. Implicit contract assertions have, to the authors' knowledge, no implementation anywhere, and P3100R8 reports no deployment of the proposed machinery. Labels are future tense in the proposal's own text - they "will provide the ability to choose and constrain the evaluation semantic in code" [1] - and P3400R3 [2] has no compiler implementation of its core-language feature. The Profiles side now has a public, experimental implementation: the authors' own C++ Alliance Clang build, disclosed in Section 12, implements the P3589R2 [3] framework attributes and an initial slice of the `std::init` profile, released as regular public builds and enforced entirely at compile time [105]. Still in development is the GCC implementation. This is a partial implementation - one profile slice. P3970R0 [26] reports that, to its authors' knowledge, a specification and an experimental Profiles implementation is being conducted in a major C++ compiler. No complete implementation is on the public record. An available compiler is not production use, so it is counted here as no deployment experience, and the same zero applies to the other side's pipeline - the in-flight Clang contracts upstreaming and any future Labels prototype - until they ship and deploy.

Two symmetries follow. First, "no deployment experience" applies to both proposals' specifications: the field experience belongs to the named-guarantee form, and neither proposal's syntax has any. Second, both forms have a deployed lineage. The compiler-inserted check at an undefined-behavior site - the sanitizers, `-ftrapv`, `-fwrapv` - is the lineage of the P3100 model, and P3100R8 itself maps those mechanisms into its semantics [1]. The vendor-defined named guarantee set is the lineage of the Profiles model.

The two lineages do not carry the same kind of experience, and the difference motivates the discount rule this comparison applies: production deployment is the experience that counts, and a reporting mode documented as negating the mitigation counts as reporting, not defending. The rule is a corollary of the deployment criterion in the disputants' own formulations: Doumler's bar is "real deployment experience across different domains and companies" obtained by shipping "in production" (quoted below), Dos Reis's is "running products (not debug, actual retail)" (quoted above), and P3608R0's gloss is "very positive field experience reports" (Section 3). It also restates what the operators of the sanitizer family themselves say

separates defending from reporting.

Kees Cook, leading kernel hardening: the sanitizers "only WARN() by default", so "system owners need to set panic_on_warn=1 too if they want to defend against attacks targeting these kinds of flaws", and his production recommendation is bounds checking with "either use panic_on_warn=1 or CONFIG_UBSAN_TRAP=y" [54]. Evgenii Stepanov, creating the one UBSan runtime "suitable for use in production": "Primary mode for this runtime, as a security hardening tool, would be abort-on-error", with "no UBSAN_OPTIONS in general", "Definitely no C++ demangling", "No stack traces" - the production variant is defined by deleting the diagnostic machinery [71]. Kostya Serebryany, co-creator of AddressSanitizer, at CppCon 2018: "Generally speaking, you cannot use ASAN in production", and "It is also not a strong security mitigation. It is easy to bypass for the attacker" [72]. Sami Tolvanen, upstreaming the kernel's control-flow integrity: the permissive reporting mode "is helpful for locating type mismatches, but should only be enabled during development" [73].

In production, the named-guarantee deployments run with measured cost: on by default for unoptimized builds in GCC 15 [16], a product build setting in Xcode 16 [14], default across Google server-side production at about 0.30% [15].

The sanitizer and trap-flag lineage deploys as opt-in, test-time tooling in every deployment on this record except the production instances that follow, and in those the response follows the purpose. Deployed as a security mitigation, it terminates. Android ships UBSan integer-overflow checking in production components since Android 7.0 and UBSan bounds checking in the Bluetooth stack and eleven media codecs since Android 10, aborting on failure, with diagnostics mode documented as negating the mitigation [50]. Official Chrome ships control-flow-integrity checking that kills the process on violation, with diagnostics marked "not for production use" [53]. Deployed as a debugging aid, it logs: the major distribution kernels (Ubuntu, Fedora, RHEL) ship the kernel's UBSan bounds checking in reporting mode, which prints a diagnostic and continues, while the kernel's trap mode serves builders who want the mitigation posture at the cost of all reporting [54]. In this lineage, the one production deployment that both reports richly and continues execution is a sampler, and its own authors state the boundary. GWP-ASan guards a small random subset of allocations across production Android and Chrome, on Android 14 in a "recoverable" mode that delivers a full crash report and lets the app continue. Its paper states that it "is not a security mitigation tool due to its low detection probability" [70]: sampling telemetry riding on crash reporting instead of a checking response. Per purpose, both shapes fix the response in the build. Neither routes a replaceable violation handler.

The objects of standardization compare evenly against the deployed record. The Profiles framework standardizes named guarantee sets selected per build, a shape its lineage deploys. The P3100 model standardizes named presets as well - in its model a profile is a named configuration preset (Section 2) - and adds machinery its lineage has not deployed: in-source Labels, the replaceable violation handler, implicit assertions (the closest deployed relative, Bloomberg's in-house library-assertion family, is treated in Section 8). Both pair a deployed lineage with an unshipped specification. The distinction on this criterion is not which proposal owns the deployed form, since both can express the terminating named-guarantee set that ships, but that P3100's added machinery has no implementation. The one place a deployed library exposes the C++26 evaluation-semantic vocabulary, libc++'s experimental assertion semantics added in LLVM 21 (2025), selects the semantic through a vendor macro per build rather than an in-source Label per assertion, and keeps the handler override at the vendor level, out of users' hands [13]. Also on the record is the first library-side adoption of the menu itself. HPX merged contract-assertion infrastructure in July 2026 with enforce, observe, and ignore modes mapped to P2900's semantics, selected per build through a CMake option instead of in source. When the review thread reached the in-source

mandatory-precondition proposal P4044R0 [108], the author's reply was "pre! is a nice fit with where we landed" [69].

Table A (above, Section 4) records the deployment ledger for the two forms of runtime checking and the two proposals' specifications, from the vendor documentation and papers cited in this section. Neither proposal's specification has shipped. The forms differ in deployment character.

On the same public list, the proposal's authors have addressed the deployment question, and two statements from that thread bear on the criterion directly. Timur Doumler, arguing against a Technical Specification route, describes the experience as something still to be obtained:

> *Realistically, the only way to get that real deployment experience across different domains and companies is to put an initial feature set into the IS and have it ship in major compiler releases - otherwise those different domains and companies simply won't use the feature in production.* [23]

Joshua Berne, defending the observe semantic, points to deployment of contract-checking facilities at scale:

> *... we've also repeatedly demonstrated that multiple different groups that have actually deployed contracts at scale in real scenarios have had the need to be able to observe contract assertions in exactly this way.* [24]

Read against this section's distinction, both statements locate the deployment on the lineage axis. Berne's evidence is for contract-checking facilities of the lineage kind - library-level checks, where the post-violation state is language-defined - not for the C++26 machinery or its Labels. Doumler's argument locates the machinery's deployment experience in the future. Neither claims field experience for the implicit-contract-assertion machinery or for Labels - the machinery the layering would make the configuration base.

The section's finding: the named-guarantee form has a decade of shipped, production-default field experience, its cost measured in recent years; both proposals' specifications have none; and the C++26 contract runtime the P3100 model builds on has a single opt-in implementation from April 2026. What EWG makes of that record against the existing-practice standard is Section 6.

---

6. Existing Practice Reads Both Ways

The existing-practice sentence of P2000R5 asks which proposal builds on what already works, and its two clauses admit two readings. This section states both.

On the reading that "previous work" and "an existing feature" mean the deployed, field-tested practice, the Profiles model builds on it. The named-guarantee check-set is the shipping practice - a decade of it, across three vendors, measured in production - and a Profiles framework that standardizes named, enforceable guarantees is the same shape as what deploys today. The field-experience gloss on this reading comes from P3608R0 [6], which applied it in this exact domain: "the standard

library hardening is existing practice, and comes with very positive field experience reports." Dos Reis stated the same reading as a test on the public SG15 list, weighing a contract-phrased rendering against what already ships:

> *Any rephrasing in terms of P2900 should bring tangible benefits over what is available today. I see none.* [21]

On the reading that "previous work" means the most recently adopted work in the standard, the P3100 model builds on it: P2900 Contracts is in C++26, and P3100 extends it. This reading is available, and on P2000R5's literal sentence it is at least as direct - vendor hardening is practice, but it is not a standard feature, and P2900 is. What this reading cannot draw on is field experience: the adopted C++26 contract runtime has, as Section 5 records, a single opt-in implementation from April 2026 and no reported production deployment. Which of a deployed non-standard practice and an undeployed standard feature better satisfies "gradually building on previous work" is the kind of question the sentence does not answer by itself.

P2000R5's sentence has a second clause - "or by providing a better alternative to an existing feature" - and it points the other way. For that reading, the strongest candidate is the systematic framework of the P3100 model, and Section 3 weighs its strength under systematic coverage, where the P3100 model leads. This clause is not converted here into a separate ownership test, because whether one mechanism is a better alternative to the other is the judgment the ballot exists to make, not a record fact a comparison can settle. By contrast, the deployed-practice clause turns on vendor documentation. The difference in treatment is disclosed here rather than left for a delegate to find.

The two readings are not adjudicated here. That adjudication is the ballot P4297 asks for. What the record settles is narrower and is common to both readings: one rests on a decade of deployed field experience, the other on the most recent adoption, and the two point at different owners for the configuration of runtime checking. The ownership question therefore needs a decision of its own rather than a default - the direction sentence supports both proposals, on different clauses, and only an explicit weighing of the deployment evidence chooses between them.

---

7. The Guarantee and the Dialect Question

The two models assign the safety guarantee to different authors, and they expose per-configuration variation in what an expression does to different degrees. Through its menu of evaluation semantics, the P3100 model makes that variation configuration-selected across every checkable core-language operation. A Profile raises the same question only where a profile author defines a non-terminating alternative meaning. This section states the assignment difference, then the variation question in two dimensions, in the proposals' own words. First comes what an expression means at run time. The second dimension is what the compiler and the `noexcept` operator may assume about it at compile time.

The models assign the guarantee to different authors

Under the Profiles model, a profile defines the guarantee itself. P3984R0 [4] states the scope of that authorship directly: "A profile cannot change the semantics of a program beyond defining the meaning of some forms of undefined behavior." Its worked example gives the profile author the choice of meaning: "signed arithmetic overflow is UB so a profile can define it to be wraparound like unsigned arithmetic (though I wouldn't do that), to be saturated arithmetic, or to throw an exception." Under the P3100 model, a profile defines nothing. Per Section 4.4 it selects a configuration preset over the proposal's evaluation semantics. The difference is who writes the guarantee: the profile author writes it, or the preset selects it from the proposal's menu.

The runtime meaning becomes configuration-selected

P3100R8 [1] states a standard against dialects in its Section 4.3.2: "We also cannot have two different language dialects where the same expression means two different things (overflow or wraparound)." Its Section 5.4 then maps the same expression, signed integer overflow, across its five evaluation semantics (P2900's four plus the proposed assume), in the proposal's own words: GCC's `-fwrapv`, "which implements wraparound for signed integer addition using twos-complement representation, is a conforming implementation of the ignore semantic, silently executing well-defined replacement behaviour"; a sanitizer that prints a diagnostic "is a conforming implementation of the enforce or observe semantic (depending on whether the process is terminated or execution continues after printing the diagnostic)"; `-ftrapv`, which aborts, is a conforming quick-enforce; and today's optimize-on-the-assumption default is a conforming assume, where "if the predicate does not evaluate to true, the behaviour is undefined."

