

Function Summaries for Scaling Code Analysis of C++

Gabriel Dos Reis
Susquehanna International Group

1 ABSTRACT

Profiles [P3589R3] are to be enforced by local static analysis [P3274R0]: one translation unit at a time, and within it one function at a time. At a call to a function defined elsewhere, such an analysis has only the callee's declaration. The declaration says little about what the callee writes, and nothing about what it retains, releases or invalidates. The analysis gives the same verdict for every definition that matches the declaration, so it must reject some correct programs or accept some incorrect ones. A function summary resolves this. It states, once per function, what a call requires and what it may do. It is verified against the function's definition where the definition is compiled, and used at every call.

This paper proposes function summaries as the means by which code analysis of C++, and with it the enforcement of profiles, scales. The proposal rests on the theory of modular static analysis, and on evidence from widespread implementations and practices:

- industrial analyzers, which reason with summaries, inferred (Infer, on code bases of millions of lines) or written (SAL, over 400,000 annotations);
- the C++ standard, which already specifies its library with summaries written in English;
- contemporary compilers and analyzers, which state parts of summaries in separate files, such as Clang's API notes, or in annotations that they check, such as Clang's lifetime and thread-safety attributes (§8).

To test the proposal, we wrote summaries for the whole standard library (5,709 function entries), and restated as summaries every function model written by hand in GCC's static analyzer. Writing the library summaries also found problems in the text of the working draft (§5).

Summaries fit the Profile Framework as specified, with header files as well as with modules. This paper proposes no change to P3589R3, nor to any specific profile paper; it is an informational paper.

2 THE PROBLEM AT A CALL

P3589R3 enforces a profile on a translation unit: an implementation "can present a profile plugin with the complete abstract semantic graph of a translation unit to an external tool or plugin for profile enforcement" (§1.1.3). P3274R0 requires the analysis to be local: "Static guarantees can, and should, be achieved through local static analysis. For logical reasons and compile-time performance, global static analysis (whole-program analysis) doesn't scale: Most C++ users do not have access to every line of code potentially executed, such as libraries, operating systems, and device drivers." P3081R1 §1.2 requires each rule to be "deterministically decidable at compile time" and "sufficiently efficient to implement in-the-box in the C++ implementation".

A translation unit calls functions that it does not define:

```
void normalize(std::vector<double>& v);    // defined in another translation unit

double scaled_front(std::vector<double>& v)
{
    double& x = v.front();
    normalize(v);
    return x;
}
```

If `normalize` scales the elements in place, `scaled_front` is correct. If `normalize` appends an element, the append can reallocate, and `x` then dangles. Both definitions match the declaration. The analysis of `scaled_front` sees the same text in both cases, so it gives the same verdict in both. If it accepts, it accepts an error; if it rejects, it rejects a correct program. The same holds for every rule whose verdict depends on what a callee does: whether an object passed by pointer is initialized after the call, how much of a buffer the call fills, whether it keeps a pointer, whether it releases a resource.

At such a call, an analysis can do one of three things:

1. Assume the worst: the call may write, invalidate, retain or release anything reachable from its arguments, from globals and from pointers that escaped earlier. Every fact about those objects is lost at the call. Correct programs are rejected, and users respond with suppressions.
2. Assume the best: the call does nothing that matters to the rule. Incorrect programs are accepted.
3. Use a statement of what the callee does, made at its interface, the same for every implementation, and checked against the callee's definition where the definition is compiled.

Analyzing the callee's body instead is whole-program analysis, which P3274R0 excludes; for calls into the operating system or into libraries distributed in compiled form, there is no body to analyze. A function summary is the statement of the third choice.

P3446R0 meets the problem in an example where a factory's result depends on only one of its two arguments: "without looking into `create_widget()`, we can't know that, so without non-local static analysis we must disallow that example. We have five choices for this potentially large class of programs:

- Rely on non-local analysis
- Disallow such examples as too complex for offering the invalidation guarantee
- Allow the potential problem to slip through
- Invent some annotation to help the programmer
- Suppress invalidation for such code".

It continues: "None are particularly attractive. I see an annotation as the least bad". The first of the five is whole-program analysis, and the second is the first choice above. The third and the fifth are the second choice, the fifth with a programmer's claim attached. The fourth is the third choice: an annotation that states what a function does is one claim of a summary.

2.1 WHAT SUMMARIES ADD TO THE PROFILE PROPOSALS

The profile proposals already decide what their rules do at calls, and summaries a mechanism to support and implement those decisions. P3081R1 §4.9 (Option 2) tracks the active member of each union. A member access is checked by "(Uindex == i || Uindex == unknown)". The index becomes unknown "For every other use of U that would not be valid or would change meaning if U was const", for example, in its note, "using U as an argument to a non-inline function that takes U by reference to non-const and might or might not modify it." Without facts about the callee, the rule must give up either the check or correct programs at such a call. It keeps the correct programs: every member access passes until the lifetime of a member begins again. A summary that states which member is active when the callee returns keeps both.

P3589R3 §1.1.2 illustrates local suppression with this example:

```
[[profiles::enforce(std::type)]];
extern int read(char* buf, int n);
int main()
{
    [[profiles::suppress(std::type)]] char buffer[1024];    // OK.
    int len = read(buffer, 1024);
}
```

It explains: "The declaration of the local variable buffer suppresses the std::type profile requirement that all variables must be initialized at definition point." A second form of the suppression carries the justification "buffer is filled in by read()". The suppressed rule is about declarations. A rule about reads of indeterminate values would need no suppression here, but only if the checker knows what read writes. §2 gives a summary of read that states this fact for every call, so that the checker can verify what the justification says.

