Profile Analysis and Verification Evidence (PAVE)

Document Number:	P4137R0
Date:	2026-04-17
Intent:	Inform
Audience:	SG12, SG23, EWG

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

Table of Contents

Abstract

The paper offers a way to measure what the type-safety profile actually covers.

P3984R0[1] defines the rules. It does not measure how much of a real codebase those rules can verify. This paper proposes PAVE - Profile Analysis and Verification Evidence - a three-phase methodology that answers the question. Phase 1 classifies every function against the profile's rules using AST predicates. Phase 2 distinguishes true positives from false positives. Phase 3 infers annotations that expand the verifiable subset. The tools exist today.

Revision History

R0: April 2026 (post-Croydon mailing)

- Initial version.

---

1. Disclosure

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

The author believes profiles is a reasonable direction for C++ safety and deserves real-world use before the committee judges the design.

This paper examines the published record. That effort requires re-examining consequential papers, including papers written by people the author respects.

This paper asks for nothing.

---

2. The Evidence Gap

P3984R0[1] defines two profiles:

- **Type safety**: Every object is used exclusively according to its definition.
- **Resource safety**: No resource is leaked.

The definitions are clear. The enforcement rules are enumerable. The paper describes what must be banned (pointer arithmetic, raw pointer subscripting, C-style casts, union access), what must be checked at run time (range errors, `nullptr` dereference, with hardened libraries per P3471R4[2], "Standard library hardening"), and what must be analyzed statically (invalidation through dangling pointers). The profiles framework itself is specified in P3589R2[3], "C++ Profiles: The Framework," with development guidance in P3274R0[4], "A framework for Profiles development." The design is coherent.

What the paper does not provide is a measurement. How much of a real codebase falls in the verifiable subset? P3970R0[5], "Profiles and Safety: a call to action," urges implementers to coordinate and experiment. The paper's own design decisions answer the question partially - by exclusion.

2.1 Trusted Constructors and Destructors

P3984R0[1] Section 2.3:

> *"We can ... Trust the definition of constructors and destructors to be correct (e.g., not leak)."*

Every constructor and destructor is trusted. The profile does not verify that a constructor initializes all resources correctly or that a destructor releases them. The guarantee that "no resource is leaked" depends on constructors and destructors being correct, but the profile does not check that they are.

2.2 Deferred Concurrency

P3984R0[1] Section 4:

> *"The type-and-resource profile assumes that there are no data races. To ensure that, a separate concurrency profile is needed. Specifying that is non-trivial and beyond the scope of this paper."*

The type-safety guarantee holds only in single-threaded code or code that has been independently verified to be free of data races. In any multithreaded program, the guarantee carries an asterisk.

2.3 Conservative Invalidation Analysis

P3984R0[1] Section 2.6 proposes a rule for portability:

> *"If a condition cannot be evaluated at compile time, whatever is guarded by that condition is assumed to be executed."*

The profile treats conditional use of banned constructs as unconditional. A function that returns a local pointer on one branch and a static pointer on the other is rejected even if the local-pointer branch is unreachable. A function that calls an invalidating function inside an `if` is treated as if the call always happens. The rule is sound - it produces no false negatives. It also rejects valid code whenever control flow is non-trivial. The invalidation analysis builds on the lifetime safety work in P1179R1[6], "Lifetime safety: Preventing common dangling," and the dangling-pointer elimination described in P3346R0[7], "Profile invalidation - eliminating dangling pointers."

2.4 Abstraction Implementations Are Excluded

P3984R0[1] Section 1:

> *"Code that needs to violate that guarantee or can't be proven not to must be isolated. Examples of such necessary code include the implementation of key abstractions (e.g., vector)."*

The implementations of `vector`, `unique_ptr`, `string`, allocators, smart pointers, and every container in the standard library require pointer arithmetic, casts, or manual memory management. The profile cannot verify them. They must be marked as

trusted and excluded from the verifiable subset.