The expected answer to this observation is that evaluation semantics select the handling of a violation, not the meaning of the expression, so a correct program means the same thing under every semantic and no dialect arises. For the overflowing case, the proposal's own mapping resists that reading: under ignore, the overflow is not handled as a violation at all but executes "well-defined replacement behaviour" - wraparound, a meaning - while under assume the identical expression is undefined and optimized on the assumption it cannot occur. Between those two, what the expression means in the case the checking exists for is different, and so is the program's standing: the overflowing execution has defined behavior under ignore and undefined behavior under assume. Which programs have defined behavior is itself selected by the configuration. And the appeal to correct programs leaves out the executions the checking exists for. Whether that satisfies Section 4.3.2's own standard is a question the wording review does not reach, because no single undefined-behavior case raises it.

The answer has a stronger, normative form: correctness is defined by the predicate, not by the handling, so an overflowing program violates the implicit contract under every semantic - ignore does not make the program correct, it declines to check it. What that answer costs is the proposal's own mapping. P3100R8 maps `-fwrapv` as a conforming ignore "silently executing well-defined replacement behaviour", and the deployed programs built on that flag - the Linux kernel and PostgreSQL among them, quoted below - are written to wraparound semantics deliberately: their overflowing executions are the program working as designed rather than bugs the chosen configuration fails to catch. The normative answer must call every such program incorrect, forfeiting the deployed lineage the mapping exists to claim. The fork is narrow:

- Either ignore permits well-defined replacement behavior. Then the semantic assigns meaning, and the dialect question stands.

- Or it does not. Then the `-fwrapv` mapping fails, taking the ignore semantic's deployed lineage with it.

One horn keeps the lineage and concedes the dialect. The other keeps the purity and forfeits the lineage. Rewording Section 5.4 chooses a horn without escaping the fork.

The strongest committee form of the status-quo objection is already in print. P1407R1, responding to "Will this encourage breaking the language into dialects?": "To a certain extent, the dialects already exist. The existence of the -fwrapv and -ftrapv compiler flags testify to this." [78] The premise is correct. What answers the inference is how the deployed variation deploys: chosen once, globally, per binary. The Linux kernel pinned `-fno-strict-aliasing` in 2003 and holds the position tree-wide across two decades - "This is why we use -fwrapv, -fno-strict-aliasing etc. The standard simply is not *important*, when it is in direct conflict with reality and reliable code generation" [79]. PostgreSQL added `-fwrapv` to its configure line in 2008, after Tom Lane found GCC 4.3 "diking out a lot of security-critical overflow checks", and the flag has applied to every build since [80].

Abseil's operating rule is categorical: "all compile options that affect the ABI of a program need to be applied to the entire build on a global basis", with `-fexceptions` and `-DNDEBUG` among its named examples [81]. The largest one-knob experiment ran on exceptions: libstdc++'s own manual states that under `-fno-exceptions` "valid C++ code with exception handling is transformed into a dialect without exception handling" [82], and P0709R3 measured the fragmentation from that single binary toggle - over half of surveyed developers under partial or total exception bans, "using a divergent language dialect" [83]. Enforced by configure lines, style rules, and link-time mismatch errors, the deployed answer to existing variation is per-binary uniformity. What has no deployed record is the part the P3100 model adds: selection per translation unit and finer, per construct, across five semantics.

The implementer record on mixing includes a relevant statement. Dionne, on translation units built with different hardening modes: the program is "technically an ODR violation" and "you may end up getting either hardening mode". On the `-fno-exceptions` analogue: "I have seen this break in practice (this can result in e.g. libc++ calling std::abort() instead of throwing an exception that the caller intended to handle)". Since libc++ began encoding the configuration in ABI tags on its own functions, he reports no field incidents from mixed hardening modes - the mixing can be engineered safe. The bound is his own: the tags protect libc++'s own symbols through name-mangling machinery, they "do not propagate from callees to callers", user inline functions remain unprotected, and "it's not a full solution" [84] - machinery the core language, where implicit assertions would live, does not have. The status quo the objection appeals to is a record of programs refusing, per binary, the variation the objection cites.

The compile-time answer becomes configuration-selected

In a second dimension, the transformations the compiler may soundly apply to an expression, and the answer the `noexcept` operator gives about it, also become configuration-selected. Documented practice includes the optimization value of undefined behavior: Chris Lattner's account for LLVM records that undefined signed overflow licenses trip-count reasoning in loops - a wrapping definition "disables these important loop optimizations" - and that undefined null dereference "enables a broad range of optimizations" [51]. A checking semantic withdraws those licenses deliberately, which is what checking is for. The consequence for this comparison is narrower: which transformations are sound, and which expressions can throw, join the properties the configuration selects.

The following function appears in no proposal. It is written here to make the selection visible. Assume `g` and `h` are declared `noexcept`:

```
void f(int* p) noexcept {
    if (p) {
        g(*p);
    } else {
        h(*p);   // reachable only when p is null
    }
}
```

In today's C++ the dereference in the else branch is undefined on every execution that reaches it, so the compiler may delete the branch together with the test on the license of the dereference alone - locally, with no proof about callers - and know that nothing in `f` throws. Under a checking semantic of the P3100 model, the same dereference evaluates an implicit contract assertion, and the deletion license is withdrawn: absent an actual proof that no caller passes null, the check and its violation path must be generated. Because a throwing violation handler can be installed at link time, the dereference is also a potential throw site inside a `noexcept` function. The exception then meets `f`'s boundary and the program terminates, so the configuration incurs the exception scaffolding without obtaining the recovery a throwing handler exists to provide - the finding P3541R1 states directly, that the `noexcept` barrier "will cause `std::terminate()` to be called, and will compromise the recover-from-a-bug use case" [52]. Under the assume semantic, today's transformations return. Three distinct outcomes result: the branch deleted and nothing throwing, under assume as today; the branch generated and nothing throwing, under quick-enforce; the branch generated and the dereference a potential throw site, under a checking semantic with a throwing handler. Which one a given evaluation gets is selected by the same implementation-defined mechanism that selects the runtime meanings above.

The proposal confronts the compile-time half directly. P3100R8's Section 5.5 concludes that the addition of implicit contract assertions "must not affect the result of the `noexcept` operator", because changing that result "would be a breaking change to existing code and thus cannot be seriously contemplated", and it sets out four design options that preserve the operator's value [1]. The proposal adopts the first, Option A, under which a violation handler may throw and the exception propagates out of the expression while the operator returns its prior value. As the proposal states, Option A "does not change the existing behaviour of the `noexcept` operator - and thus does not constitute a breaking change - but it changes its conceptual meaning: rather than meaning 'evaluating this expression cannot throw', it now effectively means 'evaluating this expression cannot throw unless there is a contract violation'" [1]. Option B, which forbids the throw and calls `std::terminate` instead, "precludes unwinding the stack in response to an implicit contract violation, which has important use cases". Options C and D introduce non-throwing semantics and were not yet known when SG21 discussed the question [1].

Reached deliberately, the resolution is tallied in the public record: P3541R1, which first proposed Options A and B, "was discussed in SG21 and resulted in strong consensus in favour of Option A and against Option B" [1]. The recorded poll preferred "the noexcept operator to return true for such expressions, despite the fact that their evaluation can call the contract-violation handler which may throw" (SF 5, F 10, N 1, A 2, SA 1, consensus), and polled the truth-reporting option to

consensus against, in the committee's public paper tracker, January 2025 [85]. In a note to [except.spec], the proposed wording records the consequence: the evaluation of an expression that is not potentially-throwing "can nevertheless exit via an exception" when an implicit assertion's handler throws [1]. This section weighs the cost of that resolution rather than the care behind it.

The redefinition keeps the operator's answer stable by changing what the answer asserts: true no longer means the expression cannot throw, only that it cannot throw unless a contract is violated. Both properties of that arrangement were named, in advance, by the design's own architects. Berne, in the paper that restored the replaceable violation handler to the C++26 design: allowing the noexcept property "to either report an incorrect value (i.e., true for an expression that will actually throw) or vary across build modes would be a fundamental flaw in a Contracts facility for C++, opening the door to major problems being undiagnosable or even caused by the contract-checking facility" [86]. And P2969R0 - by Doumler with an author of this paper (Voutilainen) and Honermann, weighing the identical choice two years before the resolution - stated the redefinition itself: treating checks as non-throwing "would require us to change the meaning of the noexcept operator and deduced exception specifications: instead of answering the question 'can this construct throw?', they would answer the question 'can this construct throw if no contracts are violated?'" [87]. The property adopted for implicit assertions is the property the handler's architect called a fundamental flaw, in the redefined form the proposal's lead author had stated verbatim.

The machinery must then track the stronger fact the operator stops reporting. P2900R14 states the tracking obligation for contract assertions generally: where a potentially-throwing handler can be installed, "the compiler has to assume that any contract assertion in the program could throw an exception", because whether the installed handler is noexcept "is a link-time decision and is unknowable at compile time", and the compiler "has to generate the correct instructions for exception handling around every contract assertion" [18]. The one existing implementation confirms the mechanics. GCC's contracts code asks per condition whether the predicate "might throw" and wraps the check in a try-catch expression when it can. It wires the `std::terminate` machinery into existence even under `-fno-exceptions`, "because a contract failure may result in std::terminate call regardless of whether the exceptions are enabled". And it splices checks inside the internal must-not-throw wrapper of `noexcept` functions [55]. P3541R1, the paper P3100R8 cites for this problem, states the same fact from the code's side: today an implementation may emit a lock release "as a regular instruction, without caring if the instruction is also invoked upon stack unwinding", and under unannounced throws it no longer may [52]. Under the P3100 model, in any configuration in which a violation can throw, that obligation covers every checkable core-language operation.

The proposal's side calls this scaffolding negligible - P3591R0 argues that for inlineable non-throwing predicates "all exception-handling scaffolding will simply go away" and what remains is cold code [88]. The implementer record describes the same boundary differently: the open LLVM issue on `noexcept` codegen reports that once a throw meets the boundary "the backtrace is useless for any further analysis because the call stack was partially unwound", and libc++'s lead maintainer states both halves, "we fail to provide a relevant stack trace" and "these try-catch create some amount of code bloat" [89].

The implementation therefore holds internally an answer the redefined operator no longer gives: whether the expression, implicit assertions included, can exit via an exception. A second operator that told that truth would confirm the split rather than repair it. Where the operator's answer feeds a `noexcept` specifier, the consequence is bounded by termination, as in the example above. Where the answer feeds expression-level reasoning in unguarded code - cleanup elision of the kind

P3541R1's lock example shows, or algorithm selection on `noexcept(expr)` - the unannounced throw arrives where the reasoning said none could [52]. Deployed code consults the operator with measurable stakes. Giving one Bitcoin Core container class a `noexcept` move constructor and move assignment roughly doubled a vector-fill benchmark, because `std::vector` switched from copying elements to moving them [56]. ITK measured `std::vector::reserve` running "more than four times faster" from the same change [90]. And marking an `unordered_map` hasher `noexcept` was reported to lower Bitcoin Core node memory usage by roughly 9%, because libstdc++ selects its per-node layout on the hasher's noexcept-ness [91].