3 WHAT A FUNCTION SUMMARY IS

A function summary describes a *function's externally observable behavior in terms of the program state at entry and at exit*. The proposed model is based on evidence from widespread implementations and practices: each of its parts is stated today by the library specification (§4), by ACSL (below), or by the analyzers and annotations of §3. It has these parts.

- **Requirements:** conditions the caller must establish. A violation is the caller's error. Each requirement says what a violation leads to: undefined behavior, erroneous behavior, or a check that stops the call before its other effects (hardened preconditions).
- **Outcomes:** alternative behaviors, chosen by the callee or by the environment. Each outcome has:
 - a **guard** on the entry state;
 - an **exit**: return, throw of a type T, or termination;
 - **postconditions** relating the entry and exit states (the result, and values at entry written e@entry);
 - a **footprint**: the locations the outcome may write. All other locations are unchanged (the frame rule).
 - **resource effects**: acquisition, release or change of state in a declared family of resources (heap blocks, descriptors, locks, coroutine frames);

- **lifetime effects:** objects whose lifetimes begin or end;
- **retained pointers:** pointers the callee keeps after it returns, and until when; and results that remain valid only until a stated event;
- **invalidation:** the pointers, references and iterators that become invalid;
- **invocations:** the callbacks the callee calls, when (before returning, later, or concurrently) and how often.
- **Template requirements:** for templates, the requirements on the template arguments, including the semantic requirements of concepts.

Two rules make summaries usable for guarantees. First, an absent claim means unknown*, never none*. A summary that says nothing about what a function writes allows it to write anything reachable. Second, the outcomes are complete only when the summary claims they are.

C has a specification language with this structure. ACSL, the specification language for C used by the Frama-C platform, specifies a function by named behaviors. Each behavior has an assumption, requirements, a footprint (assigns) and postconditions, and a specification can claim that its behaviors are complete or disjoint [ACSL, §2.3.3]. Its semantics includes the frame rule for each behavior: "For each i , if the function is called in a pre-state where A_i holds, then each memory location of that pre-state that does not belong to the set L_i is left unchanged in the post-state."

The proposed notation is YAML, in a restricted form in a profile. Summaries are written and reviewed by people, such as library authors and the authors of a specification, so the notation is chosen for review. Block style puts one fact on a line, so a change shows as a small line diff, and comments can cite the paragraph that a fact comes from. The profile is restricted so that a file can be read only one way:

- one document per file;
- no anchors, aliases, merge keys or tags (with them, one line could change the meaning of many entries);
- typed values only in their JSON spellings: null, true, false and decimal numbers;
- every expression a double-quoted string, which YAML does not reinterpret.

Each file is therefore a JSON value written in block style. Tools that want JSON can convert a file mechanically, losing only the comments. The validator that we used for the corpus of §5 works this way: it converts each file to JSON and runs its checks on the result.

Here is a summary of the read of §1.1, for a read documented to require $n \geq 0$, to store at most n characters and return how many it stored, or else to return -1 :

```
- id: example.read
  match: { cxx_name: "read" }
  signature: "int read(char* buf, int n)"
  expects: ["n >= 0", "$valid_write (buf, n)"]
  outcomes:
    - name: data
      ensures: ["$ret >= 0", "$ret <= n", "$initialized (buf, $ret)"]
      writes: ["$elements (buf, $std_range (0, $ret))"]
    - name: failed
      ensures: ["$ret == -1"]
      writes: []
  complete: true
```

With this summary, after testing $\text{len} \geq 0$, a checker accepts reads of `buffer[i]` for $0 \leq i < \text{len}$. It rejects a read of `buffer[ $\text{len}$ ]`, which is indeterminate if $\text{len} < 1024$ and out of bounds if $\text{len} == 1024$. It also rejects any read of `buffer` on the path where $\text{len} == -1$. The justification of §1.1 becomes a checked fact, stated once for every call of `read`. The summary also makes it precise: it covers the first `len` characters, and only when `read` succeeds.

4 WHY SUMMARIES MAKE ANALYSIS SCALE

Theory. Cousot and Cousot [Cousot 2002] present "four basic methods for compositional separate modular static analysis of programs by abstract interpretation: simplification-based separate analysis; worst-case separate analysis; separate analysis with (user-provided) interfaces; symbolic relational separate analysis". Worst-case separate analysis is the first choice of §1; separate analysis with user-provided interfaces is analysis with written summaries. Distefano, Fähndrich, Logozzo and O'Hearn [Distefano 2019] define compositionality: "A program analysis is compositional if the analysis result of a composite program is defined in terms of the analysis results of its parts and a means of combining them." They state its consequences: "each procedure only needs to be visited a few times, and many of the procedures in a codebase can be analyzed independently, thus opening opportunities for parallelism. A compositional analysis can even have a runtime that is (modulo mutual recursion) a linear combination of the times to analyze the individual procedures. For this to be effective, a suitable abstract domain, for instance limiting or avoiding disjunctions, should also contain the cost of analyzing a single procedure. Finally, compositional analyses are naturally incremental—changing one procedure does not necessitate re-analyzing all other procedures."

A C++ build has the same structure. Translation units are compiled independently and in parallel. A change to a function body recompiles one unit, and a change to a header or to a module interface recompiles the units that use it. Summaries follow the same dependencies: a changed body is verified again against its unchanged summary, and a changed summary invalidates the results of its callers.