2.5 The Question

Each exclusion is individually reasonable. Taken together, they raise a question the committee should answer before standardizing the guarantee: after excluding trusted constructors, trusted destructors, trusted abstraction implementations, all multithreaded code, and all code with non-trivial control flow around pointers - what percentage of a real codebase remains in the verifiable subset?

The answer might be large. The answer might be small. The answer is unknown. PAVE provides a methodology to find it.

The measurement is relevant even for code written from scratch under the profile. New code calls into standard library implementations, abstraction layers, and third-party libraries that predate the profile. The coverage measurement on the library layer tells you how much of the actual call stack is verified - including the code your profile-compliant function calls into. If the verifiable subset excludes the implementations that new code depends on, the guarantee has a hole regardless of when the calling code was written.

---

3. The Profile Rules as AST Predicates

The profile's banned constructs are syntactic. Each maps to a small set of Clang AST node types. A tool that walks the AST of every function in a translation unit can classify each function against the profile's rules without implementing a full profile checker.

Profile Rule	AST Predicate
No pointer arithmetic	`BinaryOperator`/`UnaryOperator` on pointer type
No raw pointer subscript	`ArraySubscriptExpr` with raw pointer base
No C-style cast	`CStyleCastExpr`
No `reinterpret_cast`	`CXXReinterpretCastExpr`
No union access	`MemberExpr` on `isUnion()` `RecordDecl`
No uninitialized members	`CXXConstructorDecl` initializer list vs. field list
No local pointer return	`ReturnStmt` with `DeclRefExpr` to local of pointer type
No invalidating call w/alias	Non-const call while alias to argument exists (flow-sensitive)

The first seven rules are syntactic pattern matches. A clang-tidy check or libTooling pass implementing them is a few hundred lines of code against the AST. The eighth rule - invalidation with aliasing - requires flow-sensitive analysis and is harder to

implement, but the conservative approximation ("assume non-const invalidates") is itself a syntactic check on the callee's signature.

The output of the tool is a per-function classification:

- **Clean**: no banned constructs detected.
- **Rejected**: one or more banned constructs detected, with the specific rule(s) violated.
- **Trusted**: the function is annotated as unverified (e.g., `[[suppress(profiles)]]` per P3589R2[3]).

---

4. Phase 1: Mechanical Classification

Phase 1 runs the AST-based tool over a target codebase and produces an aggregate coverage report.

The report answers the first-order question: what fraction of the codebase does the profile claim to verify? An illustrative output:

Category	Functions	Percentage
Clean	?	?
Rejected	?	?
Trusted	?	?

The numbers are unknown. That is the point. Until someone fills in the table for a real codebase, the committee is standardizing a guarantee without knowing its scope.

Phase 1 is reproducible. The same tool run on the same codebase produces the same classification. Different codebases will produce different numbers - a leaf application using standard containers will score differently from a systems library implementing those containers. Both measurements are informative. The profile is designed for the former. The measurement confirms whether the design matches the reality.

The rejected functions carry metadata: which rule each function violates. The distribution across rules is as informative as the aggregate. If 80% of rejections are pointer arithmetic and 2% are union access, the committee knows where the profile's coverage boundary lies and where hardened library alternatives would have the greatest impact.

---

5. Phase 2: Semantic Analysis

Phase 1 identifies what the profile rejects. Phase 2 asks whether the rejections are correct.

For every function classified "rejected," a deeper analysis determines whether the banned construct is actually unsafe. Three sub-classifications:

- **True positive.** The construct is genuinely unsafe. The profile correctly rejects the function. Example: unbounded pointer arithmetic indexing past an allocation.
- **False positive, annotatable.** The construct is safe, and a standard annotation would let the profile accept the function. Example: a non-const member function that does not invalidate iterators - adding `[[not_invalidating]]` resolves the rejection. P3984R0[1] introduces this attribute and notes that the compiler can verify it.
- **False positive, structural.** The construct is safe, but no annotation scheme would help. The profile's rules are fundamentally too conservative for the pattern. Example: pointer arithmetic that is bounded by a `static_assert` on the allocation size, or a `reinterpret_cast` for serialization where layout is guaranteed by the type definition.

