RESEARCH-ARTICLE

Type Deduction Analysis: Reconstructing Transparent Pointer Types in LLVM-IR

NICCOLÒ NICOLOSI, Politecnico di Milano, Milan, MI, Italy

GABRIELE MAGNANI, Politecnico di Milano, Milan, MI, Italy

EMILIO CORIGLIANO, Politecnico di Milano, Milan, MI, Italy

DAVIDE BAROFFIO, Politecnico di Milano, Milan, MI, Italy

FEDERICO REGHENZANI, Politecnico di Milano, Milan, MI, Italy

GIOVANNI AGOSTA, Politecnico di Milano, Milan, MI, Italy

Open Access Support provided by:

Politecnico di Milano

Published: 28 January 2026

Citation in BibTeX format

CC '26: 35th ACM SIGPLAN International Conference on Compiler Construction
January 31 - February 1, 2026
NSW, Sydney, Australia

Conference Sponsors:
SIGPLAN

https://doi.org/10.1145/3771775.3786268
ISBN: 9798400722745

Type Deduction Analysis: Reconstructing Transparent Pointer Types in LLVM-IR

Niccolò Nicolosi
niccolo.nicolosi@polimi.it
Politecnico di Milano
Italy

Gabriele Magnani
gabriele.magnani@polimi.it
Politecnico di Milano
Italy

Emilio Corigliano
emilio.corigliano@polimi.it
Politecnico di Milano
Italy

Davide Baroffio
davide.baroffio@polimi.it
Politecnico di Milano
Italy

Federico Reghenzani
federico.reghenzani@polimi.it
Politecnico di Milano
Italy

Giovanni Agosta
giovanni.agosta@polimi.it
Politecnico di Milano
Italy

Abstract

With version 17, LLVM finalized the transition to opaque pointer types, eliminating explicit pointee-type information from the Intermediate Representation (IR). Thus, starting from LLVM 17, each pointer type is represented in IR by the unique type `ptr`. Despite eliminating redundant pointer bitcasts and consequently reducing IR size and compile time, this change disrupts analyses that have reason to rely on pointee-type information, forcing existing compiler projects to depend on outdated LLVM versions. This information can in fact be insightful in fields like approximate computing, where the compiler can apply non-conservative optimizations, or in passes that require it to make analyses and transformations that do not impact the correctness of the program. To address this problem, we present a new Type Deduction Analysis pass that reconstructs transparent pointer types directly from opaque-pointer IR. Moreover, we illustrate two different case-studies on existing LLVM projects, namely TAFFO and ASPIS, that demonstrate the need for pointee-type information in LLVM compilers.

CCS Concepts: • **Software and its engineering → Compilers**; **Data types and structures**.

Keywords: Compilers, LLVM, Data-flow Analysis, Types

ACM Reference Format:
Niccolò Nicolosi, Gabriele Magnani, Emilio Corigliano, Davide Baroffio, Federico Reghenzani, and Giovanni Agosta. 2026. Type Deduction Analysis: Reconstructing Transparent Pointer Types in LLVM-IR. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC '26), January 31 – February 1, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3771775.3786268

CC '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2274-5/2026/01
https://doi.org/10.1145/3771775.3786268

1 Introduction

The shift of LLVM [15] from typed to opaque pointers, finalized with version 17, removes pointee-type information from the IR and replaces every pointer type with the unique opaque pointer type `ptr`[1]. This major change is motivated by the simplification induced by removing the need for pointer casts. The unique `ptr` type eliminates the need for pointer casts in the IR, reducing its size. Moreover, it simplifies LLVM passes, which have fewer instructions to consider. Both these consequences, in turn, reduce the overall compile time. However, the use of opaque pointers also has some drawbacks, since many passes made decisions by inspecting pointee types. Without this information, these passes either become overly conservative or must make deductions where possible, carrying their own metadata and consequently increasing their complexity. In extreme cases, LLVM-based projects that heavily rely on pointee-type information, such as TAFFO [7] and ASPIS [6], are forced to rely on outdated LLVM versions, since reworking their codebase to accommodate this fundamental change may require significant effort. The goal of this work is to address this gap, by providing an analysis pass to recover transparent pointer types starting from LLVM IR version 17. Thus, passes can reason on this information while maintaining, at the same time, most of the benefits of opaque pointers. We achieve this by presenting a new Type Deduction Analysis pass (TDA) that reconstructs a transparent type for each SSA value by observing its uses in the program and consistently merging the evidence they provide. TDA is interprocedural and robust to common coding patterns, including recursion and heterogeneous uses of a value that adhere to aliasing rules. The analysis can be performed at need, and downstream passes can inspect its results to know the transparent type of a value, similarly to how it was done in versions of LLVM previous to 15. In our experimental evaluation, TDA achieves the recovery of up to 100% of opaque pointer types, with an harmonic average of 84.72% in O0 and 87.99% in O3, thereby effectively eliminating the need for ad-hoc pointee-type inference in downstream LLVM passes.

[1] https://releases.llvm.org/17.0.1/docs/OpaquePointers.html

2 Related Work

The inference of pointee-types from opaque pointer types is a relatively new challenge. In fact, there are few examples in the literature that try to deal with this problem. To the best of our knowledge, only three prior works directly tackle the problem of reconstructing pointee-type information from opaque-pointer LLVM-IR. Zhou et al. [23] propose a methodology that recovers trusted type information from the debug information of the source language and then applies inference rules to propagate types across IR instructions. In this approach, being an alias analysis, each IR value is associated with a set of alias types that can contain several candidate types. The major caveat of the work is the requirement for debug information and no optimizations, which hinders the deployment of such a tool in a real-world scenario. Instead, William S. Moses et al. [17] briefly treat the problem of recovering LLVM pointee-types, implementing a first Type Analysis pass as a stepping stone to achieve their specific goals. The main theoretical contribution of this paper to the topic is the definition of a tree-like representation of types, which we will further expand in this paper. Similarly, Argyros [3] discusses the integration of a datalog-based static analysis tool called CClyzer-Soufflé [5] with opaque-pointer LLVM-IR. The tool analyzes the IR, implementing an internal type inference system to recover pointee-types. The work reports recovering on average the 76% of pointee-types of GNU Coreutils programs. The approach described is not directly integrated into the LLVM ecosystem, making it challenging for any LLVM pass to adopt it.

Prior to these specific works, there is a large body of literature that studies memory object propagation via points-to sets and alias relations [1, 11, 12, 14, 20], a field of study closely linked to pointee-type recovery. A notable example is the inclusion-based (Andersen-style) analysis [1], which computes, for each pointer, the set of allocations or abstract locations it may reference. Another relevant example is Type-Based Alias Analysis [11], which performs alias analysis based on types. Despite being related to the problem of pointee-type recovery, alias analyses focus on answering alias queries and do not provide information on types.

No previous works on pointer type recovery provide a complete formalization of such an analysis, nor formal guarantees on its results. It's important to highlight that our approach differs from the state of the art for two key reasons. First, it is the only solution that can be used at any stage of the middle-end optimization pipeline, not requiring source code information. Second, it is fully integrated within the LLVM pipeline and does not rely on external dependencies.