Inferred summaries. Infer, described in the same article, infers a summary of each procedure and uses it at calls. Of a null dereference in OpenSSL whose source was five procedures deep: "Infer finds this bug by performing compositional reasoning, which allows covering interprocedural bugs while still scaling to millions of LOC. It deduces a precondition/postcondition specification approximating the behavior of `X509_gmtime_adj`, and then uses that specification when reasoning about its calls. The specification includes 0 as one of the return values, and this triggers the error." From the same deployment: "In 2017, we looked at bug fixes in several categories and found that for some (null dereferences, data races, and security issues) over 50% of the fixes were for bugs with traces that were interprocedural. The interprocedural bugs would be missed bugs if we only deployed procedure-local analyses."

Compositionality also decides when an analysis can run. "Infer does not need to process the entire codebase in order to analyze a diff, and so is fast." In contrast, "when Infer is run in whole-program mode it can take more than an hour (depending on the app)—too slow for diff-time at Facebook." When it ran changed the outcome. Run nightly over the whole code base, "the fix rate—the proportion of reported issues that developers resolved—was near zero". Run on each code change, "the fix rate rocketed to over 70%. The same program analysis, with same false positive rate, had much greater impact when deployed at diff time." A profile is enforced by the translator (P3589R3 §1.1), that is, at every compilation of every change.

Written summaries. At Microsoft, a modular checker for buffer overflows relied on annotations of function interfaces in an annotation language, SAL [Hackett 2005]: "Lightweight annotations specify requirements for safely using each buffer, and functions are checked individually to ensure they obey these requirements and do not overflow. To date over 400,000 annotations have been added to specify buffer usage in the source code for this product, of which over 150,000 were automatically inferred, and over 3,000 potential buffer overflows have been found and fixed." The checker ran in the build: "Developers receive immediate feedback about potential overflows in their code as part of the build process, and identified issues must be fixed before code can be added to mainline branches."

Portability. Analyzers and compilers also carry facts about library functions, each tool in its own form (§8). Each tool's set has its own coverage and its own approximations, and results computed from different facts differ. P3589R3 §1 describes the consequence: "Without a common framework, the options from different suppliers addressing common problems (e.g., range errors) will be significantly different, and code relying on one toolset cannot be relied on to work on another or to offer the same guarantees." P3274R0 states the requirement: "We must exactly specify the analysis required. We cannot have the validation of a Profile depend on the cleverness of individual implementations and vary significantly among implementations." A verdict depends on the rule and on the facts used at calls. A shared set of summaries makes those facts the same for every implementation.

5 C++ ALREADY SPECIFIES FUNCTIONS BY SUMMARIES

Declarations. Separate compilation already works this way for types. A call is checked against the callee's declaration, and the definition is checked against the same declaration. Several properties of a declaration are one-fact summaries: `noexcept`, `[[noreturn]]`, and the `const` qualification of a member function, which states what the function may write in `*this` and is checked at the definition. Function interface conditions are the preconditions and postconditions written in a function's declaration: conditions on the values at entry and at exit. They do not state footprints, resource effects, retained pointers or invalidation.

Profile proposals. P3446R0 proposes annotations on declarations, among them `[[not_invalidating]]`, `[[not_returned(logger)]]` and `[[not_local]]` on functions, and `[[invalidating]]` on parameters. Its reason: "Placing the annotation on the parameter, rather than on the use, means that we don't have to examine the definition of a function to ensure safety." A compiler can then "Catch uses of unintentional invalidation looking only at a single function definition in isolation", and P3446R0 concludes: "That is local static analysis." Each annotation is one claim of a summary, and P3446R0 asks that it be checked: "A `[[not_invalidating]]` annotation can (and should) be statically verified by examining the definition of a supposedly `[[not_invalidating]]` function."

The library specification. [structure.specifications]/3 says: "Descriptions of function semantics contain the following elements (as appropriate):". The elements are Constraints, Mandates, Constant When, Preconditions, Hardened preconditions, Effects, Synchronization, Postconditions, Result, Returns, Throws, Complexity, Remarks and Error conditions. When an Effects element says "Equivalent to", [structure.specifications]/4 determines the meaning compositionally: "the semantics of the code sequence are determined by the Constraints, Mandates, ... and Error conditions specified for the function invocations contained in the code sequence." This is summary-based reasoning: the meaning of a function is derived from the summaries of its callees.

The standard states library-wide defaults in the same way. [container.reqmts]/67 is a default claim about footprints and invalidation: "Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container." For basic_string, [string.require]/4 specifies the opposite for arguments: references, pointers and iterators to its elements "may be invalidated" by "Passing as an argument to any standard library function taking a reference to non-const basic_string as an argument." P3446R0's default for parameters is that of containers: a function does not invalidate an argument unless the parameter is marked [[invalidating]]. No one default serves both; a summary model states each library's defaults explicitly.

Invalidation by an individual function can depend on a condition. [vector.modifiers]/2 says: "Causes reallocation if the new size is greater than the old capacity. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence, as well as the past-the-end iterator. If no reallocation happens, then references, pointers, and iterators before the insertion point remain valid but those at or after the insertion point, including the past-the-end iterator, are invalidated." This fact has two outcomes; neither [[invalidating]] nor [[not_invalidating]] states it. Appendix A gives its summary.

What is missing. The pieces are in use. What is missing is one model of which they are parts, with one meaning; a form that tools can read; a way to attach facts to declarations that cannot be edited, such as the C library, operating system headers, existing libraries and the standard library as each implementation ships it (P3446R0's example "assumes that vector::operator[]() is declared [[non-invalidating]]"); and the facts themselves (§5).

6 EVIDENCE

We tested whether summaries can state what specifications and tools state today, against the C++ standard library and against the hand-written models of a production analyzer.