The structural false positive category is the most important finding. It measures the irreducible gap between what the profile promises and what it can deliver. True positives validate the profile. Annotatable false positives measure the value of richer annotation vocabularies. Structural false positives measure the limits of the approach.

5.1 LLM-Assisted Triage

An LLM reading the function body can perform the initial classification. The LLM has semantic understanding that the AST walk lacks - it can reason about whether pointer arithmetic is bounded, whether a cast preserves type safety, whether aliasing is benign. The LLM classification is not a proof. It is a triage step that generates hypotheses for human review.

A practical workflow:

1. Feed each rejected function to an LLM with a rubric defining the three sub-classifications.
2. The LLM emits a classification and a one-sentence justification.
3. A human reviewer validates a stratified sample - enough to estimate the error rate of the LLM triage.

Example. Two functions rejected by Phase 1 for raw pointer operations:

```
void zero_fill(int* p, size_t n) {
    for (size_t i = 0; i < n; ++i) p[i] = 0;
}
```

LLM classification: **True positive.** The pointer arithmetic is unbounded - the caller controls `n` and nothing constrains it relative to the allocation size.

```
void copy_header(const char* buf, header& h) {
    static_assert(sizeof(header) == 16);
    std::memcpy(&h, buf, sizeof(header));
}
```

LLM classification: **Structural false positive.** The `memcpy` is bounded by `sizeof(header)`, a compile-time constant. No annotation resolves this - the profile lacks a mechanism to express "the caller guarantees at least N bytes."

The distinction matters. An AST walk rejects both functions identically. The LLM distinguishes them because it can read the `static_assert` and reason about the bound.

The LLM does not replace the formal guarantee. It expands the code surface that the formal guarantee can reach by identifying where annotations are missing and where the rules are too conservative.

---

6. Phase 3: Annotation Inference

For every function classified as "false positive, annotatable" in Phase 2, the annotation that would resolve the rejection can be generated.

`[[not_invalidating]]` is currently the only compiler-verifiable annotation that P3984R0[1] defines. It applies to one rejection category: invalidation with aliasing. Functions rejected for pointer arithmetic, casts, or union access resolve as either true positives or structural false positives - no annotation rescues them. The annotation dividend therefore measures a narrow but real lever. P3984R0[1] Section 2.6 introduces it as an optimization for the invalidation analysis:

> *"Add a [[not_invalidating]] attribute to be used to speed up analysis by marking non-const functions and functions that take non-const pointer arguments that don't invalidate."*

The attribute is compiler-verifiable. When the function definition is compiled, the compiler can confirm that the function does not invalidate. The annotation is not a trust-me marker - it is a checkable claim.

An LLM scanning a codebase can generate `[[not_invalidating]]` annotations for every non-const function that does not invalidate. The compiler verifies each one. The result is a measurable expansion of the verifiable subset.

The gap between Phase 1 coverage (before annotations) and Phase 3 coverage (after annotations) is the **annotation dividend** - the practical value of richer annotation vocabularies in the profiles framework. If the dividend is large, the framework should prioritize annotation design. If the dividend is small, annotations are not the bottleneck and the profile's rules themselves need revision.

---

7. Implications

Four outcomes are possible. Each is useful.

1. **The verifiable subset is large.** The profile delivers genuine value. The committee should advance it with confidence. PAVE provides the evidence to support that decision.
2. **The verifiable subset is small.** The profile is a compliance label - useful for governance and regulatory purposes, but not a practical tool for preventing bugs in the code where bugs are most likely to occur (abstraction implementations, systems code, concurrent code). The committee should set expectations accordingly.

3. **The annotation dividend is large.** The verifiable subset is small without annotations but substantially larger with them. The framework should invest in annotation design - more attributes, richer vocabularies, compiler-verified claims about function behavior. The annotations are the leverage point.
4. **The structural false positive rate is high.** The profile's rules reject large amounts of safe code that no annotation can rescue. The rules themselves need revision. The conservative "assume all branches execute" heuristic or the "non-const means invalidating" default may need to be relaxed, at the cost of more complex analysis.