The strongest committee-record defense of the redefinition is built on these same stakes. P2969R0 states the zero-overhead rationale: treating checks as potentially-throwing means "the addition of a contract check to a program can lead to a different branch being taken at compile time", an undesirable property because it "can cause 'heisenbugs' and performance degradations even if the contract check has the ignore semantic" [87]. In this comparison's terms: a truth-reporting operator would flip deployed vectors from moving to copying the moment a checking configuration made moves potentially throwing, and the redefinition is what keeps them moving. Three observations answer that defense. First, it concedes the divergence: the argument for answering true is not that the expression cannot throw but that reporting the truth would change program behavior - convenience ranked above accuracy in a correctness operator. Second, the fast path must be weighed against what it protects: vector consults the operator to keep the strong exception guarantee sound during reallocation - move only when the move cannot throw, because a throw from a half-moved buffer has no rollback. The redefinition keeps the answer stable by making it inaccurate, in exactly the configurations where a violation can throw. So the dispatch keeps selecting the no-rollback path where its premise is false. The fast path survives, but the property it existed to protect does not. Third, the rationale as stated has no limiting principle: it argues that the operator should return whatever answer keeps existing code on its current codegen - a compatibility schedule rather than a property report. If a boundary separates this redefinition from any other answer chosen for compatibility, P2969R0 does not state it.

The alternatives that preserve the operator's unconditional meaning are also on the record. P3100R8 rejects changing the operator's result outright - having it answer that such expressions may throw - as a breaking change that "cannot be seriously contemplated" [1]. P3541R1's configurable option - which SG21 did not pursue - has the answer vary with the checking configuration, and states the consequence plainly: "a correct program can take different paths based on configuration" [52]. That is the dialect question again, now in the type system, and it is also the position a throw-defining profile occupies by default: under such a profile the truthful answer to `noexcept(a + b)` varies with the active profile.

On this compile-time dimension the P3100 side has done the design work: it identified the interaction, weighed the options, and adopted a resolution, so the criticism here concerns the cost of the resolution chosen rather than its absence. The breadth is what separates the two models here, and it is conditional: P3100's menu attaches the question to every checkable core-language operation, but only in the configurations where a violation can throw, while a Profile raises it only where its author defines a throwing meaning. A P3984 profile that defines overflow as a thrown exception [4] puts `noexcept(a + b)` in front of the identical question, owes the committee the same analysis P3100R8's Section 5.5 performed, and has not supplied it. A profile that instead fixes a defined replacement value such as wraparound or saturation takes the whole-program shape the deployed per-binary dialects already take, and raises no `noexcept` question. What avoids the question entirely is the terminating response: a check that traps cannot throw, and the trap-configured deployments that Section 5 records never raise it. The configurations that raise it are those in which a violation can throw - which Section 8 finds are also the contested and undeployed ones. The per-configuration-variation question therefore spans both what an expression means and what

the compiler and the `noexcept` operator may assume about it, and neither proposal's wording review answers it on the other's behalf. For that reason it belongs on a ballot of its own rather than being settled inside the wording review of either proposal.

---

8. One Architecture for Every Facility: The Deployed Record

The layering under comparison rests on an architectural premise: that one handler slot, one menu of evaluation semantics, and one configuration mechanism suffice as the base in whose terms every checking facility kept alongside it is specified - called here the single-architecture premise. The premise is not that the handler runs on every violation: three of the menu's semantics (ignore, quick-enforce, and assume) never invoke it. It is that the architecture as a whole is the base everything else is defined in terms of. P3100R8 describes "unified standard contract-violation handling" as a property of its design in its Section 5.2; its Section 4.4, quoted in Section 2, suggests defining a profile as a named configuration preset over that design [1]; and its Section 7.2 extends the arrangement forward, since a facility kept alongside the base must be specified in the base's terms [1]. This section applies the deployment criterion of Section 3 to the premise itself, in four parts. First, the failure responses that have shipped, with the reasons their authors state for the diversity. Second, the closest precedents the committee has itself standardized. Third, the first integration of a pre-existing checking facility into the C++26 contract-violation machinery. Fourth, what the unification claim asserts in the one place it is said to hold today.

The deployed failure responses and their stated reasons

The oldest failure handler in this paper's record shipped as a vendor default with local replacement, specified by no standards body. When MS-DOS encountered a critical I/O error it raised interrupt 24h. COMMAND.COM supplied the default handler - the "Abort, Retry, Ignore" prompt - and any application could install its own by replacing the interrupt vector, selecting per failure among ignore, retry, abort, and, from DOS 3.0, fail [27][28]. The dialog survives in the MSVC C runtime, where Microsoft's documentation states that choosing Ignore "can result in undefined behavior since preconditions of the calling code weren't met" [29]. The exhibit is an existence proof, not a preference argument: the violation response with the longest deployment record in this comparison ran four decades as a vendor construction with local replacement and a response menu, standardized nowhere, and never as a base other checking facilities were specified in terms of.

The sanitizers - the deployed lineage of the compiler-inserted-check form, per Section 5 - divide into combinations that ship together and combinations that cannot. UBSan combines routinely with AddressSanitizer, and the LLVM sanitizers share common reporting infrastructure. The shadow-memory sanitizers do not combine: AddressSanitizer, MemorySanitizer, and ThreadSanitizer are mutually exclusive, and their maintainers describe the exclusivity as architectural - "Instrumentations are not designed to work together and there are no plans to support it because of potential complexity of such implementation" - with the same thread dismissing a combined tool as "fantasy" [30]. Chandler Carruth, on the packaging of the runtimes in 2014: "many of the sanitizers' runtimes are really mutually exclusive and should never even be in the same runtime library" [31]. What no member of the family offers is the architecture under test here - a portable, user-replaceable violation handler spanning the family. Each tool's response is its own, configured its own way.

Even inside one tool, the pressure toward diverse responses operates: reviewing an addition to UBSan's failure-response options in 2015, Richard Smith observed, "I worry that we're piling on more and more ways of responding to ubsan failures

with no underlying principled design" [32]. Smith's remark asks for the principled design the unified model now proposes. The options it worried over - trap mode for code size, a full runtime for diagnostics, user-supplied handler libraries for specialized environments - are the field's answers to constraints this section catalogues, and the pile grew after the warning, with a minimal runtime for constrained targets added in 2017. Any principled candidate would satisfy Smith's ask: a named-guarantee architecture with local response ownership answers the worry as directly as a unified handler does, and the constraints that grew the pile are absorbed by the unified model only through absence - the quick-enforce finding later in this section.

The deployed funnels mark the same boundary. glog's process-global failure function unifies checking only across the libraries that adopt glog, and Qt's installable message handler delegates reporting across every Qt module while the fatal response stays Qt's own: "The message handler should always return. For fatal messages, the application aborts immediately after handling that message." [104] Reporting is delegable. Policy is not.

The three hardened standard libraries chose three different failure mechanisms, and each vendor has stated its reasons. The one customization among them - MSVC's substitutable termination function - is selected at compile time and receives no violation object [17].

Table C (above, Section 4) records the failure mechanisms of the deployed hardened standard libraries and the GSL, from the vendor documentation and maintainer statements cited in this section. None constructs a `contract_violation` object or routes through a replaceable violation handler. The third column records what each vendor allows instead.

These are recent, argued choices, and the customization that survives at the vendor level was argued for too. When libc++'s hardening work proposed dropping its override point, Chromium objected that crash-report control in the field depended on it, and the maintainers kept it - "this is something that we need to keep" [60]. The design line they state elsewhere is "simplicity over unbounded configurability" [60]. libc++'s maintainers, on why the failure handler is not link-time replaceable: "We've been there before and we basically tried all the possible different ways of customizing this before we ended up where we are. We tried link-time overriding, but that is too heavy when you need to generate a single instruction to trap" [33]. The deployment report behind that constraint records a two percent binary-size increase from the verbose failure path as a blocker in certain environments, reduced below half a percent by the trap [34]. Microsoft's mechanism is a kernel-level fast-fail because "critical failures that may have corrupted program state and stack beyond recovery cannot be handled by the regular exception handling facility", and on a fast-fail no exception handlers run because "the program is expected to be in a corrupted state" [35]. Both halves of Microsoft's framing belong in the record: the tracked plan is to report hardened-precondition violations through C++26 contract violations once MSVC implements Contracts, and that work is unshipped. What shipped bypasses even the CRT's customizable invalid-parameter handler, on the maintainer's stated security rationale - fastfail "gives attackers fewer opportunities to exploit running code", the "new guidance from our internal teams" that displaced the configurable-handler path of the 2005 `_SECURE_SCL` era [68].

Apple's feedback paper on P2900's handler states the same constraints in general form: on constrained deployments a violation must generate "no code at all beyond the equivalent of a branch and a `__builtin_trap()`", and constructing a `std::contract_violation` object is "akin to generating an RTTI-like structure for each individual contract predicate, which doesn't scale" [37]. The same paper states that "link-time customization is basically an invitation for non-benign ODR violation bugs" [37]. Link-time replacement is the customization model P2900 standardizes for the violation handler [18].

Section 5 records the one deployed adoption of the C++26 vocabulary, and the adoption was deliberate: libc++'s LLVM 21 assertion semantics take their names from the standard by design, and its deployment report presents the correspondence as alignment with C++26 [13][34]. The shape of the adoption is the evidence here: the semantic is selected through a vendor macro per build, and the handler override is reserved to the vendor [13]. The vocabulary unified, but the handler and its configuration did not.

Wherever checking ships at scale, inside and outside C++, the same divergence appears. The requirement that the application own the response is old committee record: Electronic Arts' 2007 account of why the game industry replaced the standard `assert` states that "there is no way to override it or intercept it and redirect it to application-provided facilities", and that "assert usage is verboten because it operates outside the application's control" [61] - a demand for application ownership of one facility's response, the per-facility shape every deployed handler in this section takes. The demand recurs: a 2023 RapidJSON issue raised the same objection to terminating assertions, and the accepted remedy was the library's own customization point, `RAPIDJSON_ASSERT` - answered locally, per facility, again [99].

LLVM routed all fatal conditions through one `report_fatal_error` channel for years and split it in 2025 into separate internal-error and usage-error functions, because one channel could not serve both a compiler bug and a bad command line [38]. Chromium maintains a family of checking mechanisms with distinct responses - CHECK, DCHECK, NOTREACHED, LOG(DFATAL), DLOG(FATAL), and ReportBadMessage - the last of which terminates the sending process rather than the receiving one, because a compromised renderer process must not be able to bring down the browser by sending a message that fails the browser's checks [39]. Rust deploys the closest thing to a standardized handler-and-menu design: a per-build choice between unwinding and aborting, and on freestanding targets one replaceable panic handler. Even there, context overrides the configured policy - a panic reaching a foreign-function boundary aborts regardless of the selected strategy [41] - and a co-lead of the Rust language design team summarizes the field: "virtually every network service I know of ships either with panic=abort or without really leveraging unwinding to recover, just to take cleanup actions and then exit" [62].

Automotive functional-safety standards state the independence constraint in primary sources. The EGAS monitoring concept standardized across German engine-control manufacturers requires that "System reactions are triggered independently of the function controller in case of fault", so that a fault in the function cannot suppress the response. ZVEI's 2025 guideline for ISO 26262 software requires the same independence at the highest integrity level on the unit [96]. ISO 26262's own text is not publicly retrievable, so the standard itself is not quoted here [42]. A response policy inside the failing process receives no independence credit under the retrievable primary formulations - which cuts against rich in-process handling under either model, where a bare trap escalates the decision to an external supervisor.