6.1 THE C++ STANDARD LIBRARY

To test whether summaries can state what the standard specifies, we wrote summaries for chapters 16 to 33 of the working draft, [library] to [exec], excluding Annex D, from its rendering at eel.is generated on 2026-08-23 [eel.is]. As of 2026-10-22, the corpus has 129 files containing:

- 5,709 function entries;
- 597 types;
- 163 concepts with their semantic requirements;
- 43 resource families, among them allocator storage, storage from operator new and operator new[], exception objects, coroutine frames, mutex ownership, shared states of futures, RCU read-side regions, hazard pointers, stop callbacks and operation states.

The elements of [structure.specifications]/3 map onto the parts of a summary:

Element	Part of the summary
---------	---------------------

Preconditions	requirements; violation is undefined
Hardened preconditions	requirements; in a hardened implementation, a violation is checked and stops the call before its other effects
Effects	footprint, postconditions, ordered steps, invocations, resource and lifetime effects; "Equivalent to" code analyzed over the callees' summaries
Synchronization	synchronization (acquire, release) of the steps
Postconditions, Returns	postconditions of the returning outcomes
Throws	throwing outcomes, with what holds after each
Error conditions	outcomes distinguished by the error code
Remarks	whichever part the remark states, for example invalidation ([vector.modifiers]/2)
Complexity	recorded; not a fact about states
Constraints, Mandates, Constant When, Result	compile-time conditions and properties of the declaration, checked by the translator

Writing the summaries also found problems in the standard. A summary requires a precise answer to the question "what may this call do?", and where the text does not give one, we cannot write the summary. We checked each problem against the draft sources at commit 47974a2 of the cplusplus/draft repository (2026-10-14) [draft], against the LWG issues lists revised on 2026-10-13 [LWG] and the open issues of the editorial repository, and against text elsewhere in the library clauses that might resolve it. The problems that remain are of two kinds:

- editorial: a synopsis and a detailed specification that disagree on a parameter name, an exception specification, a constraint or explicit; names and terms used without a definition; typographical errors;
- problems that need LWG resolution: effects that are not stated, such as the invalidation caused by a reallocation; exceptions that are not stated, such as allocation failure; states that are not stated after an exception, in particular one thrown by a user-supplied operation; results that are not stated at the edges of a function's domain, such as negative counts, NaN arguments, or the absence of a result; and two statements of one fact that disagree, such as an "Equivalent to" sequence and a general requirement, or an Effects element and a Throws element.

Some of the problems had already been reported by others. We have not filed the rest yet; a later revision of this paper will report them once they have been reviewed. Two examples of problems that need LWG resolution:

- [uninitialized.fill]/3 specifies uninitialized_fill_n as "Equivalent to: for (; n--; ++first) ::new (voidify(*first)) iterator_traits::value_type(x); return first;". With a negative n of signed type, n-- stays true, and the loop constructs objects until n overflows. Its siblings test n > 0; for example, [uninitialized.construct.default]/3 is "for (; n > 0; (void)++first, --n)". And [alg.fill]/1 says "Let N be max(0, n) for the fill_n algorithms".
- Container copy assignment and vector::resize beyond the capacity can reallocate, but no invalidation is stated for that reallocation. Read literally, [container.reqmts]/67 (quoted in §4) then says that these calls invalidate no iterators. An invalidation profile needs these facts.

Formalizing the specification of C++ concurrency had the same effect. Batty, Owens, Sarkar, Sewell and Weber found that the draft standards of the time "are not yet clear and rigorous definitions, and

harbour substantial problems in their details"; their work, presented to WG21 as N3132 (2010), had "already motivated changes to the draft standard" [Batty 2011].

6.2 THE HAND-WRITTEN MODELS OF GCC'S ANALYZER

To test whether summaries can state what an analyzer encodes by hand, we restated as summaries the function models of GCC's static analyzer, at commit 2e76f0ca473 of the GCC repository (2026-10-14) [GCC]. The analyzer registers each model under the name of a function. It has 234 registrations, implemented by 78 C++ classes: 186 in the files `kf.cc`, `kf-analyzer.cc` and `kf-lang-cp.cc`, and 48 in its state machines for file descriptors (`sm-fd.cc`), streams (`sm-file.cc`) and variadic arguments (`varargs.cc`).

Every registration maps to an entry. The 228 entries, in 18 files with 10 files of shared definitions, also cover the functions that the analyzer's state machines recognize by name without a registration, and the validator of §2 reports no problem in them. The C++ code of each model becomes a description that people can review and other tools can read.

Where the notation cannot yet state a fact exactly, the entry states a weaker fact that is still sound. For example, a call of the `printf` family whose format contains `%n` has an outcome that may write any location (writes: `["$unknown"]`), because the notation has no location set for writes directed by a format. As of 2026-10-22, the open gaps that we found are:

- writes directed by a format (`%n`, the conversions of `scanf`);
- effects internal to a replaceable function, as distinct from its visible effects, such as `operator new[]` calling `operator new`;
- lifetimes bounded by the caller's frame (`alloca`), storage allocated after a stack save, and the limit on the size of the stack;
- a comparison of addresses only: C++ `memcpy` returns an object created at the address of its first argument;
- results that a later call may overwrite (`getenv`), stated today by a stricter rule;
- atomic steps that pass the value they read to a later step, and memory orders given as run-time integers, as the GCC built-ins take them;
- descriptions shared by several declarations whose types differ, such as the sized atomic built-ins; the compiler's internal functions; options of the sanitizers;
- terms for the type that a `type_info` object denotes, for the value of a `std::align_val_t`, and for the alignment of storage obtained from the `operator new` that takes no alignment;
- the state of streams and signals: a stream's mode and locks, the signal mask, a non-blocking connect that completes later, and the rules that apply to signal handlers.

