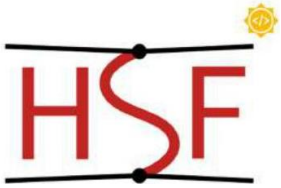
GSoC 2026 Project Proposal

CERN HSF
High Energy Physics Software Foundation

Implement CppInterOp API Exposing Memory Ownership and Thread Safety

Mentors

- Vipul Cariappa
  vipul.cariappa@gmail.com
- Vassil Vassilev
  vvasilev@cern.ch
- Aaron Jomy
  aaron.jomy@cern.ch

Personal Details

Name: Kerem Şahin
Phone: +90 535 748 5302
Email: keremsahin401@gmail.com
GitHub: github.com/keremsahn
Timezone: UTC+3 (Turkey Time)

PROJECT OVERVIEW

CppInterOp is a Clang-based library that provides an API to examine C/C++ code, enabling language bindings such as cppyy to interact with C++ code at runtime. While CppInterOp exposes function metadata, it currently lacks the ability to expose two critical properties of C/C++ code: memory ownership and thread safety characteristics of functions.

Language bindings have no way to determine if the function allocates memory on the heap, whether the caller is responsible for deallocating the memory, what allocation mechanism was used, or which function should be called to free it. Similarly, there is no information about whether a function is safe to call from multiple threads.

This project proposes to implement analysis using Clang's AST and LLVM IR to extract memory ownership and thread safety information from C++ code, and expose this information for language binding tools. The analysis starts from declaration-level checks (return type, const qualifiers, attributes) and continues to function body traversal for exposing further details.

BENEFITS TO THE COMMUNITY

C/C++ based libraries are commonly used in high-energy physics, scientific computations, and machine learning. Currently, language binding tools like cppyy allow calling C/C++ code from languages like Python and Julia, but they can not automate memory management or thread safety of C++ functions. As a result, users must manually handle low level details, which leads to memory leaks and data races and wastes resources of end-users.

This project aims to expose ownership and thread safety information so that language binding tools can handle memory management and multithreaded processes automatically, saving users from dealing with low-level details.

- **How will the HEP community benefit:** Large C++ frameworks like ROOT are heavily used from Python for scientific computations. After this project is completed, language binding tools will have enough information to save users from dealing with low-level details by handling them in the background.
- **How will open-source society benefit:** Any tool using CppInterOp can access ownership and thread safety data, and this data can be used to automate memory and thread management decisions.

IMPLEMENTATION PLAN

The implementation will consist of two main phases to expose information. Before these phases, corresponding enum classes or similar structures will be implemented to expose information in a proper and sustainable way. The design of these structures will be determined with mentors during the community bonding period.

- **Phase 1: IR Attribute Check**

Before starting traversal on Clang AST, we will check the LLVM attributes on functions, return values, and parameters. The main goal of this phase is avoiding costly AST traversal when LLVM attributes already provide the needed information.

There are three main attribute classes in LLVM; function attributes, parameter attributes, and global attributes. In our case, we will use function and parameter attributes to extract information about memory ownership and thread safety characteristics, since global attributes are not useful in this phase, and we can extract corresponding information from the Clang AST.

For example **nofree** attribute on a function means it does not directly or indirectly call a memory deallocation function, and with this information we can be sure that this function is not a deallocator.

Or **memory(none)** attribute on a function indicates that the function is pure and safe to call from multiple threads without any additional synchronization.

```
if (F->hasFnAttr(Attribute::ReadNone))
    return ThreadSafetyKind::Safe;
```

- **Phase 2: Traversal through Clang AST**

After getting the first wave of information from Phase 1, we will have some basic information about functions, but some cases will need to build upon that information, and for some other cases LLVM attributes do not give much information, so AST traversal is a must.

For example, **noalias** attribute of LLVM implies that a function's return value does not alias with any existing pointer, but can not tell us how the return value is allocated (malloc, new, new[]), and to know how to handle this chunk of memory we need the type of allocation function.

The traversal will be done via **RecursiveASTVisitor** of Clang. We will traverse through function body and search for related nodes for both memory and thread purposes.

To give a brief idea of how traversal will be done:

1) To extract information about memory allocation, we will inspect variable initializations and **ReturnStmt** node class, checking for **CXXNewExpr, CallExpr to malloc/calloc**, or address-of expressions.

2) Regarding thread safety, we will search for **unique_lock, scoped_lock, lock_guard or mutex.lock() in VarDecl and CXXMemberCallExpr** nodes considering scope. If a write operation, **BinaryOperator-UnaryOperator** etc, occurs outside the scope of these methods we will check LValue's scope and declaration.

