Static Analysis with ClangIR

Ivan Murashko

Contents

This article explores ClangIR, an intermediate representation that bridges Clang's AST and LLVM IR. ClangIR is implemented as an MLIR dialect. It keeps source-level C and C++ information for longer than LLVM IR while still giving us SSA values, regions, and structured control flow.

We will study ClangIR through one concrete analysis pass: `LifetimeCheck`. The pass detects lifetime bugs such as use-after-move and use-after-free by implementing parts of the C++ Core Guidelines lifetime safety profile. It is a good example because it needs both worlds at the same time: AST semantic information such as type categories, smart pointer detection, and move semantics; and IR information such as value flow and structured control flow.

1 MLIR and intermediate representations

We will start with an Intermediate Representation, or IR. It is one of the key components of a modern compiler. The IR allows the compiler to analyze the program and to perform transformations before final code generation. The best known example for us is LLVM IR. Clang generates LLVM IR, and then LLVM middle-end and back-end passes optimize and lower it.

The problem is that LLVM IR is already too low level for some C++ analyses. As soon as we lower from AST to LLVM IR, we lose direct access to many AST facts: declarations, references, class methods, special member functions, and standard library types are no longer represented in the same way. On the other hand, some analyses require better control-flow and data-flow structure than the AST itself provides. This is the gap where Multi-Level Intermediate Representation, or MLIR, becomes useful. ClangIR is an MLIR dialect designed to keep C and C++ semantics before lowering to LLVM IR.

1.1 Static analysis: AST vs IR

A common Clang static analysis starts from the control-flow graph, or CFG. Clang builds the CFG on top of the AST. Thus the analysis is still based on the AST as the primary data structure. The AST is not always the best data structure for static analysis. Let us recall why. The parser and semantic analyzer give us two kinds of information:

1. Syntax analysis
2. Semantic analysis

The first one represents the syntactic structure of the program. The AST is the data structure designed to represent it. Semantic analysis is about the meaning of the program. For static analysis we also have to analyze control flow, i.e., how statements are executed, and data flow, i.e., how values are propagated through the program. The AST is built with knowledge about program semantics, but most of this semantic information comes from declarations and types, not from an explicit control-flow representation.

> **Important note.**
>
> Static Single Assignment (SSA) is a form of program representation that simplifies the static analysis and program optimizations. SSA is the key component of LLVM IR. In a program written in SSA form, each variable has only one assignment. Consider the following program fragment:

```
x = 1;
y = x + 1;
x = 2;
z = x + 1;
```

The obvious conclusion may be $z = y$ because they have the same right hand side, but this reasoning is false. After careful consideration we should conclude that $z \neq y$.

The same program in SSA form is as follows:

```
x1 = 1;
y = x1 + 1;
x2 = 2;
z = x2 + 1;
```

Now even a trivial analysis concludes that $z \neq y$ because the two expressions use different SSA values.

Thus SSA makes reasoning about program meaning easier than in the non-SSA case. Therefore, it simplifies program analysis and compiler optimizations.

Program flow is important for many optimizations performed by the compiler middle-end and back-end. For that reason, compilers use intermediate representations that make control flow and data flow more explicit than the source AST.

1.2 MLIR fundamentals

MLIR stands for Multi-Level Intermediate Representation. It is a framework for building domain-specific IRs on top of LLVM infrastructure. MLIR has many dialects, and these dialects can be combined in one program. The dialects can be used for different domains, for example:

- Accelerators (amdgpu, openmp, etc.)
- Parallel programming (triton)
- High level IRs (ClangIR, VAST)

One of the most important dialects is the *llvm* dialect, which is used as a path toward LLVM IR. It is worth noting that the conversion is staged. A program can be lowered into the MLIR LLVM dialect and then translated to LLVM bitcode or assembly.

Dialects can coexist. For example, ClangIR can be combined with OpenMP-related dialects while the compiler is still working at a level higher than LLVM IR. As an IR framework, MLIR also provides infrastructure for transformations, diagnostics, source locations, and pass management.