Where cross-facility unification does deploy, it sits on the other side of the trap. The unifier production systems actually share is the out-of-process crash reporter. Crashpad's design document states why: the crashed program "has often crashed because of corrupt global state - such as heap", so reports must be generated "with as little execution in the crashed process as possible", and the handler "runs in a separate process from the client" [97]. Its predecessor Breakpad states the operating constraint flatly - "Use of the application heap is forbidden" [97]. Production unification of failure handling happens after the trap, outside the process: the trap-plus-external-supervisor architecture the terminating deployments in this section presuppose. Nor is the direction C++'s alone: Go's team calls the one non-terminating boundary its ecosystem deployed at scale, the HTTP server's per-request recovery, "a historical mistake", and .NET, having operated a unified catchable-exception

architecture over corrupted states, withdrew it in stages until "recovery from corrupted process state exceptions is not supported" [98].

The closest deployed analogue of the unified model is Bloomberg's BDE assertion family, and it is real and scoped. BDE installs a process-global, replaceable violation handler that receives a violation object, and assertion levels select checking per build [40]. The family is two-tiered, and its own public sources draw the line. `bsls_review` logs and continues: its default handler logs the violation and returns instead of aborting. The component is "designed to allow apparently working production software, which nonetheless may harbor contract violations, to increase the number of precondition checks used to catch such bugs without negatively impacting the existing behavior of the software" - library-level checks, added to legacy code during migration, where the post-violation state is language-defined [106]. `bsls_assert` terminates: Bloomberg policy holds that "tasks may not install an assertion handler that returns control to the point immediately following the detection of a failed assertion", a policy `bsls_assert` "enforces ... by terminating the task" when a handler returns [106]. The boundary between the two tiers is Bloomberg's own, and it is the defined/undefined line: the reviewed tier continues where the post-violation state is defined, the enforced tier terminates where it is not. That boundary coincides with this comparison's terminating finding for the core-language-undefined class rather than cutting against it. That is a deployed handler-and-menu architecture - in the library-assertion lineage rather than the compiler-inserted one - and it is the deployment experience the Berne statement quoted in Section 5 points to.

Its evolution is instructive, and the primary account is Berne's own 2019 talk. Blanket, program-wide continuation as the default failed in production: teams "set a violation handler that didn't abort", new bugs went "only getting logged", bad data reached a client who "lost a lot of money", and the stated conclusion was that "having blanket continuation for violation handler is not a safe approach" [92]. The response separated the two tiers rather than removing continuation. The 3.15.0 release forbade return for the enforced tier: "bsls_assert failure handers [sic] must not return control to the point of the failed assertion (perhaps after having logged the event). Such attempts are now forced to abort" [57]. The reviewed tier kept it: `bsls_review` still logs and continues, and remains the sanctioned path for adding checks to defined-state legacy code [106]. What 3.15.0 removed from the enforced tier was the continuation, not the hook: the handler is still invoked and still logs. It may no longer return execution to the point past the violation.

The handler as per-check policy engine was built and abandoned: a configurable "smart" violation handler "did all sorts of great stuff but this required a lot of configuration", left "no way to indicate just in the code" that one assertion's treatment had changed, and "was rarely used ... never really took traction" [92] - the spoken twin of P2811R4's report that building review-mode behavior inside a handler "proved unsustainable and fruitless", where once the continuation policy could be specified on the annotation itself, a review implementation "became trivial to produce" [58]. And enforcement earned its caution operationally: one added check in the string destructor, "completely innocuous" to its authors, hit "hundreds of applications" and was pulled [92].

What that record is, stated in its strongest form, is design provenance. The C++26 architecture is built from BDE's lessons: the no-return rule became enforce's termination, bsls_review's staged log-then-promote migration - library-level checks, where the post-violation state is language-defined - became the observe semantic, and the in-place recategorization of individual checks, macro to macro at the check site, is creditable lineage for the Labels direction. Bloomberg has also rebased bsls_assert's enforced macro onto `contract_assert` under an experimental implementation and built BDE plus four large

internal libraries on the result [93]. Deployment-derived design is how good facilities get built, and nothing here discounts it as design input. What it is not is deployment of the premise under comparison: the distance between "the design learned from deployment" and "the base has been deployed" is the deployment criterion's entire content. The lessons were learned inside one in-house family serving its own assertions - policy per check, a reporting handler, one facility - and the learning institution's own trajectory moved policy out of the handler and into the check site.

Two boundary facts complete the scoping. The violation object BDE's handler receives carries the failed check's level - metadata classifying the check itself, whose nearest standardized analogue on the violation object, the `detection_mode` category enumerators, P3100R8 withdrew in favor of unimplemented Labels (Section 9) [40][1]. And the scope is one institution's assertion ecosystem. Public code search finds no installation of the BDE violation handler outside BDE itself, its forks, and its vendored copies. That search ran on 2026-07-11, and the indexes do not reach private or enterprise code. What BDE is not, and what nothing on the record is, is a base in whose terms other checking facilities are specified. Between a deployed in-house handler-and-menu family and a standardized base in whose terms every facility is defined lies the distance the premise asserts and the record has not crossed.

Replaceable violation handlers are in fact common in user space, and their record sharpens the scope finding instead of blunting it. Boost.Assert has exposed a user-defined, process-global `boost::assertion_failed` for two decades - the same shape as the C++26 handler, replaced at link time by the application - and the policies deployed projects install in it differ: QuantLib throws, OSRM throws specifically to guarantee stack unwinding, userver logs and aborts with a stack trace, RethinkDB routes into its own fatal-error machinery [59]. The canonical use of such handlers includes throwing - the maintainer of libassert steers assertion-testing through a throwing handler [59]. And when the GSL deleted its response configuration - the throwing and unenforced modes, removed in 3.0.0 [36] - users from safety-critical, application-monitoring, and interactive environments spent years on the record objecting that the termination decision was not the library's to make. What no deployed handler spans is facilities: each serves exactly one assertion family's checks, and one process routinely runs multiple policies side by side - the same program can throw from `boost::assertion_failed` while its C `assert` aborts and its hardened library traps. The deployed record is per-facility handlers under application ownership. Expressing that policy matrix through one cross-facility handler would need exactly the per-check category metadata whose standardized form was withdrawn in favor of unimplemented Labels (Section 9). The record of handler customization is therefore an argument for facility-scoped response ownership rather than for a single base every facility routes through.

The committee's own precedents

The committee's own record contains both surviving and removed process-global handlers, and the difference between them is the instructive part. `std::set_terminate` customizes how an already-chosen termination happens, and `std::set_new_handler` runs a resource-exhaustion protocol - free memory and retry, throw, or terminate. Both date from C++98 and remain in the language. The one that adjudicated a violation of a language-level annotation through a replaceable global function - `std::set_unexpected`, the handler for dynamic-exception-specification violations - was deprecated in C++11 and removed in C++17 with the feature it served [44]. Beyond the handler, the removal had more causes: the specifications were enforced at run time rather than compile time, composed poorly, imposed cost, and were superseded by `noexcept` [44]. What the handler contributed is stated in Herb Sutter's assessment: "A global handler is highly unlikely to be smart enough to Do the Right Thing for any given particular case, and the result is to go to terminate(), go directly to terminate(), do not pass catch, do not collect $200" [43]. The precedent is therefore narrow, and it is the relevant one: the surviving global handlers own single-purpose protocols. On Sutter's assessment, the one asked to respond usefully to a class of violations on behalf of the whole program was unlikely to manage it, and maximal extensibility - any function could be installed - did not save the feature it served. The precedent's reach also tracks the machinery's observability. In configurations where the handler only reports on the way to a fixed termination, the adjudicating architecture is absent and the precedent does not apply. In configurations where the handler adjudicates the program-wide response - the throwing and continuing ones - the precedent applies, and those are the same configurations this section finds contested everywhere else.

The C committee's record contains the same precedent, run to the same end. Annex K's bounds-checking interfaces route every constraint violation through one process-global runtime-constraint handler, and N1967, the field-experience paper behind the annex's proposed removal, states its findings in the shape this section has been cataloguing: "virtually every function in the Bounds checking library relies on a single process-global runtime-constraint handler", so "the APIs are ill-suited for multi-threaded components". And the replaceability is itself the security concern, a user-installed handler that recovers or defers termination "increases the window of opportunity for an attacker to gain control of the vulnerable program" [94]. The committee record thus holds two standardized process-global handlers charged with adjudicating a violation class program-wide, in two languages, both condemned by their committees' own review. The platform layer moved the same direction: glibc removed its writable global malloc hooks, the removal having "eliminated a key exploit primitive from the library", and Microsoft's CRT added a thread-local variant of its invalid-parameter handler - one global slot does not compose, even within a single facility [95].

The first integration departs from the unified semantics

The first integration of a pre-existing checking facility into the C++26 contract-violation machinery is on the record, and it does not use the unified semantics. P3290R4 [45] (Berne, Doumler, Lakos) proposes a library API for invoking contract-violation handling directly and an opt-in mode in which the C `assert` macro reports its failures through the contract-violation handler. Its abstract states why an integration layer is needed at all: pre-existing facilities "occasionally have fundamentally different semantics and provide completely different control mechanisms for their behavior". On three axes, the integration departs from the semantics of `contract_assert`: the `assert` predicate is evaluated as an ordinary expression, without the const-ification applied inside contract-assertion predicates; an exception escaping the predicate propagates unchanged rather than being translated into a contract violation; and termination keeps the C facility's abort-on-failure behavior - the proposal describes invoking the handler "before aborting". One API variant goes further, terminating when an exception escapes the violation handler rather than letting it unwind as the enforce semantic does [45].

That last departure sits inside the one thing the integration claims to unify. Available for the first two - deliberate preservation of the C facility's documented evaluation semantics, with only the handling channel unified - the defense does not reach a variant whose divergence is in the handling. An exception escaping the violation handler terminates under the integration where the enforce semantic unwinds, so the unified channel itself behaves differently depending on which facility's check invoked it. P3912R0 states the general form of the divergence: always-enforced checks "represent a fundamentally different feature from build-time configurable contract assertions. They target different use cases, lead to different deployment scenarios, and follow different adoption trajectories" [46]. P3100R8's Section 7.2, quoted in Section 2, names "an incoherent and messy design" as the failure condition when two mechanisms coexist without one being specified in the other's terms. The first integration exhibits that divergence before either proposal has shipped.

Where the unification is said to hold, it holds by absence

What remains is the one place the unification is said to hold today: standard library hardening, adopted for C++26 with wording phrased as contract assertions evaluated with a checking semantic [48]. The phrasing asserts less than it appears to, and the reason is the quick-enforce semantic itself. P2900R14 specifies that "quick-enforce will not call the contract-violation handler but will instead immediately terminate the program" [18]: under that semantic no violation object is constructed and no handler runs. Two features of a contract assertion do survive under quick-enforce for a general predicate: the predicate is evaluated const-ified, and a predicate that exits via an exception terminates the program. But the hardened preconditions adopted for C++26 are comparisons on values the function already holds - predicates that const-ification does not affect and that do not throw (the exceptions: two span constructors whose predicates invoke user-defined iterator and range operations) [48]. For such a non-throwing predicate, the following code - which appears in no proposal or implementation, and is written here to make the equivalence inspectable - is everything a conforming quick-enforce evaluation requires:

```
if (!cond)
    __builtin_trap();   // or abort(), or __fastfail()
```

Nothing in it is a contract assertion, and nothing in it changes if a specification calls it one. Read the vendor statements against that equivalence. libc++'s deployment report states that "the trapping mechanism used in libc++ hardening is precisely the quick-enforce evaluation semantic" [34]. The MSVC STL documents that "as C++26 Contracts are not yet implemented, this defaults to calling __fastfail() for hardened precondition violations" [17]. Both statements are accurate, and both describe implementations containing no contract machinery, because for these predicates, under quick-enforce, conformance requires none. P3878R0 reaches the same conclusion from the implementer's side, describing what a hardened implementation that must always terminate is left to do: "not use contracts the language facility in its code, and just terminate directly, without invoking any violation handler" [47]. Where the unification is claimed to hold, conformance to it is satisfied by absence.