7 TRUST

A summary is a claim about code. How far a checker may rely on it depends on where the claim was checked.

- Verified: the definition was compiled together with the summary and checked against it. Every path ends in one of the outcomes, every write is in the footprint, and every other claim holds. P3446R0 states the same principle: "Wherever possible, verify annotations."

- Assumed: the definition is not available, as with a prebuilt library or the operating system. The summary is then part of the trusted base, and a result lists it, in the form "all obligations proven, assuming summaries S1, ..., Sn".

A definition that no longer satisfies its summary is reported where it is compiled, like any other violation of an enforced profile. A justification string in a suppression is an unchecked claim at one use. A summary is one claim per function, checked where the function is defined and used at every call.

Summaries do not replace the other sources of facts. They are the common form into which those sources translate, each with its provenance recorded: function interface conditions, as requirements and postconditions; attributes, such as `noexcept`, `[[noreturn]]`, the annotations of P3446R0 and Clang's `lifetimebound`; and summaries that tools infer from function bodies. For functions whose definitions are compiled with the checker, summaries can be inferred and then verified, so authors write summaries mainly at the boundaries of libraries.

8 SUMMARIES IN THE PROFILE FRAMEWORK

Summaries need no change to P3589R3. This section shows how they serve its mechanisms as specified, with header files and with modules.

Enforcement on a translation unit and its headers. In P3589R3, "the dominion of a profile enforcement is the set of tokens starting from the end of that attribute till the end of the translation unit containing the attribute" (§1.1.1). The dominion includes the headers that the unit includes, as in P3589R3's own example:

```
[[profiles::enforce(std::lib::hardened)]];
[[profiles::enforce(acme::hardened(fortify: 3, sanitize: thread))]];
#include <my/lib.h>
String greet(const Neighbor& n) { /* ... */ }
```

The analysis covers "the complete abstract semantic graph of a translation unit". Summaries supply what that graph does not contain: the facts about the functions that the unit calls and does not define, whether they are declared in a header or in a module interface.

Header files. Here is normalize of §1, declared in a header, defined in one source file and called in another:

```
// kai/utils.h
#include <vector>
void normalize(std::vector<double>& v);    // scales the elements

// kai/utils.cpp
[[profiles::enforce(acme::lifetime)]];
#include "kai/utils.h"
void normalize(std::vector<double>& v) { for (double& d : v) d *= 2; }

// client.cpp
[[profiles::enforce(acme::lifetime)]];
#include "kai/utils.h"
double scaled_front(std::vector<double>& v)
{
    double& x = v.front();
    normalize(v);
}
```

```
    return x;    // valid if normalize does not invalidate the elements of v
}
```

The summary of `normalize` is supplied with the header, for example as a file installed beside it, and is matched to the declaration by name and signature. It states that the call writes the elements of `v` and invalidates nothing. `kai/utils.cpp`, compiled under enforcement, is where the summary is verified against the definition (§6). `client.cpp` uses the summary at the call, and is checked without the definition. If `client.cpp` is compiled without a record that the summary was verified, its result lists the summary as an assumption (§6). Nothing in this depends on modules.

Modules and header units. P3589R3 §1.1 lists two aims of the framework:

- 4. Provide a mechanism for a library to express an interface along with guarantees that it offers through a module interface, etc.*
- 5. Provide a mechanism for a program to take dependencies on other components (modules, libraries, etc.) and to request specific guarantees from those components*

In P3589R3, `import Kai.Utils [[profiles::require(std::type)]]`; is valid when the module declaration of `Kai.Utils` names `std::type` in a profile-enforcement attribute (§1.1.5, §2.3), and "The same behavior is expected if, instead of a named module, a header unit was imported." When a module is compiled under enforcement, the definitions of its exported functions are compiled under enforcement too, so an implementation can verify their summaries there. The compiled module interface can then carry the summaries, as it carries the declarations; the standard does not specify what a compiled module interface contains. A header unit can carry the summaries supplied with its header. An importer uses the summaries at each call; for a module, the enforcement that require checks is what makes them verified. For the standard library, the implementation would provide one set of summaries, used alike by `#include` and by `import std`;

Exemption. The headers that P3589R3 §1.1.6 exempts from enforcement (its examples are and a header for a foreign-function interface) are served by summaries supplied separately from them. The functions that they declare are usually compiled elsewhere, as those of the C library and of the operating system are, so their summaries are assumed (§6). Tools already attach separate files to existing declarations, for example Clang's API notes and CodeQL's library models (§8).

Suppression. A summary turns the justification of a suppression into a checked fact (§2). A rule stated in terms of what calls do needs fewer suppressions than a rule stated in terms of declarations (§1.1).

One model for all profiles. Rules for initialization, bounds, lifetime and invalidation, resource release and concurrency ask different questions of the same call: what it writes, what it requires, and what it invalidates, retains, releases and synchronizes. If each profile defined its own annotations, each would have its own semantics of a call, and their interactions would have to be specified pair by pair.

9 CONTEMPORARY WORK

Function summaries are in use today, in pieces: separate files that attach facts to declarations, annotations that compilers check, and models built into analyzers. This section compares the proposal with them.

9.1 CLANG API NOTES

Clang's API notes are "Annotations Without Modifying Headers" [API notes]. Their documentation states the problem: "You have headers you want to use, but you also want to add extra information to the API. You don't want to put that information in the headers themselves". The solution is a "sidecar" file in "a YAML-based file format", found next to the module map of the module that it annotates when the compiler is passed `-fapinotes-modules`. Its entries name declarations, including C++ namespaces, classes, methods and fields, and attach annotations to them. The original description of the feature gives a C example: adding "a 'noreturn' attribute to the C 'exit' function", because "we can't go around and fix all of the system headers everywhere" [API notes RFC].