3 Type Deduction Analysis

This section describes our approach to TDA. The key challenge in this analysis is progressively refining opaque pointers into transparent types. To address this challenge, we first introduce a novel formalism for representing types as ordered trees. Then, we integrate this formalism into a forward interprocedural data-flow analysis that retrieves and merges type information about SSA values by examining how they are used throughout the code.

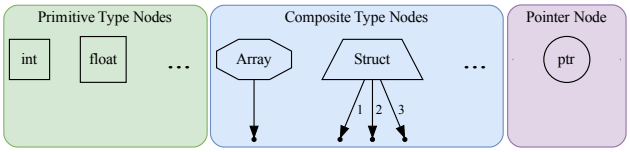

Figure 1. Nodes in the tree type representation.

3.1 Tree Type Representation

We represent a type as an ordered tree. Each node in the tree is a primitive data type (e.g. int, float), a composite type (e.g. any struct or array type), or a pointer node. As shown in Figure 1, nodes can be grouped in three classes: primitive type nodes, composite type nodes, and pointer nodes. Edges can only start from composite-type nodes or pointer nodes, entailing that primitive type nodes can only be leaf nodes. Each edge from composite type node t to a generic node n, implies that the type represented by the sub-tree with n as root is contained in type t. Each edge from pointer node p to a generic node n implies that the type represented by the sub-tree with p as root, is a pointer to the type represented by the sub-tree with n as root. A pointer node can have at most one exiting edge.

Definition 1 (Opaque pointer)**.** A type t is an opaque pointer if and only if its tree representation is a single leaf pointer node.

Definition 2 (Transparent type)**.** We define a type t as transparent if and only if its tree representation does not contain any opaque pointer.

The children of each composite node are ordered with respect to the type declaration. Figure 2 presents notable examples of this tree representation. Following the logic applied to nodes, we can group types in the same three classes, based on the class of their root node, obtaining primitive types, composite types, and pointer types.

In case of self-referential composite types, the representation must be refined not to grow indefinitely. It is shown in Figure 3 how the self-referential composite type `RecStruct` would not be representable by a finite tree, and how the problem can be addressed by stopping the expansion of the contained type `RecStruct`. In general, recursion can be easily detected by checking if the path from the root to a node n contains another instance of node n itself.

3.2 Type Merging

To proceed with the definition of type deduction analysis, we need to introduce the concepts of *type compatibility* and *type*

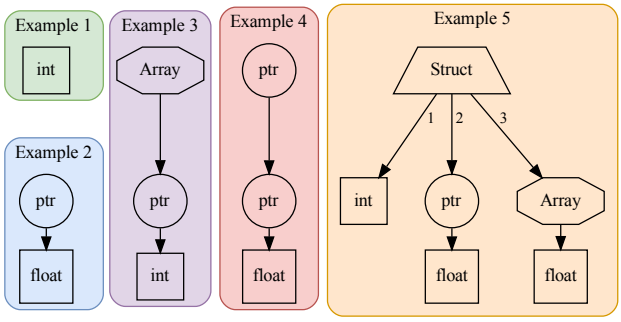

Figure 2. Representation of types `int`, `float*`, `int*[]`, `float**` and `Struct{ int, float*, float[] }`.

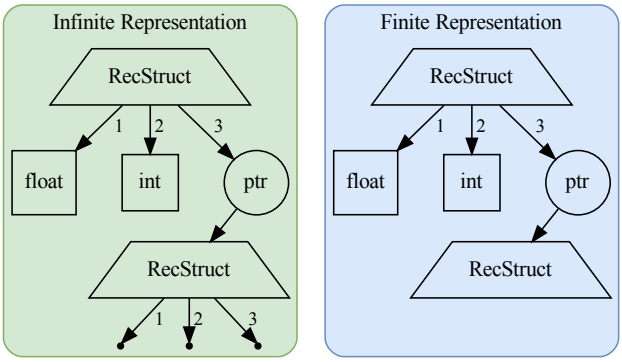

Figure 3. Infinite (left) and finite (right) representation of the self-referencing composite type `RecStruct{ float, int, RecStruct* }`.

merging. First of all, we define the conditions under which two types t_1 and t_2 are compatible in algorithm 1. Given t_1 and t_2, represented by their roots pair (r_1, r_2), we check if they are in the same class of nodes (primitive, composite, or pointers). If they are not, t_1 and t_2 are not compatible. In case they are, there are three possibilities. If both are primitive types, t_1 and t_2 are compatible if and only if $r_1 = r_2$. If r_1 and r_2 are composite types, t_1 and t_2 are compatible if and only if they have the same number of outward edges and, for each pair of child nodes (c_1, c_2) such that the index of c_1 is equal to the index of c_2, c_1 and c_2 are compatible. If r_1 and r_2 are pointer nodes, we check whether neither is opaque: in case this condition is respected, t_1 and t_2 are compatible if and only if their children c_1 and c_2 are compatible. Otherwise, if at least one among t_1 and t_2 is opaque, we say that t_1 and t_2 are compatible, and we call (r_1, r_2) a *mergeable pair*.

Given two roots r_1 and r_2 and their corresponding types t_1 and t_2 that are compatible, their merge is performed as shown in algorithm 2. The merge procedure takes two roots r_1 and r_2 as arguments and returns a merged root r_3 corresponding to a merged type t_3. If the roots are primitive nodes, t_1 and t_2 are the same type as a consequence of the compatibility definition, thus the resulting type is $t_3 = t_2 = t_1$.

If t_1 and t_2 are pointer types and their roots pair (r_1, r_2) is a mergeable pair, there are two possible outcomes for the result type t_3. If both t_1 and t_2 are opaque, then t_3 is an opaque pointer. Otherwise, t_3 is the non-opaque pointer type among t_1 and t_2. In the case that t_1 and t_2 are pointer types, but their roots pair (r_1, r_2) is not a mergeable pair, the result type t_3 is a pointer to the merge of the roots' children pair (c_1, c_2). Similarly, if t_1 and t_2 are composite types, their roots r_1 and r_2 are equal as a consequence of the compatibility definition, and the result t_3 has root $r_3 = r_2 = r_1$ and children equal to the merge of each children pair (c_1, c_2) such that the index of c_1 is equal to the index of c_2.

From now on, given two types t_1 and t_2, we will use the notation $t_1 \simeq t_2$ to denote they are compatible and the notation $t_1 \cup t_2$ to represent their merge.

3.3 Aliasing Assumptions

The fundamental assumption we make for TDA is that the program P on which we perform TDA is a well-formed program that follows specific aliasing rules. In particular, each pointer type t in program P can be aliased as t itself or as a *probe type* x. An example is C++, for which the probe types are `char*`, `unsigned char*`, and `std::byte*` as specified in [13], all equivalent to `i8*` when lowered. This extends the previous definition of compatibility and merge as follows. Let t_1, t_2 be non-opaque pointer types:

$$t_1 = x \implies t_1 \simeq t_2 \wedge t_1 \cup t_2 = t_2$$
$$t_2 = x \implies t_1 \simeq t_2 \wedge t_1 \cup t_2 = t_1$$

This implies that, given the set V of the SSA values of P and the set A_v of the alias types of v in P:

$$a_1 \simeq a_2 \qquad \forall\, a_1, a_2 \in A_v$$

3.4 The TDA Lattice

Since TDA is a data-flow analysis [18], in order to have guarantees on its convergence, we need to define its support lattice. Given program P respecting the assumptions defined in Section 3.3, let V be the set of SSA values of P and T_v be the set containing all possible alias types for $v \in V$ and a sentinel value $\bot_v$. At first, we will demonstrate that T_v is a lattice. Then we will introduce the concept of type mapping, and we will define the support of TDA as the set of all possible type mappings, demonstrating that it is, in turn, a lattice.