All four outcomes inform the committee's decision. None requires prejudging the result. The methodology is neutral - it measures what is there.

The methodology is not limited to the type-safety profile. Any profile whose rules can be expressed as AST predicates - the initialization profile (P3402R3[8], "A Safety Profile Verifying Initialization"), the casting profile, the ranges profile - can be measured the same way. PAVE is a general framework for empirical evaluation of profiles.

---

8. The Tools Exist

The methodology described in this paper requires no novel tooling.

Phase 1 uses Clang's AST, which is mature, well-documented, and available on every major platform. clang-tidy already implements several C++ Core Guidelines[9] checks[10] that overlap with profile rules:

clang-tidy Check	Profile Rule Overlap
`cppcoreguidelines-pro-bounds-pointer-arithmetic`	No pointer arithmetic
`cppcoreguidelines-pro-bounds-array-to-pointer-decay`	No raw pointer subscript
`cppcoreguidelines-pro-type-reinterpret-cast`	No `reinterpret_cast`
`cppcoreguidelines-pro-type-cstyle-cast`	No C-style cast
`cppcoreguidelines-pro-type-union-access`	No union access

These checks do not produce the aggregate classification that PAVE requires, but they demonstrate that the AST predicates are well-understood and already implemented. A custom libTooling pass that combines them into a single classification tool is straightforward engineering.

Phase 2 requires an LLM with a rubric. The rubric is the three-way classification defined in Section 4. No specialized model is needed - any LLM capable of reading C++ function bodies and reasoning about pointer safety can perform the triage. Human review validates the output.

Phase 3 is a direct extension of Phase 2. The LLM generates candidate annotations. When a profile-aware compiler is available, it verifies them mechanically. Today the measurement - counting annotatable functions and computing the annotation dividend - can be performed by the LLM triage step without compiler cooperation.

Phases 1 and 2 require no compiler modification and no new language features. They can be executed today, on any codebase that compiles with Clang. Phase 3's verification step benefits from compiler support for `[[not_invalidating]]`, which no shipping compiler yet provides; the measurement itself does not depend on it.

---

9. Availability

If empirical coverage data is desired before advancing profiles to the IS, the methodology described in this paper provides a reproducible way to generate it.

A minimal execution:

- Run PAVE Phase 1 on at least one substantial, publicly available C++ codebase. Report the clean/rejected/trusted distribution.
- Run PAVE Phase 2 on a stratified sample of the rejected functions. Report the true positive / annotatable false positive / structural false positive distribution.
- Run PAVE Phase 3 on the annotatable false positives. Report the annotation dividend.

The results will either confirm that the type-safety profile covers a meaningful fraction of real code or reveal that the coverage is narrower than expected. Either outcome informs the committee's decision.

Implementers, profile proponents, and independent researchers are welcome to execute the methodology and publish results.

Acknowledgments

Thanks to Bjarne Stroustrup for P3984R0 and the profiles vision. Thanks to Gabriel Dos Reis for P3589R2 and the framework specification. Thanks to Herb Sutter for the lifetime analysis that informed the invalidation rules.

References

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

[2] P3471R4 - "Standard library hardening" (Konstantin Varlamov, Louis Dionne, 2025).

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

[4] P3274R0 - "A framework for Profiles development" (Bjarne Stroustrup, 2024).

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

[6] P1179R1 - "Lifetime safety: Preventing common dangling" (Herb Sutter, 2019).

[7] P3346R0 - "Profile invalidation - eliminating dangling pointers" (Bjarne Stroustrup, 2024).

[8] P3402R3 - "A Safety Profile Verifying Initialization" (Marc Laverdiere, Cody Lapkowski, Christof Gros, 2025).

[9] C++ Core Guidelines

[10] clang-tidy C++ Core Guidelines checks