API notes and summaries are similar in these respects:

- both are separate files that attach facts to declarations that cannot be edited, the case of §7;
- both are YAML, with entries matched to declarations by name;
- both are selected by version: an API note by the Swift language version, a summary by library version, language standard, target and macros;
- both are tied to the way declarations reach the compiler: API notes are "picked up by the module-building logic in the compiler", and §7 supplies summaries with header files and in compiled module interfaces;
- some facts are the same. Nullability of parameters and results is a requirement and a postcondition. NoEscape on a block parameter is a claim about retained pointers. SwiftReturnOwnership: retained, under which "the caller receives an owned reference", is an acquisition. SwiftRetainOp and SwiftReleaseOp name the operations of a reference-counted type, as a summary names the operations of a resource family.

They differ in these respects:

- An API note states what an attribute, a macro or a type written in the header could state; the reference documents many keys as "Equivalent to" one. It has no outcomes, no conditions relating values at entry and at exit, no footprints, no invalidation and no invocations. For the read of §2, an API note can mark buf non-null. It cannot state that $n \geq 0$ is required, which elements are written, or that the call has two outcomes.
- Most keys control how an API is imported into Swift: "Many API notes affect how a C API is imported into Swift."
- API notes apply to modules only: "Since they're identified by module name, API notes cannot be used to modify arbitrary textual headers." Summaries are matched to declarations by name and signature, so they apply to declarations in header files as well as in modules (§7).
- An API note takes effect as if its attribute were written on the declaration, and is checked only as far as that attribute is. A summary is verified as a whole against the definition, where the definition is compiled (§6).

API notes show that a compiler can attach facts to existing declarations from separate files during normal compilation. Summaries extend what such files state, from attributes to behavior.

9.2 ANNOTATIONS THAT COMPILERS CHECK

Clang's Lifetime Safety Analysis "is a C and C++ language extension which warns about potential dangling pointer defects in code" [Clang lifetime]. It "tracks pointer validity through intra-procedural data-flow analysis", and meets the choice of §1 at calls. Without annotations, "the analysis treats function calls optimistically, assuming no lifetime effects, thereby potentially missing dangling pointer issues". With annotations such as `[[clang::lifetimebound]]` and `[[clang::noescape]]`, it can "enforce lifetime safety at call sites with higher accuracy". It validates annotations against definitions, and it can infer `lifetimebound`. This is the architecture that the proposal generalizes: local analysis, facts at calls, facts verified at definitions, and inference, here with a small fixed set of facts. The documentation describes no annotation for invalidation; the experimental invalidation check covers operations of standard containers and of owners such as `std::unique_ptr`.

Clang's Thread Safety Analysis [Clang thread safety] "is strictly intra-procedural, just like ordinary type checking. It relies only on the declared attributes of a function". Its attributes are requirements and resource effects for one kind of resource, capabilities such as locks. For example, `ACQUIRE` means that "The given capability must not be held on entry, and will be held on exit", and `TRY_ACQUIRE` is for a function that "returns a boolean value indicating success or failure", an outcome distinguished by the result. The analysis "is used extensively in Google's internal code base".

Microsoft's SAL "provides a set of annotations that you can use to describe how a function uses its parameters, the assumptions that it makes about them, and the guarantees that it makes when it finishes" [SAL]. It comes closest to the read summary of §2. Under `_Success_(expr)`, "each annotation (anno) on the function and in post-condition behaves as if it were coded as `When_(expr, anno)`", and `_Out_writes_to_(s,c)` states that "In post-state, the elements up to the c-th element must be valid." A function has one success condition: "There can be at most one Success annotation on a function". A summary can have any number of outcomes, each with its own exit and footprint, such as the four of Appendix A.

Other annotations state single facts (§8.4): Clang's `nonblocking` and `nonallocating`, the element counts of `-fbounds-safety`, and GCC's `access`, `malloc` (deallocator) and `fd_arg`. Each annotation of this section is part of a summary, with its own semantics, and translates into a summary with its provenance (§6).

9.3 MODELS IN ANALYZERS

Analyzers carry their own models of library functions. A checker of Clang's static analyzer contains summaries in its source [Clang-SLF], and the analyzer's taint analysis reads rules for functions from a YAML file [Clang taint]. GCC's analyzer has 234 registered models of individual functions, implemented by 78 C++ classes (§5.2). CodeQL reads "models (summaries, sinks, and sources) of your code's C and C++ dependencies in data extension files", written in JSON or YAML, each with its provenance [CodeQL

MaD]. Clang's Scalable Static Analysis Framework, "a framework for writing cross-translation unit analyses in a scalable and extensible setting", writes the summaries that it extracts from each translation unit to a file, for example in JSON [Clang SSAF]. Each set has its own format, semantics and coverage. Written models are trusted, not checked against the definitions that they describe, and inferred ones serve the analysis that computes them.