The confinement is general, and it covers both integrations on the record. The two semantic features that would distinguish the machinery even without a handler - const-ification, and the translation of predicate exceptions into violations - are exercised by neither: hardening's predicates, outside the two span-constructor exceptions noted above, never reach them, and the C `assert` integration omits both by design [45]. Across the two integrations, the unified semantics have never been exercised by a facility joining the framework: the integration that would have exercised them departed from them, and the deployment said to demonstrate them almost never reaches them.

The equivalence is confined to quick-enforce, and the confinement is what matters. Under enforce the handler does run - the hook, invoked before a fixed termination - and Section 5 credits what that provides: one customization point for reporting across every facility that routes through it. The hook and the continuation are separate properties, and what enforce provides is the hook without the continuation. Telemetry does not require the contract handler either: the sanitizer runtimes of Section 5 and the out-of-process crash reporters recorded earlier in this section both log violations without one.

But the configurations in which the unified machinery becomes observable are the contested ones. A checking but non-terminating semantic on a hardened precondition is the possibility P3878R0 identifies as defeating a hardened implementation's purpose [47]. And whether the violation handler is replaceable at all is implementation-defined: P3846R0 lists it among five deliberately implementation-defined properties, replaceability being "viewed by some platforms as a necessary feature for even the most basic viability of the proposal and by other platforms as a security risk" [49]. That accommodation of the constraint Apple's paper states has a consequence: the one customization point credited in Section 5 is itself only conditionally present. For the terminating configurations that production hardening deploys today, a profile defined as a named preset over evaluation semantics [1] and a profile that defines the violation response directly, as P3984R0 does [4], are observably identical.

The first field migration onto the machinery itself - to the authors' knowledge the one sustained public account, run on an experimental implementation - points the same way: ScyllaDB selected enforce, the handler-running terminating semantic, and pinned it per assertion through a vendor attribute, "Scylla is always enforced. Always.", foreclosing the per-build menu [63]. The pinning is de-configuration, not configuration - a per-assertion guarantee, chosen by the code's owner, that no build may override - and it predates the migration: ScyllaDB's tree-wide `SCYLLA_ASSERT` macro "is always defined and is not conditional on NDEBUG" [100]. The engineer who ran the migration states the design misfit on the public reflector: "the person who write the contract is not the one who selects the semantics for the application. Is this aspect of contracts aligned with hardened libraries needs? The discussion seems to reveal that not." [101] Redpanda reached the same endpoint

independently, replacing build-conditional asserts after "at least 2 difficult-to-analyze bugs" were introduced because checks were "compiled out due to our use of -DNDEBUG" [100]. Where configurability itself was the bug vector, the fix in both organizations was to remove the menu rather than enrich it. The implementer statement of the general form is John Spicer's, from the SG21 reflector: the key issue "is how you apply various 'checking modes' to different components (e.g., libraries)", and "this is a problem that needs to be solved in order for contracts to be viable in the real world" [102] - named, scoped, component-attached guarantees. The in-source pinning ScyllaDB reached for is the capability Labels propose and have not shipped.

The same standard applies on the Profiles side: P3984R0's non-terminating definitions - overflow as saturation, or as a thrown exception [4] - are just as undeployed as the machinery this section examines, and Section 5's matches-deployed-form finding covers the framework's named-set shape without reaching those definitions. What deploys today fixes its response in the build - terminating in the mitigations, logging in the kernel's diagnostics - and gives the violating case no defined alternative meaning under either model's vocabulary. The layering's distinct content appears only in the contested configurations, and Section 7 records the meaning question those configurations raise.

The section's finding, in four parts, one per subsection. First, deployed failure responses are diverse for stated engineering reasons: single-instruction code generation, distrust of in-process handlers in corrupted states, process topology, mechanism independence - and the closest deployed relative, BDE's in-house handler-and-menu family, serves its own assertions rather than other facilities. Second, the C++ committee's one global handler charged with adjudicating a violation class for the whole program was removed with the feature it served, while the single-purpose handlers survive. The C committee's equivalent was condemned by its own field-experience review. Third, the first pre-existing facility integrated into the contract-violation machinery departed from the unified semantics on three axes. Fourth, the deployment claimed for the unified model conforms because, for hardening's predicates, quick-enforce requires none of the model's machinery. No deployment of the single-architecture premise - the base every facility is specified in terms of - exists on the record. On the deployment criterion, the premise the layering presupposes is untested where it is not already contradicted.

---

9. What Configuration Ownership Costs in Practice

The layering is not only a question of order on a diagram. It has already had a concrete consequence in the wording of a companion paper. When one feature owns the configuration mechanism, a change to that mechanism can strand another paper's wording through independent revision. That has happened once.

P3081R2 [9] (Sutter, February 2025, the latest revision) proposes wording that adds three enumerators to the `detection_mode` enumeration - `detection_mode::type`, `detection_mode::bounds`, and `detection_mode::lifetime` - each defined so that "the contract assertion was evaluated as part of" the corresponding profile, and its mechanism passes "the detection_mode value corresponding to P" on a failed check. Those values were to be produced by P3100's machinery: P3543R0 [10] (Gill, Jabot, Lakos, Berne, and Doumler) states that P3081's runtime checks "are already preconditions introduced as implicit preconditions into the language itself by [P3100]."

P3100R8 [1] then withdrew that producer. Its Section 5.6:

> *... unlike earlier revisions of this paper and unlike [P3081R1], which adopted its library API from those earlier revisions, we no longer propose to add new enumerators to the enumeration detection_mode to encode the category of error (Initialization, Bounds, and so on); instead, this encoding can be accomplished more effectively and flexibly via Labels (see Section 7.1).*

The two papers are now inconsistent in the record: P3081R2's proposed wording still adds the `detection_mode` enumerators, while P3100R8 proposes none and routes category encoding through Labels instead. No later P3081 revision has reconciled the two. None of this is a feature breaking another feature. Both are unadopted proposals, and the change has a legitimate design rationale. It is coordination cost: when the P3100 proposal moved category encoding to Labels, P3081R2's enumerators lost the mechanism that would have produced them, and the dependency sat unreconciled through the next revision cycle. P3100R8's own Section 7.2 anticipates that cost when it states that, if both features are kept, one must be specified in terms of the other. Configuration ownership decides which paper absorbs that cost when either side revises.

The episode is also evidence on the base question itself, all of it stated here. First, a Profiles-side paper adopted the P3100 arrangement, taking the contract machinery as the producer of its library API. The episode shows the cost even a willing adopter pays when the base revises underneath it - and the framework papers this comparison examines, P3589R2 [3] and P3984R0 [4], do not adopt that arrangement.

Second, the episode is an inconsistency inside the Profiles corpus too: P3081 adopted the arrangement, P3589R2 and P3984R0 arrange the pieces the other way, and by Section 7.2's own standard the Profiles papers are currently unspecified in each other's terms. The exposure itself is not one side's property - whichever feature is made the base, a revision of the base can strand any paper specified in its terms. What the record adds is that the cost is no longer hypothetical: it has been paid once, by a willing adopter, without a published reconciliation, and the ownership decision assigns who absorbs it next time.

Third, the exhibit is time-stamped, not time-decaying: a later P3081 revision that drops the enumerators would absorb the base's change, and the episode stays on the record. Whether reconciling borrowed wording is the borrower's job or the lender's relocates the cost without disputing it. The finding concerns the cost's existence, and it assigns no blame for the delay.

---

10. Objections

Each heading below states an objection in its strongest form. Each response draws only on evidence already presented.

"The two features are complementary, not competing, so there is nothing to own"

Complementarity is not symmetric here, and the proposal says so. P3100R8 Section 7.2 states that for granular in-source control "we need to agree whether this happens" through Labels or through the Profiles framework, and that "If we want to have both, we need to specify one in terms of the other to avoid an incoherent and messy design" [1]. There the proposal itself states that one feature must be defined in the other's terms. A divided-territory version of the objection - each feature primary in its own sphere - collapses at the collision points, as Section 2's partition analysis records. Section 9 shows the cost of leaving the question unsettled: one paper's wording has already been stranded by the other's revision. Complementary features that must be specified one in terms of the other still have an ownership question, and it is the one compared here.

"The Profiles framework is not implemented either, so the deployment criterion is neutral"

Correct about the specifications, and Section 5 states it: neither P3589's framework syntax nor P3100's Labels has a released compiler implementation. The deployment evidence is about the forms, and there the criterion is not neutral, for the reason Section 5 states as its discount rule: the named-guarantee lineage runs in production with measured cost, the inserted-check lineage as opt-in test-time tooling outside its enumerated production instances. Applied evenly the criterion still separates the two objects: both standardize an unshipped specification over a deployed lineage, and the surviving asymmetry is that P3100's added machinery - Labels, implicit assertions, the replaceable handler - has no implementation.

"The decade you count is library hardening and static analysis; deployed runtime checking of core-language UB is our lineage, the sanitizers and trap flags"

The strongest form of the deployment objection, and its factual half is stated in Section 5: the compiler-inserted check at an undefined-behavior site is the P3100 model's lineage, and P3100R8 maps those mechanisms into its semantics. What the objection does not supply is equivalence of experience. The sanitizer and trap-flag lineage deploys as opt-in, test-time tooling outside its production instances, and those instances fix a handler-free response in the build - terminating where deployed as mitigation (Android's UBSan checks, Chrome's control-flow integrity, Apple's `-fbounds-safety`), logging where deployed as diagnostics (distribution-kernel UBSan reporting) [50][53][54][74]. The named-guarantee lineage deploys in production with measured cost, at operating-system-vendor and hyperscaler scale. And the direction of standardization differs: the P3100 model does not propose to standardize `-fwrapv` and the sanitizers as they deploy today - it proposes Labels, implicit assertions, and the replaceable handler, none of which its lineage has deployed - while the Profiles model proposes to standardize named per-build guarantee sets, which is what its lineage deploys. As for the aggregation half - the charge that the decade's rows are static analysis and library preconditions rather than core-language checking - Section 5 tags the rows by domain and states the unit of comparison: the ledger is organized by configuration form rather than check domain. Restricting the view to core-language checking strengthens the finding, because the core-language deployments on the record are themselves named-guarantee deployments - per-build, response-fixed, the newest of them trap-only by design. Table A carries both halves.

"No feature can have deployment experience before it ships in the IS, so the criterion structurally privileges incumbency"

This is the normative force of the Doumler statement quoted in Section 5, and it is a fair description of how new core-language features arrive: constexpr, concepts, and coroutines shipped on design and implementation experience, without production deployment behind them. The criterion as applied here does not demand pre-adoption deployment of anyone's syntax. It weighs the deployed forms, which both sides have, and asks what each proposal adds beyond its deployed lineage - a question that is answerable today and that the ballot proposed in P4297 would put to the room with this evidence in front of it. A criterion both sides can satisfy in the same way is not incumbency protection. The asymmetry is in the record, not the rule. As Section 5 states, the finding also survives the objector's own unit: under form, domain, lineage, or a no-discount counting, no deployment on the record selects semantics per assertion in source or routes a replaceable handler.