High-level IRs are the most interesting for this article. For instance, VAST uses MLIR to represent an AST-like program form that can later be used for advanced static analysis. MLIR has several properties that make it suitable for static analysis. The most important are:

- MLIR operations normally carry source locations, which helps diagnostics.
- MLIR dialects can define higher-level operations that are easier to analyze than a raw AST for control-flow-based tasks.
- MLIR regions and blocks give us a structured way to represent execution order and nested control-flow constructs.

2 ClangIR basics

2.1 ClangIR architecture

ClangIR, or CIR, is a Clang-native MLIR dialect designed to preserve C/C++ semantics before lowering to LLVM IR. The dialect definition and op/type/attr registration live in:

```
clang/lib/CIR/Dialect/IR/CIRDialect.cpp
clang/include/clang/CIR/Dialect/IR/CIRDialect.h
```

CIR's frontend builds MLIR directly from the Clang AST (not from LLVM IR). Entry points are `cir::CIRGenerator` and the frontend action:

```
clang/include/clang/CIR/CIRGenerator.h
clang/lib/CIR/FrontendAction/CIRGenAction.cpp
```

Keeping a C/C++-semantic IR makes it easier to build language-aware optimizations and analyses before aggressive lowering destroys structure.

Clang CIR is built directly from the Clang AST. The CIR generator walks AST declarations and statements to emit CIR ops. Those ops carry language semantics via CIR-specific types, attributes, and interfaces. Per-translation-unit state in CIRGenModule keeps mappings from AST entities to CIR constructs. MLIR provides the infrastructure to register the CIR dialect and run passes. AST-derived metadata is preserved through early CIR passes. A `DropAST` pass removes AST attachments before final emission. This keeps source-level meaning intact while enabling staged lowering.

2.2 AST attribute access in ClangIR

CIR stores AST pointers inside CIR attributes, and the analysis accesses them through AST attr interfaces on operations and types before `DropAST` runs.

How it is done:

- CIR ops/types carry AST attrs like `ASTFunctionDeclAttr`, `ASTVarDecl Attr`, `ASTRecordDeclAttr`. See `CIRAttrs.td`.
- The interfaces are in `ASTAttrInterfaces.h` and the matching `.td` file. They expose helpers like `getDeclName()`, `getTLSKind()`, etc., and use `getAst()` under the hood.

Typical access patterns:

- From a CIR op (e.g., function op):

```
#include "clang/CIR/Interfaces/ASTAttrInterfaces.h"

auto astAttr = funcOp.getAstAttr(); // OptionalAttr on the op
if (auto funcDeclAttr =
↪  mlir::dyn_cast<cir::ASTFunctionDeclAttr>(astAttr)) {
  const clang::FunctionDecl *FD = funcDeclAttr.getAst();
  // Use FD or interface methods, e.g.
  ↪  funcDeclAttr.getMangledName()
}
```

- From a CIR type (e.g., record type):

```
auto recordType = mlir::dyn_cast<cir::RecordType>(ty);
if (recordType) {
  auto astAttr = recordType.getAst();
  if (astAttr) {
    const clang::RecordDecl *RD = astAttr.getRawDecl(); //
    ↪  debug-focused
  }
}
```

2.3 Build and installation

For our experiments we will use our basic Debug configuration as the start point. We need to enable two projects: `clang` and `mlir`. Also we require the configuration option `CLANG_ENABLE_CIR` to be set on:

`-DCLANG_ENABLE_CIR=ON`. The resulting configuration we will use for builds will look like:

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug
  -DCMAKE_INSTALL_PREFIX=../install
  -DLLVM_ENABLE_PROJECTS="clang;mlir;clang-tools-extra"
  -DLLVM_USE_SPLIT_DWARF=ON -DBUILD_SHARED_LIBS=ON
  -DCLANG_ENABLE_CIR=ON ../llvm
```

Figure 1: CMake configuration that enables ClangIR

You can run the ClangIR tests with:

```
$ ninja check-clang-cir
```

2.4 Building and testing

The LifetimeCheck pass and the tests used in this series refer to the ClangIR incubator checkout. To build ClangIR with the LifetimeCheck pass enabled, ensure your CMake configuration includes `-DCLANG_ENABLE_CIR=ON`, as shown in Figure 1.

Run the lifetime check tests:

```
$ ninja -C build clang
$ ninja -C build check-clang-cir
$ ./build/bin/llvm-lit \
  clang/test/CIR/Transforms/lifetime-check-use-after-move.cpp
$ ./build/bin/llvm-lit \
  clang/test/CIR/Transforms/lifetime-check-smart-pointer-after-move
  .cpp
```

Figure 2: Building ClangIR and running LifetimeCheck tests

Test files use the standard LIT format with `RUN` and `CHECK` directives:

```
// RUN: %clang_cc1 -std=c++17 -triple x86_64-unknown-linux-gnu \
// RUN:   -fclangir -fclangir-lifetime-check="history=invalid,null"
  \
// RUN:   -clangir-verify-diagnostics -emit-cir %s -o %t.cir

void testUseAfterMove() {
  std::unique_ptr<int> ptr = std::make_unique<int>(42);
```

```
7     std::unique_ptr<int> ptr2 = std::move(ptr);
8     int value = *ptr;  // expected-warning {{use of invalid pointer}}
9                        // expected-note@-2 {{became invalid here}}
10 }
```

The smart-pointer tests use small test-local stand-ins annotated with `[[gsl::Owner(T)]]`. The examples in the article keep the familiar `std::unique_ptr` name when explaining the lifetime model, but standard-library owner recognition requires explicit modeling rather than being inferred from `<memory>` by itself.

2.5 Compiler passes overview

Before diving into the LifetimeCheck analysis implementation, it is important to understand the concept of compiler passes and how ClangIR fits into the pass infrastructure.

A compiler pass is a discrete phase of compilation that transforms or analyzes the intermediate representation (IR) of a program. Modern compilers organize their work into multiple passes, each with a specific responsibility.

Compiler passes are useful because they let us focus on one task at a time (modularity), combine passes in different orders (reusability), and enable or disable them through compiler options.

Passes can perform different tasks. One important application is optimization: for example, the **-O3** flag selects a set of optimization passes. Another application is transformation, such as dead code elimination or inlining. In this article we are mostly interested in analysis passes.

Analysis passes can answer different questions. A pass may analyze control flow, data flow, aliasing, ownership, or lifetime. The LifetimeCheck pass is in this last group.

3 The lifetime safety problem

> **Important note.**
>
> The code examples in this section use API names and methods from the ClangIR implementation, especially `LifetimeCheck.cpp`. Some validation logic and edge case handling is simplified for clarity. Simplified sections are marked with comments. The focus is on demonstrating analysis patterns rather than production-ready error handling.

In this section, we will walk through a practical analysis implemented for ClangIR: the LifetimeCheck pass. The goal is to detect lifetime bugs such as

use-after-move and use-after-free while preserving enough semantic context to explain why the bug is unsafe.

3.1 Motivation for lifetime analysis

One of the most dangerous classes of bugs in C++ programs involves incorrect lifetime management: using pointers after the memory they point to has been freed (use-after-free), accessing moved-from objects (use-after-move), or dereferencing dangling references. Consider this simple example:

```
1 std::unique_ptr<int> ptr = std::make_unique<int>(42);
2 std::unique_ptr<int> ptr2 = std::move(ptr);
3 int value = *ptr;  // ERROR: use-after-move!
```

After the move on line 2, `ptr` is in a moved-from state (equivalent to null for smart pointers). Dereferencing it on line 3 is undefined behavior. While this example is trivial, such bugs become much harder to spot in real code with complex control flow, function calls, and multiple indirections.

The C++ Core Guidelines define a lifetime safety profile [1] that provides rules for detecting such issues, as formalized in the P1179 paper [2]. The LifetimeCheck pass implements part of this profile. It demonstrates why ClangIR is interesting for static analysis: it gives us AST semantic information and an IR structure in the same place.