Definition 3 (Compatibility and merge for $\bot_v$)**.** We extend for $\bot_v$ the definitions of compatibility and merge as follows:

$$\bot_v \simeq t \qquad \forall\, t \in T_v \tag{1}$$
$$\bot_v \cup t = t \qquad \forall\, t \in T_v \tag{2}$$

By construction of T_v, we know that:

$$t_1 \simeq t_2 \qquad \forall\, t_1, t_2 \in T_v \tag{3}$$
$$t_1 \cup t_2 = t_3 \in T_v \qquad \forall\, t_1, t_2 \in T_v \tag{4}$$

Algorithm 1 Pseudo code of the type compatibility algorithm. Function class(x) returns the class of node x: Primitive, Composite or Pointer. Function deg(x) returns the number of children of node x. Function idx(y) returns the index of node y. Function opaque(x) returns true if x is an opaque pointer, false otherwise. Function pointee(x) returns the child of non-opaque pointer node x.

```
1:  function COMPATIBLE(r1, r2)
2:      if class(r1) ≠ class(r2) then
3:          return false
4:      end if
5:      if class(r1) = Primitive then
6:          return r1 = r2
7:      else if class(r1) = Composite then
8:          if deg(r1) ≠ deg(r2) then
9:              return false
10:         end if
11:         for all c1, c2 | idx(c1) = idx(c2) do
12:             if ¬COMPATIBLE(c1, c2) then
13:                 return false
14:             end if
15:         end for
16:         return true
17:     else if class(r1) = Pointer then
18:         if opaque(r1) ∨ opaque(r2) then
19:             return true
20:         else
21:             c1 ← pointee(r1)
22:             c2 ← pointee(r2)
23:             return COMPATIBLE(c1, c2)
24:         end if
25:     end if
26: end function
```

Algorithm 2 Pseudo code of the type merging algorithm. Function class(x) returns the class of node x: Primitive, Composite or Pointer. Function idx(y) returns the index of node y. Function opaque(x) returns true if x is an opaque pointer, false otherwise. Function pointee(x) returns the child of non-opaque pointer node x. Function pointerTo(x) returns a pointer to type x. Function setChild(x, y, z) sets y as a child to x with index z.

```
1:  function MERGE(r1, r2)
2:      if class(r1) = Primitive then
3:          return r1
4:      else if class(r1) = Pointer then
5:          if opaque(r1) ∧ opaque(r2) then
6:              return r1
7:          else if opaque(r1) then
8:              return r2
9:          else if opaque(r2) then
10:             return r1
11:         else
12:             c1 ← pointee(r1)
13:             c2 ← pointee(r2)
14:             r3 ← pointerTo(MERGE(c1, c2))
15:             return r3
16:         end if
17:     else if class(r1) = Composite then
18:         r3 ← r1
19:         for all c1, c2 | idx(c1) = idx(c2) do
20:             c3 ←MERGE(c1, c2)
21:             setChild(r3, c3, idx(c1))
22:         end for
23:         return r3
24:     end if
25: end function
```

Definition 4 (Partial ordering of T_v)**.** So, we define a partial ordering of T_v as follows:

$$t_1 \leq t_2 \iff t_1 \cup t_2 = t_2 \qquad \forall\, t_1, t_2 \in T_v \tag{5}$$

and we write $t_1 < t_2$ for $t_1 \leq t_2 \wedge t_2 \not\leq t_1$.

Lemma 1 (Merge ordering)**.** From the definition of merge given in Section 3.2 and from (5), it follows that:

$$t_1 \cup t_2 = t_3 \implies t_1 \leq t_3 \wedge t_2 \leq t_3 \qquad \forall\, t_1, t_2, t_3 \in T_v \tag{6}$$

Lemma 2 (T_v is a lattice)**.** T_v is a lattice with respect to (5).

Proof 1. (Proof of lemma 2)**.** In order to demonstrate lemma 2, we need to prove that the extremes $\top_v$ and $\bot_v$ of T_v are unique.

Uniqueness of $\top_v$: we demonstrate with a proof by contradiction that there must exist a unique type $\top_v \in T_v$ such that:

$$t \leq \top_v \qquad \forall\, t \in T_v$$

Let different types $t_1, t_2 \in T_v$ that are both greater than all other $t \in T_v \setminus \{t_1, t_2\}$. In formulas:

$$t < t_1 \qquad \forall\, t \in T_v \setminus \{t_1, t_2\}$$
$$t < t_2 \qquad \forall\, t \in T_v \setminus \{t_1, t_2\}$$
$$t_1 \not< t_2 \ \wedge\ t_2 \not< t_1 \ \wedge\ t_2 \neq t_1$$

because $t_1 \simeq t_2$ for (3), they can be merged and by (4) and (6):

$$t_1 \cup t_2 = t_3 \in T_v \mid t_1 \leq t_3 \wedge t_2 \leq t_3$$

but knowing that $t_1 \neq t_2$ by construction, we deduce that $t_1 < t_3 \wedge t_2 < t_3$, contradicting the original hypothesis. Therefore, $\top_v$ must be unique.

Uniqueness of $\bot_v$: we demonstrate with a proof by contradiction that there must exist a unique type $\bot_v \in T_v$ such that:

$$\bot_v \leq t \qquad \forall\, t \in T_v$$

Let different types $t_1, t_2 \in T_v$ that are both smaller than all other $t \in T_v \setminus \{t_1, t_2\}$. In formulas:

$$t_1 < t \qquad \forall\, t \in T_v \setminus \{t_1, t_2\}$$

$$t_2 < t \qquad \forall\, t \in T_v \setminus \{t_1, t_2\}$$
$$t_1 \not< t_2 \ \wedge\ t_2 \not< t_1 \ \wedge\ t_2 \neq t_1$$

because $t_1 \simeq t_2$ for (3), they can be merged and by (2):

$$t_1 \cup t_2 = t_1 \wedge t_2 \cup t_1 = t_2$$

but knowing that $t_1 \neq t_2$ by construction, we are contradicting the original hypothesis. Therefore, $\bot_v$ must be unique.

Because $\top_v$ and $\bot_v$ are unique, it follows that T_v is a lattice with respect to (5).