The criterion's two applications obey one separating principle: deployment evidence transfers across spellings of a deployed form, but not to architectures that have never existed anywhere. When it standardizes a form the field already runs, a proposal's syntax needs no pre-adoption deployment - the spelling is new, the thing is not. The single-architecture premise is not a spelling of anything deployed, so there is no deployed referent for evidence to transfer from. Section 8 reaches that finding without applying a stricter rule to premises. The sharpened form of this objection - that the criterion so applied would have rejected P2900, constexpr, and concepts - dissolves on the same distinction: those adoptions added new capability with no incumbent deployed form on either side, so there was no displacement question to answer. Here a deployed form exists and displacement is the question. The P2900 precedent, read closely, runs the other way: that minimum-viable-product scope was limited to explicit assertions at API boundaries, with BDE-class design experience behind it (Section 8). P3100 enlarges the scope to every checkable core-language operation plus the base role over other facilities - a broader claim on a thinner record. Finally, the structural variant - only a standard can create a cross-facility base, so demanding a deployed one demands the impossible - is an argument about sequencing rather than default ownership: adopt the checking without the base role, and assign basehood when a record exists. Impossibility of prior evidence argues for deferring the assignment instead of making it by default.

"The discount rule and the matches-deployed-form test are the authors' inventions, carrying no provenance"

The two weighing rules that produce Section 5's finding deserve the same provenance audit as the criteria, and Section 5 supplies it. The discount rule is a corollary of the deployment criterion in the disputants' own formulations - Doumler's production-use bar, Dos Reis's "actual retail" - and it restates what the lineage's own operators say separates defending from reporting: Cook's panic-or-trap production recommendation, Stepanov's production runtime defined by deleting diagnostics, Serebryany's "you cannot use ASAN in production", Tolvanen's development-only permissive mode. Because it repeats what the record's operators practice, the discount rule is sourced to the record rather than invented over it. The matches-deployed-form comparison is the existing-practice criterion applied evenly: P2000R5's sentence asks what builds on previous work, and asking what each proposal adds beyond the deployed lineage is that question, phrased so vendor documentation can answer it. Applied evenly it settles the base role for neither (Section 3); what survives is the factual asymmetry, that P3100's added machinery is the unshipped part. Rejecting either rule, a reader still holds Table A's rows, which stand independently of the weighing. The re-weighing invitation in Section 11 covers that reader by name.

"Weigh the criteria by their provenance, and the one polled criterion decides for P3100"

The weighting move is legitimate, and Section 3 accepts its premise: the criteria differ in provenance, and systematic coverage alone carries a poll. What the move cannot deliver is an ownership verdict, because the polled criterion does not discriminate between the candidate owners - a profile and implicit assertions guard the enumerated cases identically (Section 3) - and the poll's own text assigns the configuration of neither feature to either. Weighting it dominant raises the P3100 model's total on the criterion it already leads, a lead granted above, and leaves configuration ownership unmoved.

"The parts are deployed; the systematization is the contribution, so deployment of the parts is deployment of the whole"

The parts and the organizing work earn credit here: the enumeration of the undefined-behavior cases (Section 3) and the vocabulary that maps deployed mechanisms into one model (Section 5) are real contributions, stated as such. At the step from parts to whole, the objection fails: the parts ship without the elements the whole adds. The sanitizers, trap flags, and hardened libraries deploy no violation object, no replaceable cross-facility handler, no in-source selection among evaluation semantics (the deployed in-source control is binary suppression, per-site check-or-not attributes), and no translation of predicate exceptions into violations - those are the whole's additions, and Section 5 finds each of them undeployed. Making the distinction concrete, the one deployed adoption of the model's vocabulary is libc++, which took the names without the machinery (Section 5). The organizing work could have been standardized over the deployed forms as they deploy. What is proposed instead is the layer no part of the lineage includes, and naming deployed parts does not lend them the whole's deployment record.

"Quick-enforce accommodating the trap constraint is a feature of the architecture, not a weakness"

Accepted - and the feature has a cost the objection does not name. Accommodation by quick-enforce is accommodation by absence: Section 8 shows that for hardening's predicates a conforming quick-enforce evaluation contains none of the machinery, so where the architecture accommodates the deployed constraint, specifying a profile in its terms adds nothing observable - a preset selecting quick-enforce and a profile defining termination directly are the same profile. Feature and vacuity are one fact seen from two sides: a feature that works by containing nothing cannot also be the thing other facilities are specified in terms of. The ownership claim must then be earned entirely by the configurations in which the machinery is present, and those are the contested ones: the non-terminating semantics P3878R0 finds defeat a hardened implementation, the replaceable handler Apple's feedback paper treats as a security risk [37], and the noexcept interaction of Section 7. Where the accommodation operates, the two models are observably identical. The base question can only be decided where they differ, and where they differ is where the objections sit.

The zero-overhead restatement of this objection - the design achieves its stated goal of costing nothing, and calling that vacuity renames an achievement - changes neither half: the achievement is real, and an architecture whose presence is unobservable in the deployed configurations still cannot ground an ownership claim there. Nor does the vocabulary version - libc++ adopting the standard's semantic names is standardization working - survive the adopters' own account, which Section 5 cites and the sources state. The semantics "closely mimic C++26 Contracts evaluation semantics". Their selection is a configure-time knob whose default mapping is that "production-capable modes map to quick-enforce (i.e., trap)". Already a

year and a half earlier, the handler override had been withdrawn from users, the maintainers writing "we feel that this should primarily be controlled by vendors, not users" [103]. Names adopted, menu vendor-held, handler withheld: vocabulary standardization without basehood, in the adopters' own words.

"The contract-phrased specification offers option value: capability can grow later without respecification"

The strongest analytical form of the previous objection, and its premise is true: wording phrased as contract assertions can grow enforce-with-handler reporting, per-assertion Labels, or violation telemetry without anyone rewriting the hardening wording, while a bare trap specified as a bare trap grows nothing. Four findings weigh the option. First, the capability it would unlock - handler-running, non-terminating checking - is what production deployments avoid and hardening implementers reject: the one migration's engineer states that builder-selected semantics do not fit hardened libraries' needs, Redpanda pinned always-on after configurability itself shipped bugs, and P3878R0 finds the non-terminating semantics defeat a hardened implementation's purpose (Section 8). Second, holding the option is not free: Section 7's overheads - the tracking obligation, the exception scaffolding, the redefined operator - are incurred in every configuration in which a violation can throw, whether or not the capability is ever used. Third, the option is severable from ownership: hardening was adopted contract-phrased while the base role stayed unresolved (Section 8), so the committee demonstrably can hold the option without assigning the base. Fourth, the record already shows what happens when the wording actually grows: it has grown once - the C `assert` integration - and that one growth required three departures from the unified semantics, one inside the handling channel itself (Section 8). Together the four argue for keeping the wording flexible rather than for assigning the base by default.

"SG21 settled the noexcept interaction deliberately; re-opening it through an EWG paper is venue-shopping"

The premise is accurate and Section 7 credits it: the resolution was reached after weighing the options, polled, tallied, and recorded. What the objection misses is what the polls themselves say about the question's address. SG21's remit is the Contracts facility, and its polls on this question were expressly contingent on a design that belongs to another room: each poll's lead-in reads "If P3081 Profiles add implicit contract checks to core language expressions such as pointer dereference or array indexing" [85]. The study group answered for the machinery it owns, conditioned on a core-language decision it does not. Implicit contract assertions attach that question to every checkable core-language operation, so `noexcept(a + b)` becomes a question about the checking configuration in every expression, for every user, whether or not they write contracts. And the identical question now binds a proposal that was never in SG21's room: P3984's throw-defining profile faces the operator with no SG21 resolution covering it (Section 6). A question whose own poll text conditions it on another room's decision, and which now attaches to both competing proposals, sits in EWG because of where it lives, not who raised it - and Section 6 weighs the resolution's cost while crediting the diligence behind it to the P3100 side.

"A correct program never reaches a checking point, so the configuration-selected variation is hypothetical"

Two findings already in the body answer this. First, the compile-time costs are borne by violation-free executions: within any configuration in which a violation can throw, the branch-deletion license is withdrawn and the exception scaffolding is generated whether or not any violation occurs (Section 7). A variation in what the compiler may do with correct code, and in what `noexcept` tells correct code, is not hypothetical in the executions the objection appeals to. Second, the premise concedes the dialect finding rather than answering it: the variation it calls hypothetical is the fate of the violating executions - the executions the checking exists for - and whether those have defined replacement behavior or undefined behavior optimized against is what the configuration selects (Section 7). A facility whose configured meanings differ only in the violating cases is not dialect-free. The variation sits where the checking is load-bearing.

"The diversity of deployed failure responses is history, not design; the unified handler is the improvement"

The direction of the record is the opposite: the diversity is recent, argued, and re-derived under pressure. libc++'s maintainers tried link-time handler replacement and withdrew it on measured code-size grounds [33][34]. The GSL shipped configurable violation responses and deleted them in 3.0.0 [36]. LLVM operated a unified fatal-error channel for years and split it in 2025 [38]. The automotive safety standard denies independence credit to response policy inside the failing process [42]. Improvement over deployed practice is a legitimate ambition, and Sections 5 and 8 credit the capabilities the unified model would add. What the objection does not supply is a test the premise has passed: the one standardized global handler charged with adjudicating a violation class for the whole program was removed with the feature it served [43][44], and the deployment criterion of Section 3 - proposed by this paper's authors, as disclosed there - asks for the evidence before the commitment.

"The unified handler is designed to be extensible; new facilities extend it rather than diverge from it"

The extension points are vocabulary - enumerators on the violation object, Labels when they exist [2] - and the recorded divergences are not vocabulary. A check that must compile to one instruction [33], a failure path that must invoke no in-process handler [35], a response that must terminate a different process than the detecting one [39], and a mechanism that must stay independent of what it monitors [42] are constraints on the response architecture, and no enumerator or Label reaches them. Through quick-enforce, which conforms by using none of the machinery (Section 8), the proposal accommodates the first two. The extension record where it has been exercised: the `detection_mode` enumerators for profile categories were withdrawn by the P3100 proposal's own revision, stranding P3081R2's wording (Section 9) [1][9], and the C `assert` integration arrived with the const-ification, exception-translation, and termination departures Section 8 records [45]. Extensibility of the handler was the one property `std::set_unexpected` possessed in full, and the feature it served was removed regardless [43][44]. The forward-looking form of this objection - choose the architecture that enables the most future work - presupposes the architecture's fitness for facilities that do not yet exist, and the direction paper's sentence asks for building on previous work, which predicted future work is not [5]. The claim of being "designed to be extensible" is the class of claim the deployment criterion exists to weigh.

"This is Profiles advocacy wearing the costume of a comparison"

On page one, the authors' preference is disclosed, and the reader should discount for it. The criteria in Section 3 carry their provenance inline - an advisory Direction Group paper, a polled Hagenberg mandate, a standard proposed by this paper's own authors and not yet taken, and the disputants' own texts - so the reader can weigh each criterion by its source. The deployment facts in Section 5 are vendor documentation. Where the criteria admit two readings, Section 6 states both.

"P3100 builds on P2900, the most recently adopted work, which is exactly what the direction paper asks"

That reading is real and Section 6 states it, including the respect in which it is the more literal reading of P2000R5's sentence. What it cannot draw on is field experience, and the same section states why that matters: the two readings of one sentence point at different owners, which is the reason the choice belongs on an explicit ballot rather than being resolved as a byproduct of one proposal's wording review.

---

11. Conclusion

The comparison above rests on one premise from the proposal's own text: kept together, one of these two features is specified in the other's terms. The body measured the two candidate owners criterion by criterion, and the findings are at different resolutions, which the reader deserves stated plainly rather than averaged.