9.4 COMPARISON

Work	Facts stated	Where written	Checked against the definition
Clang API notes [API notes]	attributes and types: nullability, non-escaping blocks, ownership of results, retain and release operations; mostly for import into Swift	YAML file beside a module map	as far as each attribute is
Clang lifetime safety [Clang lifetime]	what a result may refer to; non-escaping parameters; owner and pointer types	attributes in the source	yes
Clang thread safety [Clang thread safety]	capabilities required, acquired and released; one acquisition conditional on the result	attributes in the source	yes
Clang function effects [Clang function effects]	the function does not allocate, or does not block	attributes on function types	yes
-fbounds-safety [Clang bounds safety]	element counts of pointers at interfaces	type annotations in the source	yes, at compile time or by run-time checks
GCC attributes [GCC attributes]	access modes and maximum sizes; deallocators; requirements on file descriptors	attributes in the source	by diagnostics, as far as each attribute is
SAL [SAL]	buffer sizes and initialization; value ranges; one success condition	annotations in the source	yes
Written models (Clang, GCC, CodeQL)	each tool's own: value ranges, taint flow, state transitions	the tool's source code, or YAML or JSON files	no; they are trusted
Inferred summaries (Infer, Clang's framework)	each analysis's own	computed by the tool	derived from it
This proposal	requirements, outcomes, footprints, resource and lifetime effects, retained pointers, invalidation, invocations	YAML files supplied with headers, or compiled module interfaces	yes, where the definition is compiled

Every element of the proposal is in contemporary use: separate files (API notes, CodeQL's data extensions, Clang's taint configuration), YAML and JSON, facts checked against definitions (lifetime safety, thread safety, function effects, SAL), inference (lifetime safety, Infer, Clang's framework), and models of library functions in every analyzer. Contemporary work does not have one model in which these facts have one meaning, that states conditional behavior as outcomes, and that compilers, analyzers and the authors of libraries and of their specifications share. That is what this paper proposes.

10 CONCLUSIONS

1. At a call to a function defined elsewhere, a local analysis that has only the declaration must reject some correct programs or accept some incorrect ones (§1). Summaries let the rules of the profile proposals keep their facts across such calls (§1.1).
2. The facts that resolve it are statements of what the callee does, which specifications state and declarations do not (§4).
3. Industrial analyzers reason with such statements, inferred or written. Compositionality also lets an analysis run on each change, where the same analysis had a fix rate over 70%, against near zero when run nightly (§3).
4. Summaries can state what the standard specifies for its whole library, and what an analyzer's hand-written models encode. Writing them is also a review of the specification: it found effects, exceptions and results that the text does not state, and statements of one fact that disagree (§5).
5. A summary is verified where its function's definition is compiled, and is otherwise listed as an assumption (§6). Summaries serve the mechanisms of P3589R3 as specified, for header files and modules alike (§7).
6. Contemporary compilers and analyzers state parts of summaries, in separate files or in annotations, checked or trusted, each with its own semantics. None states them in one model shared across tools (§8).

The question for code analysis of C++, and for profiles, is therefore not whether a local analysis can see across calls; by itself, it cannot. The questions are where the facts at calls come from, who states them, and how they are checked. Function summaries answer all three. The facts come from interfaces and specifications. They are stated once per function, by the authors of the function or of its specification, or inferred by tools. They are checked against definitions where the definitions are compiled.

11 REFERENCES

- [P3589R3] G. Dos Reis, "C++ Profiles: The Framework", P3589R3, 2025-12-01. <https://open-std.org/JTC1/SC22/WG21/docs/papers/2026/p3589r3.pdf>
- [P3274R0] B. Stroustrup, "A framework for Profiles development", P3274R0, 2024-05-05. <https://open-std.org/JTC1/SC22/WG21/docs/papers/2024/p3274r0.pdf>
- [P3446R0] B. Stroustrup, "Profile invalidation – eliminating dangling pointers", P3446R0, 2024-10-14. <https://open-std.org/JTC1/SC22/WG21/docs/papers/2024/p3446r0.pdf>
- [P3081R1] H. Sutter, "Core safety profiles for C++26", P3081R1, 2025-01-06. <https://open-std.org/JTC1/SC22/WG21/docs/papers/2025/p3081r1.pdf>
- [Cousot 2002] P. Cousot, R. Cousot, "Modular Static Program Analysis", CC 2002, LNCS 2304, pp. 159–178.
- [Distefano 2019] D. Distefano, M. Fähndrich, F. Logozzo, P. W. O'Hearn, "Scaling Static Analyses at Facebook", Communications of the ACM 62(8),

pp. 62–70, 2019. doi:10.1145/3338112

- [Hackett 2005] B. Hackett, M. Das, D. Wang, Z. Yang, "Modular Checking for Buffer Overflows in the Large", Microsoft Research technical report TR-2005-139, 2005. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-139.pdf>
- [ACSL] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, "ACSL: ANSI/ISO C Specification Language", version 1.24, CEA-List and Inria. <https://frama-c.com/download/acsl.pdf>
- [GCC] GCC repository, commit 2e76f0ca47367a2a90307bf1c5f8805a37138d84 (2026-10-14), files gcc/analyzer/kf.cc, kf-analyzer.cc, kf-lang-cp.cc, sm-fd.cc, sm-file.cc and varargs.cc. <https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=2e76f0ca47367a2a90307bf1c5f8805a37138d84>
- [Clang-SLF] Clang static analyzer, StdLibraryFunctionsChecker.cpp, header comment. https://clang.llvm.org/doxygen/StdLibraryFunctionsChecker_8cpp_source.html
- [API notes] "API Notes: Annotations Without Modifying Headers", Clang 24.0.0git documentation. <https://clang.llvm.org/docs/APINotes.html>
- [API notes RFC] G. Horváth, "[RFC] Adding API Notes", cfe-dev, 2017-05-17. <https://lists.llvm.org/pipermail/cfe-dev/2017-May/053860.html>
- [Clang lifetime] "Lifetime Safety Analysis", Clang 24.0.0git documentation. <https://clang.llvm.org/docs/LifetimeSafety.html>
- [Clang thread safety] "Thread Safety Analysis", Clang 24.0.0git documentation. <https://clang.llvm.org/docs/ThreadSafetyAnalysis.html>
- [Clang function effects] "Function Effect Analysis", Clang 24.0.0git documentation. <https://clang.llvm.org/docs/FunctionEffectAnalysis.html>
- [Clang bounds safety] "-fbounds-safety: Enforcing bounds safety for C", Clang 24.0.0git documentation. <https://clang.llvm.org/docs/BoundsSafety.html>
- [Clang taint] "Taint Analysis Configuration", Clang 24.0.0git static analyzer documentation. <https://clang.llvm.org/docs/analyzer/user-docs/TaintAnalysisConfiguration.html>
- [Clang SSAF] "Scalable Static Analysis Framework", Clang 24.0.0git documentation. <https://clang.llvm.org/docs/ScalableStaticAnalysis/index.html>
- [GCC attributes] "Common Attributes", GCC manual, entries access, malloc and fd_arg; gcc/doc/extend.texi at the commit of [GCC]. <https://gcc.gnu.org/onlinedocs/gcc/Common-Attributes.html>
- [SAL] "Understanding SAL", "Annotating function parameters and return values" and "Annotating function behavior", Microsoft Learn. <https://learn.microsoft.com/en-us/cpp/code-quality/understanding-sal>
- [CodeQL MaD] "Customizing library models for C and C++", CodeQL documentation. <https://codeql.github.com/docs/codeql-language-guides/customizing-library-models-for-cpp/>