□

Definition 5 (Set of type mappings $\mathcal{M}$)**.** Subsequently, we define the set of all possible type mappings for program P as:

$$\mathcal{M} = \left\{ \mu : V \to \bigcup_{v \in V} T_v \;\middle|\; \mu(v) \in T_v \right\}$$

Definition 6 (Merge of type mappings)**.** Then, we extend the merge definition for type mappings:

$$\mu_1 \cup \mu_2 = \mu_3 \iff \mu_3(v) = \mu_1(v) \cup \mu_2(v) \quad \forall\, \mu_1, \mu_2 \in \mathcal{M},\ \forall\, v \in V \tag{7}$$

Definition 7 (Partial ordering of $\mathcal{M}$)**.** We define a partial ordering of $\mathcal{M}$ by the following:

$$\mu_1 \leq \mu_2 \iff \mu_1 \cup \mu_2 = \mu_2 \qquad \forall\, \mu_1, \mu_2 \in \mathcal{M} \tag{8}$$

Definition 8 (Extremes of $\mathcal{M}$)**.** Finally, we define the extremes of $\mathcal{M}$:

$$\bot_{\mathcal{M}} \in \mathcal{M} \mid \bot_{\mathcal{M}}(v) = \bot_v\ \forall\, v \in V \tag{9}$$
$$\top_{\mathcal{M}} \in \mathcal{M} \mid \top_{\mathcal{M}}(v) = \top_v\ \forall\, v \in V \tag{10}$$

Corollary 1 ($\mathcal{M}$ is a lattice)**.** As a corollary of lemma 2, $\mathcal{M}$ is a lattice with respect to (8).

Proof 2. (Proof of corollary 1)**.** It is trivial to derive from (7), (8), (9) and (10) that:

$$\bot_{\mathcal{M}} \leq \mu \qquad \forall\, \mu \in \mathcal{M} \tag{11}$$
$$\mu \leq \top_{\mathcal{M}} \qquad \forall\, \mu \in \mathcal{M} \tag{12}$$

As a direct consequence of lemma 2 and by (9) and (10), the extremes $\bot_{\mathcal{M}}$ and $\top_{\mathcal{M}}$ of $\mathcal{M}$ are unique. Therefore, $\mathcal{M}$ is a lattice with respect to (8). □

3.5 Intraprocedural TDA

Unlike the most common data-flow analyses, which are performed on the control-flow graph (CFG), TDA is performed on an enriched version of the def-use graph (DUG) of a program, to which we will refer as enriched def-use graph (eDUG). The eDUG of program P is built starting from the DUG of program P by adding a start node s and an end node e. Node s is connected with an edge from s to each node in the DUG that does not have any entering edge. Analogously, node e is connected to each node n in the DUG that does not have any exiting edge, with an edge from n to e. Then, node e is connected to node s with an edge from e to s, resulting in a cyclic graph in which data can flow from any node n to any other node m. An example is illustrated in listings 1 and 2, showing a simple C++ program and its LLVM-IR respectively, and in Figure 4, depicting the resulting eDUG.

```
int a = 0;
int* b = (int*) malloc(sizeof(int));
a + *b;
```

Listing 1. Example C++ program.

```
%a = alloca i32
%b = alloca ptr
store i32 0, ptr %a
%c = call ptr @malloc(i64 4)
store ptr %c, ptr %b
%0 = load i32, ptr %a
%1 = load ptr, ptr %b
%2 = load i32, ptr %1
%add = add i32 %0, %2
```

Listing 2. LLVM-IR of the program in listing 1.

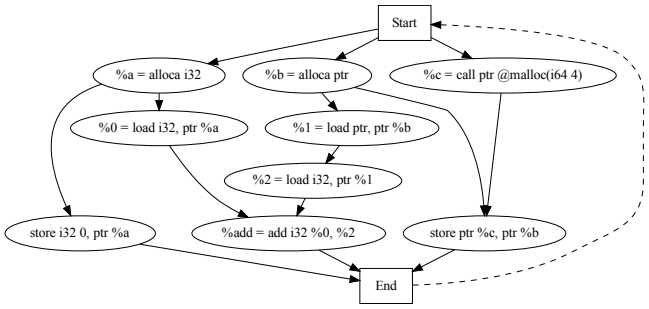

Figure 4. Enriched def-use graph of the program LLVM-IR in listing 2. The additional start and end nodes are represented as rectangles. The return edge from end to start is represented with a dashed line.

Definition 9 (Interpretation function γ)**.** Let N be the set of nodes in the eDUG of program P, we define the interpretation function γ as:

$$\gamma : N \times \mathcal{M} \times V \to T_v$$

where $\gamma(n, \mu, v)$ represents the type of value v as interpreted by node n starting from mapping μ. For example, in the context of LLVM-IR, consider node n for instruction:

```
%0 = load ptr, ptr %a
```

and mapping μ such that:

$$\mu(\texttt{\%a}) = \texttt{i32**} \wedge \mu(v) = \bot_v\ \forall\, v \in V \setminus \{\texttt{\%a}\}$$

The interpretation function $\gamma(n, \mu, v)$ will be such that:

$$\gamma(n, \mu, \texttt{\%0}) = \texttt{i32*}$$
$$\gamma(n, \mu, \texttt{\%a}) = \texttt{i32**}$$
$$\gamma(n, \mu, v) = \bot_v \qquad \forall\, v \in V \setminus \{\texttt{\%a}, \texttt{\%0}\}$$

Definition 10 (Gen function Γ)**.** We define the gen function:

$$\Gamma : N \times \mathcal{M} \to \mathcal{M} \mid \Gamma(n, \mu_1) = \mu_2 \wedge \mu_2(v) = \gamma(n, \mu_1, v)$$
$$\forall\, v \in V$$

Definition 11 (TDA flow equations)**.** We define the flow equations of TDA as follows. For each node n of the eDUG, let $\mathcal{P}_n$ be the set of predecessors of n:

$$\mathrm{IN}[n] = \bigcup_{p \in \mathcal{P}_n} \mathrm{OUT}[p]$$
$$\mathrm{OUT}[n] = \Gamma(n, \mathrm{IN}[n]) \cup \mathrm{IN}[n]$$

Intuitively, $\mathrm{IN}[n]$ is the merge of the mappings that node n takes as inputs from its predecessors, which are used to produce the output mapping $\mathrm{OUT}[n]$ by means of the interpretation function γ. Note that there is no explicit $\mathrm{KILL}[n]$ because mappings are automatically updated by merge.

Thus, the final result of TDA is a mapping $\mu^* \in \mathcal{M}$, and because $\mathcal{M}$ is a lattice, it is guaranteed that $\mu^* = \top_{\mathcal{M}}$.

3.6 Interprocedural TDA

For interprocedural TDA we need to further extend the eDUG to consider procedure calls. We will refer to this graph as interprocedural extended def-use graph (ieDUG).

Definition 12 (Merge node)**.** First, we introduce the definition of a merge node. We define a merge node for the set of values $S \subseteq V$, a node m_S such that:

$$\gamma(m_S, \mu, v) = \bigcup_{s \in S} \mu(s) \qquad \forall\, v \in S$$
$$\gamma(m_S, \mu, v) = \bot_v \qquad \forall\, v \notin S$$

and we represent it in the graph with a node containing the vararg pseudo-instruction:

```
merge v1, v2, ...
```

We build the ieDUG of program P in four steps. Firstly, given the set F of procedures in P, for each procedure $f \in F$ we generate its def-use graph DUG_f. We decorate DUG_f by adding a node d for the declaration of f, and we connect d to each node n of DUG_f that does not have any entering edge, with an edge from d to n.