On deployment and field experience, the record separates the two. Both forms have deployed lineage, and the difference is in kind: the named-guarantee form runs in production across three vendors' libraries with measured cost - on by default in GCC 15 unoptimized builds and Xcode 16, opt-in in others - while the inserted-check form is opt-in test-time tooling outside its enumerated production instances. Both proposals' specifications are unshipped, and applied evenly the deployment record settles the base role for neither: each pairs a deployed lineage with an unshipped specification, and both can express the terminating named-guarantee set that ships. What it establishes is that the shipping practice terminates and that P3100's added machinery - implicit assertions, Labels, the replaceable handler - has no implementation. On systematic coverage, the P3100 model leads, though the lead does not resolve ownership: its enumeration of the undefined-behavior cases has no equivalent on the Profiles side and serves every safety effort, yet both candidate owners consume that enumeration identically. On existing practice, P2000R5's sentence supports both proposals on different clauses, and the two readings name different owners. On the guarantee and dialects, the models assign the guarantee to different authors, and the per-configuration-variation question spans the expression's runtime meaning and the compile-time properties the configuration selects. Under P3100's menu it attaches to every checkable operation, where the proposal's own resolution leaves the `noexcept` operator answering true for expressions that can throw. A Profile raises it only where its author defines an undeployed non-terminating meaning. On the single-architecture premise, deployed failure responses are diverse for stated engineering reasons. The standardized global handlers charged with adjudicating a violation class program-wide were removed with their feature in C++ and condemned by field-experience review in C. The first integration into the contract-violation machinery departed from the unified semantics, and the deployment claimed for the model conforms by containing none of its machinery. P3984R0's undeployed semantic definitions are held to the same standard. On coordination, ownership left unsettled has already cost one companion paper its wording mechanism through independent

revision.

The choice of owner is a decision for EWG, taken explicitly, with this record in front of it - the decision P4297 proposes to schedule by name. Until it is taken, the relationship is an open design question: approvals of individual undefined-behavior wording cases neither settle it nor foreclose it, and the two proposals continue to assert incompatible arrangements of the same pieces in parallel.

Per the discount rule of Section 5, the findings are falsifiable, and the falsifiers are deployment events, not shipping events. The deployment finding moves the day a Labels implementation reaches production deployment, a replaceable cross-facility violation handler ships in a production configuration, or the observe semantic of the unified model's compiler-inserted implicit assertions runs in production at scale (distinct from a library-assertion family such as BDE's `bsls_review`, whose observe-style continuation is treated in Section 8 and is design lineage rather than deployment of the machinery under comparison). The single-architecture finding of Section 8 moves the day a checking facility outside the contract family ships specified in the unified machinery's terms without departing from it. And the finding cuts the other way on the same rule: a vendor withdrawing hardening, or field failures of the named-guarantee form, would weaken the record this paper assembles. Any of these events belongs in a revision of this comparison, whichever side it favors.

Whoever writes the dedicated ballot proposal builds on the comparison assembled here. If the committee adopts the evidence standard P4297's Poll 3 proposes, this paper is the record that standard asks for. If it prefers another standard, the ledger in Section 5 is organized to be re-weighed under it.

---

12. Disclosure

The authors provide information and serve 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 [105], and the GCC implementation is in development. Ville Voutilainen is a longtime WG21 participant and a co-author of P3608R0, which Sections 3 and 6 cite, and of P3878R0, which Section 8 cites.

This paper is a comparison, not a request. It places the public record for two competing proposals in front of EWG (the Evolution Working Group) and measures each against criteria already in the committee's record, with the provenance of each criterion stated, and it rests on published papers and primary vendor documentation. It is a companion to P4297, and that pairing carries a disclosure of its own: read as one act, the pair does ask for something - an explicit decision - and the reader should see that construction plainly rather than discover it.

The authors favor the Profiles direction, and the reader should weigh the paper accordingly. In Section 5, the deployment and implementation facts stand on vendor documentation and the public committee record independent of that preference, save the authors' own Clang implementation, which is disclosed as such. The paper works only from the public record, and committee-internal materials may contain answers the record does not. It uses machine-assisted drafting.

This paper asks for nothing.

References

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

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

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

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

[5] P2000R5 - "Direction for ISO C++" (J. Garland, P. McKenney, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong, 2026).

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

[7] P3874R1 - "Should C++ be a memory-safe language?" (Jon Bauman, Timur Doumler, Nevin Liber, Ryan McDougall, Pablo Halpern, Jeff Garland, Jonathan Müller, 2026).

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

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

[10] P3543R0 - "Response to Core Safety Profiles (P3081)" (Mungo Gill, Corentin Jabot, John Lakos, Joshua Berne, Timur Doumler, 2024).

[11] clang-tidy checks, LLVM 3.8 - C++ Core Guidelines checks (LLVM Project, 2016).

[12] Use the C++ Core Guidelines checkers (Microsoft Learn, retrieved 2026).

[13] libc++ Hardening Modes (LLVM Project, retrieved 2026); assertion-semantics timeline in the Libc++ 21 Release Notes (2025).

[14] C++ Language Support (Apple Developer, retrieved 2026).

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

[16] libstdc++ macros documentation and API evolution (GCC, retrieved 2026).

[17] microsoft/STL VS 2022 Changelog and STL Hardening (Microsoft STL team, retrieved 2026).

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

[19] GCC 16 Release Series changes and the GCC 16.1 C++ Dialect Options manual (GCC, 2026); the experimental C++26 label at C++ Standards Support in GCC.

[20] C++ Support in Clang (LLVM Project, retrieved 2026).

[21] Gabriel Dos Reis, isocpp-sg15, 2025-10-14 - message in the thread "P3835 - Different contract checking for different libraries" (public SG15 archive); quoted in Sections 5 and 6.

[22] Gabriel Dos Reis, isocpp-sg15, 2025-10-14 - same thread (public SG15 archive).

[23] Timur Doumler, isocpp-sg15, 2025-10-21 - same thread (public SG15 archive).

[24] Joshua Berne, isocpp-sg15, 2025-10-17 - same thread (public SG15 archive).

[25] P3656R1 - "Initial draft proposal for core language UB white paper: Process and major work items" (Herb Sutter, Gašper Ažman, 2025); reproduces the Hagenberg poll and tally.

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

[27] Ralf Brown's Interrupt List, INT 24h - the MS-DOS critical error handler: entry conditions, action codes, and handler replacement (retrieved 2026).

[28] KB Q67586 - "Action Taken on Abort, Retry, Ignore, Fail" (Microsoft Knowledge Base, archived; retrieved 2026).

[29] assert macro documentation (Microsoft Learn, retrieved 2026).

[30] google/sanitizers issue #1039 - "How to use simultaneously Asan and Msan?" (Google sanitizers issue tracker, retrieved 2026).

[31] Chandler Carruth, llvm-dev, February 2014 - on segregating sanitizer runtime libraries (LLVM mailing list archive).

[32] Richard Smith, cfe-commits, August 2015 - review of an added UBSan failure-response option (LLVM mailing list archive).

[33] llvm-project pull request #93333 - libc++ assertion-handler customization discussion (LLVM Project, retrieved 2026).

[34] Practical Security in Production - libc++ hardening deployment report (Louis Dionne, Konstantin Varlamov, and coauthors, ACM Queue; retrieved 2026).

[35] __fastfail (Microsoft Learn, retrieved 2026).

[36] GSL 3.0.0 release (Microsoft C++ Team Blog, retrieved 2026).

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

[38] llvm-project pull request #138251 - replacing report_fatal_error with reportFatalInternalError and reportFatalUsageError (LLVM Project, 2025).

[39] Chromium checks documentation and Mojo documentation (Chromium Project, retrieved 2026).

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

[41] Rust RFC 1513 - panic runtime strategies; Rust RFC 2070 - the replaceable panic handler for freestanding targets; and rust-lang/rust issue #52652 - abort at foreign-function boundaries (Rust Project, retrieved 2026).

[42] Software FMEA under ISO 26262 - monitor-independence requirements (Orbital Jump, retrieved 2026). ISO 26262 itself is not publicly retrievable; the claim rests on this public summary.

[43] Sutter's Mill #22: "A Pragmatic Look at Exception Specifications" (Herb Sutter, 2002).

[44] P0003R5 - "Removing Deprecated Exception Specifications from C++17" (Alisdair Meredith, 2016); the revision adopted at Issaquah (November 2016).

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

[46] P3912R0 - "Design considerations for always-enforced contract assertions" (Timur Doumler, Joshua Berne, Gašper Ažman, Oliver Rosten, Lisa Lippincott, Peter Bindels, 2025).

[47] P3878R0 - "C++26 Contracts are not a good fit for standard library hardening" (Ville Voutilainen, Jonathan Wakely, John Spicer, Stephan T. Lavavej, 2025).

[48] P3471R4 - "Standard Library Hardening" (Konstantin Varlamov, Louis Dionne, 2025).

[49] P3846R0 - "C++26 Contract Assertions, Reasserted" (Timur Doumler, Joshua Berne, 2025).

[50] UndefinedBehaviorSanitizer and Security enhancements - UBSan integer-overflow checking hardening the Android media framework since Android 7.0, bounds checking in the Bluetooth stack and eleven named codecs since Android 10, abort on failure; diagnostics mode documented as negating the mitigation (Android Open Source Project, retrieved 2026).

[51] What Every C Programmer Should Know About Undefined Behavior #1/3 (Chris Lattner, LLVM Project Blog, 2011) - the optimization license of undefined signed overflow and null dereference, with worked examples.

[52] P3541R1 - "Violation handlers vs noexcept" (Andrzej Krzemieński, 2025) - the paper P3100R8 cites for the noexcept interaction; catalogues the stack-unwinding and noexcept-operator questions for unannounced throws.

[53] Control Flow Integrity - CFI enforced in official Chrome builds, SIGILL on violation, diagnostics "not for production use" (Chromium project documentation, retrieved 2026).

[54] CONFIG_UBSAN_TRAP introduction - kernel trap mode trading reporting for size, "prepared to deal with kernel code being aborted" (Kees Cook, kernel-hardening list, 2020); distribution-kernel reporting mode per x86/kconfig/64 defconfig patch - "UBSAN is actively enabled in all 3 major Linux distros I checked" (Ingo Molnar, linux-kernel list, 2025); the WARN-versus-defend statement in security things in Linux v5.7 (Kees Cook, 2020) and the production recommendation in kconfig-hardened-check issue #53 (Kees Cook, 2021).

[55] GCC contracts: per-condition exception scaffolding and terminate machinery under -fno-exceptions (GCC implementation commits, Nina Ranns and collaborators, 2024-2026).

[56] bitcoin/bitcoin pull request #27334 - "util: implement noexcept move assignment & move ctor for prevector" (Martin Leitner-Ankerl, 2023) - the noexcept additions that let vector reallocation move instead of copy.

[57] BDE 3.15.0 release notes - "Assertion Handlers that Return are Now Disallowed" (Bloomberg BDE team, 2018).

[58] P2811R4 - "Contract-Violation Handlers" (Joshua Berne, 2023) - the Bloomberg deployment retrospective behind the C++26 handler design.

[59] Boost.Assert documentation and deployed handler definitions: QuantLib, OSRM, userver, RethinkDB; throwing-handler testing guidance in libassert issue #114 (retrieved 2026).

[60] RFC: Hardening in libc++ - Chromium's override-point objection and the maintainers' response (Hans Wennborg, Konstantin Varlamov, LLVM Discourse, 2023).