- [N3132] M. Batty, S. Owens, S. Sarkar, P. Sewell, T. Weber, "Mathematizing C++ Concurrency: The Post-Rapperswil Model", N3132, 2010-08-23.

<https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3132.pdf>

- [Batty 2011] M. Batty, S. Owens, S. Sarkar, P. Sewell, T. Weber, "Mathematizing C++ Concurrency", POPL 2011, pp. 55–66.
- [eel.is] Working Draft, Programming Languages — C++, rendered at <https://eel.is/c++draft/>, generated on 2026-08-23 from the LaTeX sources.

Paragraph numbers in this paper are those of this rendering.

- [draft] cplusplus/draft, the LaTeX sources of the working draft, commit 47974a2fafeca9a2207d3b5eff3dfd65ce771277 (2026-10-14).

<https://github.com/cplusplus/draft/commit/47974a2fafeca9a2207d3b5eff3dfd65ce771277> Every paragraph quoted in this paper was checked to be unchanged in these sources.

- [LWG] C++ Standard Library Issues Lists, revision D127, revised on 2026-10-13.
<https://cplusplus.github.io/LWG/lwg-active.html>

The documentation of Clang (version 24.0.0git, the development version),

CodeQL and Microsoft Learn cited above was read on 2026-10-22.

12 APPENDIX A. A SUMMARY OF `std::vector::push_back(const T&)`

This entry is simplified from our corpus. Three things are omitted:

- the facts it inherits from the container requirements it refines (the values of the kept elements and of the new element);
- its calls to the allocator;
- one outcome kept for a literal reading of `[vector.capacity]/7` and `[vector.modifiers]/2`. Under that reading, a vector that has spare capacity but was never reserved may still reallocate.

```
- id: std.vector.push_back
match: { cxx_member: { class_template: "std::vector", member: push_back } }
signature: "constexpr void push_back(const T& x);"
outcomes:
  - name: appended                                # no reallocation
    when: "$size (*this) < $capacity (*this)"
    ensures:
      - "$size (*this) == $size (*this)@entry + 1"
      - "$data (*this) == $data (*this)@entry"
      - "$capacity (*this) == $capacity (*this)@entry"
    writes: ["$size (*this)", "$data (*this)[$size (*this)@entry]"]
    # only the past-the-end iterator: iterators whose position was the old end
    invalidates: [{ to: "$pos == $size (*this)@entry", may_be: singular }]
  - name: reallocated
    when: "$size (*this) == $capacity (*this)"
    ensures:
      - "$size (*this) == $size (*this)@entry + 1"
      - "$capacity (*this) > $capacity (*this)@entry"
    writes: ["$size (*this)", "$capacity (*this)", "$data (*this)"]
```

```
invalidates:
  - "$elements ($data (*this)@entry, $std_range (0, $size (*this)@entry))"
  - { to: "$pos == $size (*this)@entry", may_be: singular }
- name: threw_no_effects
  exit: "throw(...)"
  writes: []
- name: move_threw # "the effects are unspecified"
  when_type: "not Cpp17CopyInsertable<T> and not $nothrow_move_constructible (T)"
  exit: "throw(...)"
  writes:
    - "$size (*this)"
    - "$capacity (*this)"
    - "$data (*this)"
    - "$elements ($data (*this)@entry, $std_range (0, $size (*this)@entry))"
  invalidates:
    - "$elements ($data (*this)@entry, $std_range (0, $size (*this)@entry))"
    - { to: "$pos == $size (*this)@entry", may_be: singular }
  invariants: unknown
```

The outcomes follow [vector.modifiers]/2. `appended` and `reallocated` are its two invalidation cases, with "the new size is greater than the old capacity" written as `$size (*this) == $capacity (*this)` at entry. `threw_no_effects` is "there are no effects". `move_threw` is "Otherwise, if an exception is thrown by the move constructor of a non-`Cpp17CopyInsertable` `T`, the effects are unspecified": the call may have written and invalidated anything in the vector, and the vector's invariants are not known to hold.

Suppose a checker sees `v.reserve(v.size() + 1)` before `v.push_back(x)`. It can keep a reference to `v[0]` valid across a call that returns normally, because `reallocated` is then excluded.