Secondly, let N be the set of all nodes in DUG_f for all $f \in F$, let $C \subset N$ be the set of call site nodes in program P, let A_c be the set of actual arguments of call $c \in C$ and A_f be the set of formal arguments of procedure $f \in F$ called by c, let d_f be the definition node of f. We define function $\alpha : A_c \to A_f$ be such that $\alpha(a_c) = a_f$ where a_f is the corresponding formal argument of f with respect to a_c. For each call node $c \in C$, for each $a_c \in A_c$ we add a merge node m_{a_c} for values $\{a_c, \alpha(a_c)\}$, connecting it to node c with an edge from c to m_{a_c}, and to d_f with an edge from m_{a_c} to d_f.

Thirdly, let $R_f \subset N$ be the set of nodes for return instructions of procedure f and v_r be the return value of $r \in R_f$. For each call node $c \in C$ to $f \in F$, we add a merge node m for values $\{c\} \cup \{v_r \mid r \in R_f\}$, connecting m to each node $r \in R_f$ with an edge from r to m, and to c with an edge from m to c. Assuming there is no call site to the entry point f_e of program P, the second and third steps of this process will not be performed for f_e. Thus, we can finally extend the resulting graph using the algorithm for building the eDUG, described in Section 3.5.

To make an example, consider the simple program in listing 3. The program has two calls to function fun, which has two arguments and two different return instructions. Figure 5 shows the resulting ieDUG, in which the additional merge nodes make it possible for TDA to merge the types of the function's formal arguments with the ones of the actual arguments of each call, and to merge the type of the call itself with the return type of the function.

Using the flow equations from definition 11, we can perform interprocedural TDA on ieDUG of program P, obtaining the same convergence guarantees of intraprocedural TDA.

```
define i32 @fun(i32 %fun_a, i32 %fun_b) {
entry:
  ...
if.then:
  ...
  ret i32 %fun_0
if.end:
  ...
  ret i32 %fun_1
}
...
%call1 = call i32 @fun(i32 %a, i32 %b)
%call2 = call i32 @fun(i32 %b, i32 %c)
...
```

Listing 3. Example program LLVM-IR with procedure calls. The dots are used to represent omitted parts of the program.

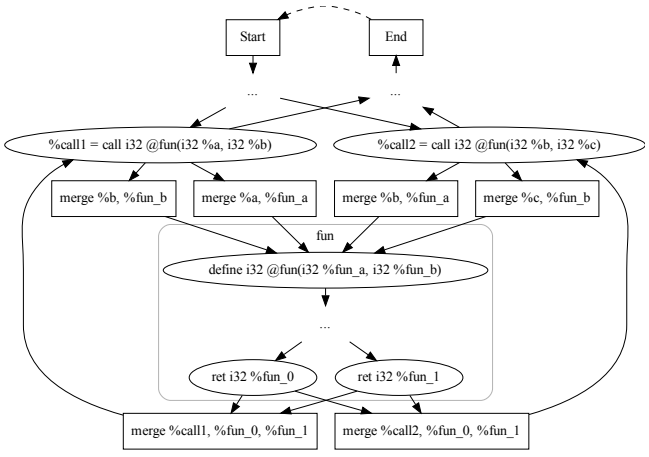

Figure 5. Interprocedural enriched def-use graph of the program in listing 3. The additional merge nodes, the start node, and the end node are represented as rectangles. The return edge from end to start is represented with a dashed line. The def-use graph of function fun and its declaration node are represented in the homonymous cluster.

3.7 The TDA LLVM Pass

The LLVM implementation of TDA[2] follows the formal setting illustrated in the previous sections and realizes TDA as an LLVM module analysis pass. The implementation builds the ieDUG of the module, where each node is mapped to a specific LLVM value — instructions, functions, global variables, etc… — and defines a concrete interpretation function for each node.

In order to proceed, it is important to note that even though the majority of LLVM values produces an SSA value, it is not always the case. For example, an `alloca` instruction produces an SSA value, while a `store` instruction does not.

Definition 13 (Helper function σ)**.** Hence, we define a helper function σ to retrieve the SSA value of the LLVM value associated to a node, if any. Let V^+ be the set of SSA values of the LLVM module with an additional sentinel value `null`, and N be the set of nodes of its ieDUG. We define the helper function σ as:

$$\sigma : N \to V^+$$

where $\sigma(n) = \texttt{null}$ in case n is the node of an LLVM value that does not produce an SSA value.

Definition 14 (IR type function τ)**.** Additionally, let V be the set of SSA values of the LLVM module and T_v be the set of all the possible alias types of $v \in V$, we define IR type function τ as:

$$\tau : V \to T_v$$

where $\tau(v)$ is the type provided by the LLVM IR for value v.

Before defining the interpretation functions for LLVM, we need to introduce the following type operators.

Definition 15 (Pointer-to operator $\uparrow$)**.** Given type t we define $t^{\uparrow}$ as a pointer to t.

Definition 16 (Dereference operator $\downarrow$)**.** Given type t and value $v \in V$, let r be the root node of t. We define $t^{\downarrow}$ as the sub-tree with the child of r as root, in case t is a non-opaque pointer, or $\bot_v$ otherwise.

Finally, we need to note that LLVM's `getelementptr` instruction produces an SSA value whose dereferenced type is contained in the type of its pointer operand. For this reason we need two further dedicated operators.

Definition 17 (Get-element function ϵ)**.** Let $n \in N$ be the node of a `getelementptr` whose indices are $i_0, \ldots, i_k$, and let $\sigma(n) = v$. We define the get-element function ϵ such that $\epsilon(n, t_1) = t_2$ where t_2 is the sub-tree of t_1 indexed by $i_0, \ldots, i_k$ if it can be determined at compile time, or $\bot_v$ otherwise.

Definition 18 (Merge-element function η)**.** Let $n \in N$ be the node of a `getelementptr` whose pointer operand is value $p \in V$ of type t_p, and whose indices are $i_0, \ldots, i_k$. Let $\sigma(n) = v$. We define the merge-element function η such that $\eta(n, t_1) = t_2$ where, if $\epsilon(n, t_p) \neq \bot_v \wedge \epsilon(n, t_p) \simeq t_1$, then t_2 is equal to t_p except for its sub-tree indexed by $i_0, \ldots, i_k$ which is replaced by $\epsilon(n, t_p) \cup t_1$; otherwise t_2 is $\bot_p$.