[61] N2271 - "EASTL - Electronic Arts Standard Template Library" (Paul Pedriana, 2007) - the game industry's account of replacing the standard assert.

[62] Unwind considered harmful? (Niko Matsakis, 2024) - production Rust panic-strategy practice.

[63] Known pitfalls in C++26 Contracts - production migration of ScyllaDB assertions to contracts, always-enforced semantics pinned per assertion (Ran Regev, using std::cpp 2025, Madrid, March 2025; video published 2025-06-16; auto-transcribed).

[64] Rationale for Ada 2005, Section 5.4: The Ravenscar profile (John Barnes, 2005-2006) - the language designers' own statement of what a standardized profile is; Ravenscar has been part of the Ada standard since Ada 2005.

[65] Annocheck: Examining the contents of binary files (Nick Clifton, Red Hat Developer, 2019) - the Fedora toolchain's audit of build configuration; the audited hardening flags are injected by the distribution's default build configuration (redhat-rpm-config).

[66] blhc - build log hardening check (Simon Ruderich, v0.14, 2024; retrieved 2026) - Debian's audit that hardening flags set by `dpkg-buildflags` survived the build system.

[67] llvm-project pull request #205739 - first upstreaming step of a P2900 Contracts implementation in Clang (Chuanqi Xu, opened 2026-06-25); the author's plan and the quoted rationale are in the PR discussion.

[68] microsoft/STL pull request #5274 - STL Hardening implementation: the fastfail rationale and `_SECURE_SCL` history (Stephan T. Lavavej, 2025); the contracts-based reporting plan is tracked in microsoft/STL issue #5300, which records the feature as blocked on the unimplemented Contracts machinery.

[69] TheHPXProject/hpx pull request #7129 - contract-assertion infrastructure with per-build ENFORCE/OBSERVE/IGNORE modes mapped to P2900 semantics (opened 2026-03-30, merged 2026-07-05); the "pre! is a nice fit with where we landed" comment is in the review thread (2026-04).

[70] GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production (Serebryany, Kennelly, Phillips, Denton, Elver, Potapenko, Morehouse, Tsyrklevich, Holler, Lettner, Kilzer, Brandt, 2023) - the sampling design, the Android 14 recoverable mode, and the authors' own scope statement.

[71] LLVM Differential D36810 - "Minimal runtime for UBSan" (Evgenii Stepanov, 2017) - the design statement and review discussion for the production UBSan runtime.

[72] Memory Tagging and how it improves C/C++ memory safety (Kostya Serebryany, CppCon 2018; auto-transcribed) - ASan's production and mitigation limits stated by its co-creator; the same position in the 2018 LLVM dev meeting slides.

[73] Linux kernel commit cf68fffb66d6 - "add support for Clang CFI" (Sami Tolvanen, merged v5.13) - the commit message's statement on permissive mode; the same prohibition in AOSP kernel CFI documentation: "Permissive mode must not be used

in production."

[74] RFC: Enforcing bounds safety in C (-fbounds-safety) (Yeoul Na, Apple, 2023) - "turning OOB accesses into deterministic traps", "adopted on millions of lines of production C code"; deployment scope in the EuroLLVM 2023 slides.

[75] C++ Memory Safety in WebKit (Geoffrey Garen, C++Now 2025; auto-transcribed) - hardening at the extensive level in release builds, "The end to end performance cost in WebKit has been zero"; the build setting on the slide deck.

[76] Mozilla Bugzilla 1989114 - "Support enabling STL hardening in release builds" (Nika Layzell, fixed in Firefox 145, 2025) - the cross-vendor build flag; the negligible-cost motivation in meta bug 1986488 (Andrew McCreight).

[77] dnf5 issue #2553 - libstdc++ assertion abort in Fedora Rawhide under GCC 15's default-on checks, with the failed `string_view` precondition named in the diagnostic (December 2025).

[78] P1407R1 - "Tell Programmers About Signed Integer Overflow Behavior" (Scott Schurr, 2019); the dialects-already-exist response quoted in Section 7.

[79] Linus Torvalds on flag-pinned semantics: -fno-strict-aliasing, LKML 2003; "This is why we use -fwrapv...", LKML 2018; the tree-wide swap in kernel commit a137802ee839 - "Don't use '-fwrapv' compiler option: it's buggy in gcc-4.1.x" (2009).

[80] pgsql-hackers, "Need -fwrapv or -fno-strict-overflow for gcc-4.3" (Tom Lane, Kris Jurka, March 2008) - the "diking out... security-critical overflow checks" finding; the permanent flag in PostgreSQL's configure.ac.

[81] Abseil FAQ (Abseil team, retrieved 2026) - ABI-affecting compile options "need to be applied to the entire build on a global basis", with `-fexceptions` and `-DNDEBUG` among the named examples.

[82] libstdc++ manual, "Exceptions: Doing without" (GCC, retrieved 2026) - `-fno-exceptions` produces "a dialect without exception handling", in the vendor's own words.

[83] P0709R3 - "Zero-overhead deterministic exceptions" (Herb Sutter, 2019); the survey figures on exception bans and the "divergent language dialect" characterization.

[84] ODR, libc++ hardening, Profiles and Contracts (Louis Dionne, 2025) - mixed-configuration ODR mechanics, the field-breakage report for `-fno-exceptions` mixing, the ABI-tag mitigation and its stated limits; the tag machinery introduced in llvm-project pull request #69669 (2023).

[85] cplusplus/papers issue #2178 - SG21 poll record for P3541R1, posted to the tracker 2025-01-10: the operator-returns-true option polled to consensus (SF 5, F 10, N 1, A 2, SA 1); the truth-reporting option polled to consensus against.

[86] P2811R6 - "Contract-Violation Handlers" (Joshua Berne, 2023), Section 6: an incorrect or build-mode-varying `noexcept` answer as "a fundamental flaw in a Contracts facility for C++".

[87] P2969R0 - "Contract annotations are potentially-throwing" (Timur Doumler, Ville Voutilainen, Tom Honermann, 2023) - the redefined operator question stated verbatim, and the zero-overhead rationale; co-authored by an author of this paper, as Section 12 discloses for P3608R0 and P3878R0.

[88] P3591R0 - "Contextualizing Contracts Concerns" (Joshua Berne, Timur Doumler, 2025) - the scaffolding-cost rebuttal quoted in Section 7.

[89] llvm-project issue #53062 - `noexcept` boundary destroying the crash backtrace ("partially unwound", 2022, open); the maintainer statement of both halves in Exception unwinding through a noexcept function (Louis Dionne, LLVM Discourse, 2022).

[90] ITK Discourse: "Regarding C++11 noexcept" (Niels Dekker, 2019) - the measured four-times speedup of `std::vector::reserve` from a `noexcept` move constructor, merged as ITK pull request #382.

[91] noexcept affects libstdc++'s unordered_set (Arthur O'Dwyer, 2024, with the 2024-08-27 update crediting Martin Leitner-Ankerl) - the libstdc++ per-node caching mechanism and the reported Bitcoin Core 9% memory reduction from a `noexcept` hasher (2019, commit 67d9990).

[92] Contract use: Past, Present, and Future (Joshua Berne, CppCon 2019; auto-transcribed) - the production failure of blanket continuation, the abandoned configurable "smart" handler, and the string-destructor check withdrawal, in the architect's own account.

[93] P3336R0 - "Usage Experience for Contracts with BDE" (Joshua Berne, 2024) - rebasing `bsls_assert` on `contract_assert` under an experimental GCC implementation.

[94] WG14 N1967 - "Field Experience With Annex K - Bounds Checking Interfaces" (Carlos O'Donell, Martin Sebor, 2015) - the process-global runtime-constraint handler's field record and the removal proposal.

[95] glibc malloc-hook removal: Securing malloc in glibc: Why malloc hooks had to go (Siddhesh Poyarekar, 2021) and the glibc 2.34 release announcement; the CRT's thread-local variant in _set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler (Microsoft Learn, retrieved 2026).

[96] Standardized E-GAS Monitoring Concept for Gasoline and Diesel Engine Control Units, Version 6.0 (EGAS Workgroup, 2015; mirrored copy) - "System reactions are triggered independently of the function controller in case of fault"; ZVEI Best Practice Guideline: Software for Safety-Related Automotive Systems (ZVEI, January 2025) - interference detection and mitigation developed at the unit's maximum ASIL.

[97] Crashpad overview design and Breakpad client design (Google, retrieved 2026) - out-of-process crash handling and the in-process constraints that force it.

[98] Go: Ian Lance Taylor, golang-nuts, 2025-12-08 - the net/http per-request recovery "a historical mistake"; .NET: HandleProcessCorruptedStateExceptionsAttribute (Microsoft, retrieved 2026) - "Recovery from corrupted process state exceptions is not supported."

[99] Tencent/rapidjson issue #2161 - "RapidJSON is great, but the assertions make it too risky for production use" (2023-2025); the customization-point remedy is in the thread and the library's configuration documentation.

[100] scylladb commit aa1270a00c4 - "treewide: change assert() to SCYLLA_ASSERT()" (Avi Kivity, August 2024); the Redpanda account in scylladb/seastar discussion #2539 (Noah Watkins, November 2024).

[101] Ran Regev, isocpp-sg15, 2025-10-14 - the P3835 thread (public SG15 archive); quoted in Section 8.

[102] John Spicer, quoted in Andrzej Krzemieński's SG15 message, 2025-10-01 (public SG15 archive) - checking modes must attach to components for contracts to be viable.

[103] libc++ assertion-semantics record: llvm-project pull request #149459 - semantics "closely mimic C++26 Contracts evaluation semantics" (Konstantin Varlamov, LLVM 21, July 2025); pull request #167636 - the `LIBCXX_ASSERTION_SEMANTIC` configure-time knob and the production-modes-to-quick-enforce default mapping (November 2025); pull request #77883 - withdrawal of the compile-time and link-time handler overrides, "we feel that this should primarily be controlled by vendors, not users" (January 2024).

[104] glog failure-function documentation (Google logging library, retrieved 2026) and Qt logging documentation, qInstallMessageHandler (Qt 6, retrieved 2026) - the reporting-delegable, policy-fixed shape of the deployed cross-library funnels.

[105] C++ Alliance Clang Profiles implementation: framework and `std::init` profile documentation - the P3589R2 framework attributes and the compile-time `std::init` profile, marked experimental; and public build releases - regularly released dated builds, latest 2026-07-10 (retrieved 2026-07-12).

[106] Bloomberg BDE assertion components - `bsls_review.h`, `bsls_review.cpp`, and `bsls_assert.h` (Bloomberg Finance L.P., Apache-2.0; commit `d47fd39`, `main` HEAD on 2026-07-13). The two-tier design in the components' own documentation: `bsls_review` is "designed to allow apparently working production software, which nonetheless may harbor contract violations, to increase the number of precondition checks used to catch such bugs without negatively impacting the existing behavior of the software" (its default handler logs and returns), while `bsls_assert` holds that "tasks may not install an assertion handler that returns control to the point immediately following the detection of a failed assertion" and "enforces that policy by terminating the task" when a handler returns.

[107] P3835R0 - "Contracts make C++ less safe - full stop" (John Spicer, Ville Voutilainen, Jose Daniel Garcia Sanchez, 2025); the SG15 thread it prompted is cited at [21] and [101].

[108] P4044R0 - "Just `pre!`. Mandatory precondition for contracts" (Lucian Radu Teodorescu, 2026); discussed in the HPX review thread cited at [69].