POTENTIAL PROBLEMS AND SOLUTIONS

The implementation plan and the project itself consist of many edge cases due to the complexity of C/C++ memory management and the lack of built in ownership semantics, and I will introduce some of the potential problems and my approach to them.

Problem 1) Function Calls in Allocator-Deallocator Bodies

Analyzing every function that relates to a chunk of memory allocated/deallocated can cause performance problems and make the analysis almost impossible in large codebases.

To solve this problem, the implementation will adopt **Clang Static Analyzer**'s approach, **pointer escape analysis**. Instead of recursively analyzing every function call in the call chain, we track whether the pointer is directly handled (allocated/deallocated) in the current function body. If the pointer is passed to another function, it is treated as "escaped" and marked as Unknown, avoiding deep chain traversal.

Problem 2) Allocator-Deallocator Pairing

This case is the hardest part of the implementation and requires solutions beyond static analysis, dynamically calling functions and tracking pointers.

Static analysis can give us a possible list of deallocators for every allocator by checking scope proximity and return/parameter type match, but can not guarantee a definitive pairing. For example, in a container class **remove**, **clear**, and **pop** functions may all appear as possible deallocator candidates due to **free/delete** statements, while only one of them is actually responsible for the deallocation.

To determine allocator-deallocator pairs deterministically, we will first narrow down candidate deallocators statically, then verify the pairing dynamically by invoking the allocator, obtaining a pointer, and passing it to each candidate while tracking memory state to confirm correct deallocation, similar to how **AddressSanitizer** detects memory errors by tracking memory state at runtime.

Those two are the problems that I wanted to introduce and explain my approach since they require attention to have a highly optimized and deterministic analyzer. Solutions for all other cases I thought of remain within the bounds of static analysis.

TIMELINE

Duration	Tasks and Deliverables
Community Bonding May 1 – May 25	Finalize design with mentors, determine data structures and API pattern. **Deliverable:** Design agreed with mentors
Milestone 1 May 26 – Jun 15	Implement data structures for ownership and thread safety. Implement LLVM attribute checks for both topics. Unit tests for each attribute check. **Deliverable:** IR-based analysis working and tested
Milestone 2 Jun 16 – Jul 8	Declaration-level checks and body traversal for ownership detection. Unit tests for each ownership case. **Deliverable:** Ownership analysis working and tested
Milestone 3 Jul 9 – Aug 3	Scope scanning, type alignment, body analysis for deallocator detection. Unit tests for pairing logic. **Deliverable:** Deallocator pairing working and tested
Midterm Aug 4	**Deliverable:** Memory ownership analysis and deallocator pairing complete
Milestone 4 Aug 5 – Sep 4	Declaration checks and body walk for thread safety detection. Unit tests for each safety case. **Deliverable:** Thread safety analysis working and tested
Milestone 5 Sep 5 – Sep 21	Full integration testing on clang-repl and Cling. Documentation. **Deliverable:** All features verified end-to-end and documented
Milestone 6 Sep 22 – Oct 5	Demonstrate use of new APIs in cppyy as an exemplar. **Deliverable:** cppyy integration demo working
Final Oct 5	**Deliverable:** All features working, tested, and documented

RELATED WORK

PR #839 in CppInterOp: Adding API functions for detecting interpreter language and standard

- Interpreter language and standard detection support is added to C-kernels to be testable.
- Fixed CppInterOp #744 and xeus-cpp #441

ABOUT ME

I am currently pursuing my bachelor's degree in computer science and engineering at Boğaziçi University, Istanbul. I am new to the open-source community and started my journey with LLVM because of my interest in compiler infrastructure. I have experience in C++ and Python, and I have been studying compiler theory to build a solid foundation.

PERSONAL MOTIVATION

My biggest motivation is knowing that this work will be used by scientists in high-energy physics. Contributing to tools that are used by scientists to make the world a better place makes me excited. Furthermore, this project is my first step into the Compiler Research community, and I am excited to use my theoretical knowledge of LLVM and Clang internals through a hands-on experience.

POST GSoC

After implementing this project successfully, I would be glad to take a role in integrating the APIs implemented in this project into cppyy. The idea of building a compile-time tool — similar to AddressSanitizer or ThreadSanitizer but operating statically — that automatically enforces memory and thread safety for language bindings excites me as a long-term goal for the community. I am also interested in taking on any other broader role in the Compiler Research community and contributing to CppInterOp and cppyy's long-term development.