The interpretation functions defined for LLVM are summarized in Table 1. Each node $n \in N$ not reported in the table has its interpretation function defined as follows for each $v \in V$:

$$\sigma(n) \neq \texttt{null} \implies \gamma(n, \mu, v) = \begin{cases} \tau(v) & v = \sigma(n) \\ \bot_v & \text{otherwise} \end{cases}$$

$$\sigma(n) = \texttt{null} \implies \gamma(n, \mu, v) = \bot_v$$

Table 1. LLVM interpretation functions

Node n	Interpretation function
`%v = alloca t`	$\gamma(n, \mu, \%v) = \texttt{t}^{\uparrow}$ $\gamma(n, \mu, \ v) = \bot_v \quad \forall\, v \in V \setminus \{\%v\}$
`%v = load t, ptr %p`	$\gamma(n, \mu, \%v) = \texttt{t} \cup \mu(\%p)^{\downarrow}$ $\gamma(n, \mu, \%p) = \texttt{t}^{\uparrow} \cup \mu(\%v)^{\uparrow}$ $\gamma(n, \mu, \ v) = \bot_v \quad \forall\, v \in V \setminus \{\%v, \%p\}$
`store t %v, ptr %p`	$\gamma(n, \mu, \%v) = \texttt{t} \cup \mu(\%p)^{\downarrow}$ $\gamma(n, \mu, \%p) = \texttt{t}^{\uparrow} \cup \mu(\%v)^{\uparrow}$ $\gamma(n, \mu, \ v) = \bot_v \quad \forall\, v \in V \setminus \{\%v, \%p\}$
`%v = getelementptr t, ptr %p, i64` i_0`, …, i64` i_k	$\gamma(n, \mu, \%v) = \epsilon(n, t^{\uparrow})^{\uparrow} \cup \epsilon(n, \mu(\%p))^{\uparrow}$ $\gamma(n, \mu, \%p) = \eta(n, \mu(\%v)^{\downarrow})$ $\gamma(n, \mu, \ v) = \bot_v \quad \forall\, v \in V \setminus \{\%v, \%p\}$
`@g = global t %v`	$\gamma(n, \mu, @g) = \texttt{t}^{\uparrow} \cup \mu(\%v)^{\uparrow}$ $\gamma(n, \mu, \%v) = \texttt{t} \cup \mu(@g)^{\downarrow}$ $\gamma(n, \mu, \ v) = \bot_v \quad \forall\, v \in V \setminus \{@g, \%v\}$

The pass iterates the flow equations on the ieDUG until a fixed point is reached, leading to the result type mapping μ^*.

3.8 Tackling Non-idealities in TDA

Real-world scenarios present two major non-idealities having an effect on μ^* that the TDA LLVM pass needs to consider.

Firstly, an LLVM module does not always coincide with a whole program. In fact, a module can rely on external functions for which only the declaration is available. As a consequence, any external function f_{ex} in the module has an empty DUG, resulting in the declaration node of f_{ex} in the ieDUG having a single exiting edge, connected to the end node. Moreover, the type of any value that is not used throughout the module code — possibly being used outside of the analyzed module or not used completely in case of dead code — cannot be deduced by the analysis. Even when a value is used in the module, it is not always possible for TDA to deduce its type. A simple example is a value whose real type is `struct{`t_1`,`t_2`}*`, only used by a `getelementptr`

[2]Source code available at: https://doi.org/10.5281/zenodo.17570563

that indexes t_1. In this case, TDA can make deductions on the first field t_1, but it has no way to make deductions on the second field t_2, which can remain not transparent in case it is a pointer type or contains any pointer type.

Secondly, LLVM does not guarantee that the aliasing assumptions illustrated in 3.3 are respected. Therefore, any violation of these assumptions in the module can lead to the pass attempting to merge incompatible types for a value $v \in V$, which can be resolved as $\tau(v)$. This becomes even more relevant when considering code subject to the optimizations performed by other passes. In fact, optimizations often transform the code in such ways that it becomes difficult to make deductions on types. For example, they introduce different forms of type punning and canonicalize `getelementptr` instructions by removing leading and trailing zero indices. Also clang itself optimizes the IR at the front-end level often violating aliasing rules and sometimes removing some struct types completely. Fortunately, it also provides some information on the original types accessed by `load` and `store` instructions in the form of TBAA metadata. Thus, we use the additional information of TBAA metadata for the deduction process, in the interpretation function of these instructions. Moreover, to address the problem of intentional type punning from developers, we consider a set of different alias types for the same value, when needed.

Another thing to consider are function pointers and indirect calls. Despite function pointers themselves not constituting a problem for TDA, the ieDUG cannot be fully generated for their calls, as it is not always possible to know at compile time which function will be executed at runtime. Therefore they are treated as another minor non-ideality.

These non-idealities can degrade the results of the TDA pass, in practice possibly implying that — depending on the module considered — given the set V of the SSA values of the module, $\exists\, v \in V$ such that $\mu^*(v)$ is not transparent.

4 Case Studies

To demonstrate the need for pointee-type information in LLVM, we examine two representative LLVM frameworks whose core analyses and transformations depend on it. The first, TAFFO, is a precision-tuning framework that converts selected floating-point computations to fixed-point. The second, ASPIS, is a compiler-level soft-error hardening suite that compares structured data under redundant execution. In both, opaque pointers represent a substantial obstacle that can be surpassed by integrating the TDA LLVM pass.

4.1 TAFFO

TAFFO [7] (Tuning Assistant for Floating-point to Fixed-point Optimization) is a compiler framework for approximate computing — more precisely, for precision-tuning — that aims to reduce the execution time of programs by converting selected floating-point computations into fixed-point computations, while maintaining a good level of accuracy in the results. TAFFO is composed of a pipeline of LLVM passes. At first, the initializer pass parses program annotations that state assumptions on specific variables and attaches metadata to the IR values that will be subject to precision tuning. Additionally, for each call site, the pass clones the called function so that different precision choices can be explored independently. Then, the value-range analysis pass (VRA) infers the ranges of relevant IR values starting from their metadata and using interval arithmetic. Subsequently, the data type allocation pass (DTA) uses the inferred ranges to choose a suitable fixed-point representation for each convertible value, merging similar formats across computations to avoid excessive heterogeneity. Lastly, the conversion pass transforms the IR according to the decisions of the previous pass. The whole process requires information on types and numeric representations. Specifically, to identify values whose type contains floating-points and transform them into values whose type is equivalent but instead contains fixed-points. Indeed, the initializer needs to identify the floating-point values that are candidates for conversion; the VRA and DTA need to build and transform values' metadata to represent their ranges and their target fixed-point formats; the conversion pass needs to generate values and instructions that respect these types. Often, many of these types happen to be pointers or composite types containing pointers, reason why TAFFO has depended so far on LLVM version 15 or older.

By performing TDA as the first step of the pipeline, TAFFO can query its results instead of relying on the IR. Thanks to the integration with TDA, we managed to update TAFFO to support LLVM versions greater than 15, restoring its ability to reason about types and successfully enabling a series of new research opportunities that were being detained by the outdated LLVM support.