```cpp
#include <memory>

void example() {
  std::unique_ptr<int> owner = std::make_unique<int>(42);
  int *ptr = owner.get();      // ptr -> valid memory
  std::unique_ptr<int> moved = std::move(owner);

  if (owner) {                 // OK: operator bool is safe
    // Not reached after the move
  }

  *owner;                      // ERROR: operator* requires non-null
  *ptr;                        // OK: memory is now owned by 'moved'
}
```

Line	Category	LifetimeCheck analysis
4	Owner	pmap[owner] = {owned_object}, owners.insert(owner)
5	Pointer	pmap[ptr] = {owned_object}, ptrs.insert(ptr)
6	Move	pmap[owner] = {invalid}, pmap[moved] = {owned_object} owner now moved-from, ptr points to valid memory owned by moved
8	Check	isSmartPointerSafeMethod("operator bool") = true (OK)
12	Check	isSmartPointerSafeMethod("operator*") = false (ERROR) **ERROR**: use of invalid pointer 'owner'
13	No issue	ptr points to valid memory owned by moved No warning: memory remains valid while moved is in scope

Figure 3: Lifetime analysis example with detailed tracking

The source in Figure 3 is the C++ pattern we want the checker to reason about. The row-by-row table describes the intended LifetimeCheck state transitions and matches the ClangIR tests from this point when the smart-pointer type is represented by an annotated test class. A plain system `std::unique_ptr` is not classified as an Owner by this pass without such an annotation.

Important note.

Note that dereferencing `ptr` on line 13 does not generate a lifetime warning. While `owner` has been moved-from, the memory it originally

owned is still valid because it is now owned by `moved`. The lifetime checker correctly recognizes that this dereference is safe as long as `moved` remains in scope. This demonstrates the precision of the analysis—it tracks actual object lifetimes, not just moved-from status of the original owner.

3.2 Why ClangIR for lifetime analysis

Traditional approaches face trade-offs:

- **AST-based analysis** has full semantic information but poor control flow representation. Building a CFG from AST is complex.
- **LLVM IR-based analysis** has excellent control flow (SSA form) but loses high-level semantics—smart pointers become raw pointers, and move operations are hard to identify.

The LifetimeCheck example shows why ClangIR is useful:

- **AST attributes** on operations provide semantic information: "Is this a move constructor?" and "Is this type a std::unique_ptr?"
- **SSA form and structured control flow** simplify dataflow analysis.
- **High-level operations** preserve C++ semantics: `StoreOp`, `LoadOp`, and `CallOp` carry more meaning than LLVM's generic instructions.

Feature	Clang AST	ClangIR	LLVM IR
Control flow	Nested AST statements	Structured operations	Basic blocks
SSA form	No	Yes	Yes
Type information	Full C++ types	Preserved C++ types	Lowered types
Smart pointers	AST node types	AST attrs	Lost in lowering
Move semantics	Explicit AST nodes	Explicit CIR attrs	Usually lost
Source locations	Always available	Required	Optional metadata
Dataflow	Requires CFG	Natural	Natural

Figure 4: Comparison of analysis capabilities across representations

AST represents control flow implicitly through nested statement nodes such as `IfStmt` and `WhileStmt`. Extracting explicit control flow requires building a CFG. This is why ClangIR is more convenient for the dataflow part of the analysis.

4 Conclusion

We have seen why ClangIR is a useful point for C++ static analysis: it keeps source-level semantic information while giving us an IR with SSA values and structured control flow. The next article, Building the LifetimeCheck Pass, starts from this motivation and follows the concrete state model used by the pass.

References

[1] Bjarne Stroustrup and Herb Sutter. C++ Core Guidelines. https://isocpp.org/guidelines, 2025. Lifetime Safety Profile: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-lifetime.

[2] Herb Sutter. Lifetime safety: Preventing common dangling. C++ Standards Committee Paper P1179R1, ISO/IEC JTC1/SC22/WG21, November 2019.