4.2 ASPIS

ASPIS [6] (Automatic Software-based Protection and Integrity Suite) is a suite of LLVM-based passes for Software Implemented Hardware Fault Tolerance (SIHFT) against so-called Single-Event Upsets (SEUs). These events, commonly modeled as single-bit flips, can produce errors in the program, which may lead to catastrophic consequences in safety-critical systems. ASPIS provides a cost-effective solution to protect against both families of errors, implementing IR-level transformation passes for data-flow and control-flow hardening. For data-flow protection, ASPIS implements the Error Detection by Duplicated Instructions (EDDI) [19] and Recursive EDDI (REDDI) [10] algorithms at the IR Level. EDDI transforms the original code, duplicating program data and code, and interleaving original and duplicated instructions and storage. REDDI, instead, generalizes EDDI, enabling selective hardening of critical resources along with all their data dependencies. Listing 4 provides an example of ASPIS-hardened code that performs the addition of two numbers.

```
bb_1:
  %1 = load i32, ptr %a, align 4
  %1_dup = load i32, ptr %a_dup, align 4
  %2 = add nsw i32 %1, 10
  %2_dup = add nsw i32 %1_dup, 10
  %3 = icmp eq i32 %2, %2_dup
  br i1 %3, label %bb_2, label %bb_err
bb_2:
  store i32 %2, ptr %b, align 4
  store i32 %2_dup, ptr %b_dup, align 4
```

Listing 4. LLVM-IR snippet of an ASPIS-hardened program.

The critical part of this mechanism is the comparison of the two copies of the data. ASPIS inserts consistency checks before `store`, `branch`, and `call` instructions. In the example, the `icmp` and `br` instructions are inserted before the `store` to check the consistency of the stored data. ASPIS also creates a branch to `bb_err`, which calls a user-defined error handling routine to execute in the case of a runtime data mismatch. Listing 4 shows that the comparison mechanism is trivial for simple types, producing a simple compare instruction for checking consistency and a conditional jump. For composite types like arrays and structs, the comparison has to be performed over each element or field, and the comparisons have to be checked by one or more `and` instructions:

```
%1 = icmp eq i32 %field1, %field1_dup
%2 = icmp eq i32 %field2, %field2_dup
%3 = and i1 %1, %2
br i1 %3, label %bb_2, label %bb_err
```

However, due to the absence of typed pointers, EDDI cannot perform the comparison, since two pointers to redundant memory regions always differ, and their content cannot be compared because we do not know the data structure layout. Moreover, REDDI introduces the mechanism of temporary argument duplication to limit the hardening of the program only to the critical resources. This duplicates the arguments of a function call not yet hardened, immediately prior to the invocation. Thus, the call can be performed on the hardened function without requiring recursive hardening of its parameters. Specifically, for each non-hardened argument, a temporary duplicate is generated by allocating dedicated memory and performing a `memcpy` over the size of the variable. Correct execution of this procedure requires precise knowledge of both the type and the size of the argument.

TDA significantly enhances the logic of EDDI comparisons and REDDI temporary argument duplication, so that even in cases where the LLVM values are pointers, knowing the number of pointer indirections and the layout of the pointed memory allows them to be compared and copied nonetheless. Moreover, TDA tackles the limitations of previous works, since it also accepts optimized IR as input.

5 Experimental Evaluation

TDA was tested on three different benchmark suites already used to validate TAFFO and ASPIS, namely PolyBench, AxBench, and ABSURD.

PolyBench [22], even though originally proposed for testing polyhedral analyses in compilers, is a benchmark suite composed of 30 different numerical computations, extracted from a vast range of different domains, ranging from linear algebra to image processing, physics simulation, and more. Nowadays, it has been widely accepted as a standard benchmark suite in a variety of computational domains. Moreover, PolyBench was also adopted to test the performance of TAFFO in previous works[8, 16].

AxBench [21] is a benchmark suite specifically made for approximate computing research. The suite is written in C++ and it is composed of applications from different domains like computer vision, data analytics, multimedia, and more. Like PolyBench, also AxBench was used to evaluate the performance of TAFFO in previous works [9]

ABSURD[3] is a suite of benchmarks for real-time systems written in C and C++. ABSURD implements algorithms for aviation applications (Airborne Collision Avoidance System and Lateral Navigation algorithm) and for data processing (artificial neural networks, Canny edge detector, image scaling algorithm, and a JPEG compressor). These benchmarks mimic realistic workloads for safety-critical systems and have been used in previous works to evaluate ASPIS [6, 10].

All the benchmarks were run on an Intel(R) Core(TM) i9-14900HX CPU, using the front end of clang-18 with two optimization levels: O3 and O0. The LLVM-IR outputted by clang was then processed by the TDA pass prior to any other transformation pass. The quantitative results of TDA's type inference are shown in Figure 6. From the result of the TDA pass, for each benchmark and for each optimization level x, we gathered different metrics[4]: the total number of pointer values in the module, the total number of alias pointer types Px recovered by TDA — always greater or equal with respect to the number of pointer values in the module —, the number of transparent types recovered, the number of partially transparent types recovered (meaning types which are neither transparent nor opaque pointers), and finally the number of opaque pointer types which remained opaque after TDA. Then, we calculated the percentages of transparent, partially transparent and opaque aliases recovered, relative to the total number of aliases Px. Respectively Tx, PTx and OPx.

Figure 6 shows that the TDA pass was able to recover up to 100% transparent types in most benchmarks from the ABSURD suite, both in O0 and O3 optimization levels, with an harmonic average of 84.72% and 87.99% for O0 and O3 respectively. We can notice that TDA performs on average marginally better in O3 pipelines, with singular examples in which T3 is significantly greater than T0, like for `fft`, `inversek2j` and `sobel`. This is because, despite the O3 optimizations introduce more non-idealities, they also reduce the number of pointers to deduce and thus the total number of

[3] ABSURD suite available at: https://github.com/HEAPLab/ABSURD
[4] Raw results available at: https://doi.org/10.5281/zenodo.17570563

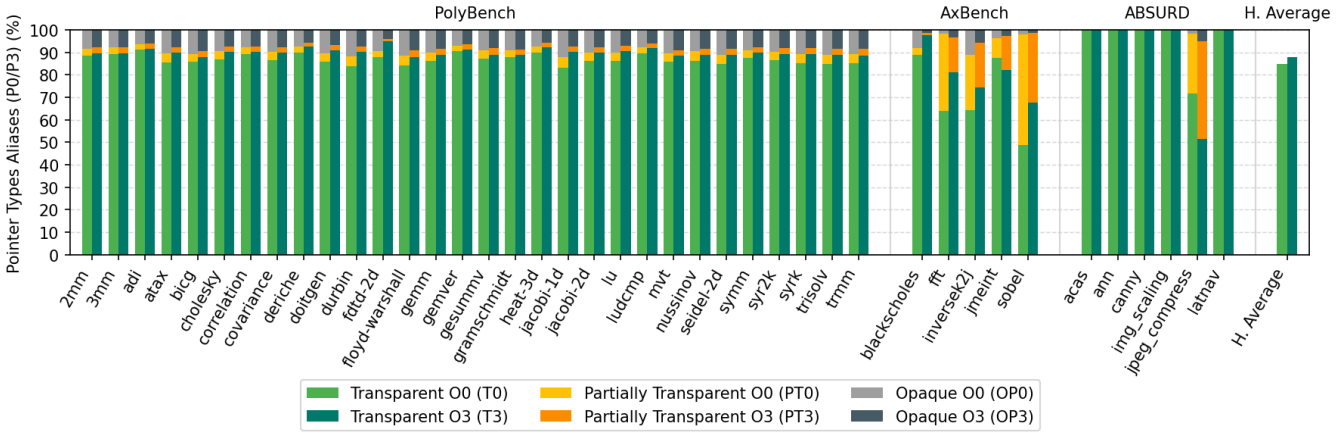

Figure 6. Distribution of pointer types aliases recovered by TDA among all benchmarks compiled in O0 and O3 optimization levels, categorized as transparent, partially transparent and opaque.

aliases. Another note to make is that even in cases in which T0 and T3 are low, like in the aforementioned benchmarks themselves, the respective percentages of opaque pointers OP0 and OP3 are still marginal, with the remaining aliases being partially transparent like showed by PT0 and PT3.

Additionally, we assessed the time impact of TDA relative to the O3 compilation pipeline, resulting in an harmonic average of 8.57% increase in time. From a theoretical standpoint, we also know that TDA runs in polynomial time, as its time complexity is the same of a dataflow analysis: $O(N \cdot |S| \cdot f)$, where N is the number of nodes of the flow-graph, $|S|$ is the cardinality of the largest output set, and f is the complexity of the flow equations. In our case this results in $O(N \cdot |V| \cdot |T|)$, where $|V|$ is the cardinality of the set of values $|V|$ in the program, and $|T|$ is the number of nodes in the graph of type T with the largest graph in the program.

It is worth noting that, despite the limited state of the art on the subject, it was not possible to compare TDA with other works based on the same metrics. In [23], the authors utilize TBAA metadata and the CodeQL [4] tool as a ground truth to verify the accuracy of the derived types. Unfortunately, neither of these ground truths is suitable for our tool. The TDA pass relies on the information of TBAA metadata, which makes it meaningless to use them as ground truth since our results are always a superset of the results achievable using TBAA only. CodeQL, on the other hand, is a tool that creates a queryable database from source code, extracting type information in the process. However, TDA is designed to be used at any stage within the LLVM optimization pipeline. This means that most of the relations with the original variables, such as variable names and stack space allocation, are optimized out in the process and are not recoverable, making CodeQL not usable as a ground truth for this work. On the other hand, the work of [20] only discusses coverage results, without providing further information on the validation approach. Due to these reasons, we have decided to move forward with validation through experimentation. We modified TAFFO and ASPIS to incorporate inquiries about type information from the TDA pass. Then, we used all the benchmarks reported in Figure 6 with their corresponding toolchain and verified that with the use of the TDA pass, their expected behavior was fully restored. The experiments showed that the types recovered from TDA were always perfectly adequate for the purpose of the frameworks under test, empirically demonstrating that TDA is able to recover the most relevant types.

6 Conclusions

This article tackled the problem introduced by LLVM's transition to opaque pointers. While this shift has simplified the IR and reduced compile times, it has also complicated many existing LLVM passes and forced some projects to remain tied to outdated LLVM versions. To bridge this gap, we have developed Type Deduction Analysis, an on-demand analysis pass for LLVM 17 and later. Our approach successfully reconstructs transparent pointer types by conducting an interprocedural data-flow analysis of SSA value uses throughout the program. The effectiveness of our solution was tested on two open-source toolchains, TAFFO and ASPIS, achieving up to 100% pointer type recovery while maintaining their runtime behavior correct. In the future, we aim to expand support for additional LLVM front-ends. Currently, only Clang generates TBAA metadata, which creates a dependency between our TDA and a specific front-end. Moreover, we will focus on extending TDA to emit confidence-annotated type outputs for variables whose type cannot be assessed univocally, so that subsequent transforms can use this information to perform conservative or aggressive optimizations.

Acknowledgements

This work has received funding from the National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data, and Quantum Computing and from the ISOLDE Chips JU project (grant nr. 101112274).

Data-Availability Statement

The replication package of this paper and the raw results of the experimental campaign are available as an additional artifact [2]. The open-source repository of TDA is available on GitHub.

References

[1] Lars Ole Andersen and Peter Lee. 2005. Program Analysis and Specialization for the C Programming Language. https://api.semanticscholar.org/CorpusID:20876553

[2] Anonymous. 2025. *Type Deduction Analysis Supplementary Material.* doi:10.5281/zenodo.17570563

[3] Anargyros A Argyros. 2024. *Recovering LLVM-IR type information using static analysis.* Master's thesis. University of Athene.

[4] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:25. doi:10.4230/LIPIcs.ECOOP.2016.2

[5] George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. 84–104. doi:10.1007/978-3-662-53413-7_5

[6] Davide Baroffio, Federico Reghenzani, and William Fornaciari. 2024. Enhanced Compiler Technology for Software-based Hardware Fault Detection. *ACM Trans. Des. Autom. Electron. Syst.* 29, 5, Article 91 (Sept. 2024), 23 pages. doi:10.1145/3660524

[7] Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. 2022. TAFFO: The compiler-based precision tuner. *SoftwareX* 20 (2022), 101238.

[8] Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. 2021. Architecture-aware Precision Tuning with Multiple Number Representation Systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC).* 673–678. doi:10.1109/DAC18074.2021.9586303

[9] Daniele Cattaneo, Michele Chiari, Gabriele Magnani, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. 2021. FixM: Code generation of fixed point mathematical functions. *Sustainable Computing: Informatics and Systems* 29 (2021), 100478. doi:10.1016/j.suscom.2020.100478

[10] Emilio Corigliano, Davide Baroffio, and Federico Reghenzani. 2025. *Hardening mixed-criticality systems: a low-overhead compiler-based approach.* Master's thesis. Politecnico di Milano.

[11] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based alias analysis. *SIGPLAN Not.* 33, 5 (May 1998), 106–117. doi:10.1145/277652.277670

[12] Susan Horwitz. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 1–6. doi:10.1145/239912.239913

[13] International Organization for Standardization and International Electrotechnical Commission 2023. ISO/IEC 14882:2023 — Programming languages — C++.

[14] William Landi and Barbara G. Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. 27, 7 (July 1992), 235–248. doi:10.1145/143103.143137

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[16] Gabriele Magnani, Daniele Cattaneo, Lev Denisov, Giuseppe Tagliavini, Giovanni Agosta, and Stefano Cherubin. 2025. Synergistic Memory Optimisations: Precision Tuning in Heterogeneous Memory Hierarchies. *IEEE Trans. Comput.* 74, 9 (2025), 3168–3180. doi:10.1109/TC.2025.3586025

[17] William S. Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. *CoRR* abs/2010.01709 (2020). arXiv:2010.01709 https://arxiv.org/abs/2010.01709

[18] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2004. *Principles of program analysis.* Springer Science & Business Media.

[19] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002), 63–75. doi:10.1109/24.994913

[20] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. 2, 1 (April 2015), 1–69. doi:10.1561/2500000014

[21] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test* 34, 2 (2017), 60–68. doi:10.1109/MDAT.2016.2630270

[22] Tomofumi Yuki. 2014. Understanding polybench/c 3.2 kernels. In *International workshop on polyhedral compilation techniques (IMPACT).* 1–5.

[23] Jinmeng Zhou, Ziyue Pan, Wenbo Shen, Xingkai Wang, Kangjie Lu, and Zhiyun Qian. 2025. Type-Alias Analysis: Enabling LLVM IR with Accurate Types. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 2203–2226.

Received 2025-11-11; accepted 2025-12-10