NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Recovering LLVM-IR type information using static analysis

Anargyros A. Argyros

Supervisor: **Yannis Smaragdakis,** Professor NKUA

ATHENS

JANUARY 2O24

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάκτηση πληροφορίας τυπών στο LLVM-IR με χρήση στατικής ανάλυσης

Ανάργυρος Α. Αργυρός

Επιβλέπων: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΑΝΟΥΑΡΙΟΣ 2024

BSc THESIS

Recovering LLVM-IR type information using static analysis

Anargyros A. Argyros
S.N.: 1115201900014

SUPERVISOR: **Yannis Smaragdakis,** Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάκτηση πληροφορίας τυπών στο LLVM-IR με χρήση στατικής ανάλυσης

Ανάργυρος Α. Αργυρός
Α.Μ.: 1115201900014

ΕΠΙΒΛΕΠΩΝ: Γιάννης Σμαραγδάκης, Καθηγητής ΕΚΠΑ

ABSTRACT

The ”LLVM Compiler Infrastructure Project” [3] is an extremely popular collection of tools and technologies for building compilers. One of LLVM’s central features is a Static Single Assignment (SSA) form code representation known as the LLVM Intermediate Representation (LLVM-IR).The LLVM-IR is a target language for a lot of compilers that want to make use of the LLVM framework such as Clang/Clang++, Rustc, Swift and more.

A lot of static analysis tools choose to run their analyses at the LLVM-IR level as it encodes the necessary information required to perform those analyses, while simultaneously filtering out language specific high-level concepts that may confuse or add complexity to the process. In addition, analyzing at the intermediate representation level allows them to be compatible with a plethora of languages, as the respective LLVM-IR will be generated by the language’s native compiler. Cclyzer-Soufflé is such a tool, it utilizes the LLVM framework to parse the LLVM-IR generated from a language’s native compiler, generates facts about the program’s source code and then executes various static analysis algorithms defined in datalog.

In version 17 of LLVM framework a decision was made to move from a strongly typed pointer type system to opaque pointer types.This change obscured a lot of information about pointer types in the LLVM-IR, that many static analysis tools including Cclyzer-Soufflé required in order to work effectively. Although pointer type information is no longer directly available, a significant portion remains in the LLVM-IR and can be inferred though static analysis methods. In this work we make use of static analysis to recover missing information about pointer types from the LLVM-IR level as well as integrate this type-inference mechanism we developed, in the Cclyzer toolchain.

SUBJECT AREA: Static Analysis

KEYWORDS: Static Analysis, Type Inference, Datalog, LLVM-IR, Compilers

ΠΕΡΙΛΗΨΗ

Το "LLVM Compiler Infrastructure Project" [3] είναι μια εξαιρετικά δημοφιλής συλλογή εργαλείων και τεχνολογίων για την κατασκευή μεταγλωττιστών. Ένα από τα κεντρικά χαρακτηριστικά του LLVM είναι ή αναπαράσταση κώδικα σε μορφή Static Single Assignment (SSA) γνωστή ως LLVM Intermediate Representation (LLVM-IR). Η LLVM-IR αποτελεί την γλώσσα που στοχεύουν να παράξουν πολλοί μεταγλωττιστές που θέλουν να κάνουν χρήση του πλαισίου LLVM όπως οι Clang/Clang++, Rustc, Swift και άλλοι. Πολλά εργαλεία στατικής ανάλυσης επιλέγουν να εκτελούν τις αναλύσεις τους σε επίπεδο LLVM-IR καθώς σε αυτην κωδικοποιούνται οι απαραίτητες πληροφορίες που απαιτούνται για την εκτέλεση αυτών των αναλύσεων, ενώ ταυτόχρονα φιλτράρονται έννοιες υψηλού επιπέδου που είναι ειδικές μιας γλώσσας και που μπορεί να μπερδέψουν ή να προσθέσουν πολυπλοκότητα στην διαδικασία της ανάλυσης. Επιπλέον, η ανάλυση του στο επίπεδο της ενδιάμεσης αναπαράστασης τους επιτρέπει την συμβατότητα με μια πληθώρα γλωσσών, καθώς το αντίστοιχο LLVM-IR μπορεί να δημιουργηθεί από εγγενείς μεταγλωττιστές της γλώσσας. Το Cclyzer-Soufflé είναι ένα τέτοιο εργαλείο, που χρησιμοποιεί το LLVM framework για την ανάλυση του LLVM-IR που δημιουργείται από τον εγγενή μεταγλωττιστή μιας γλώσσας, καταγράφει έννοιες και γεγονότα σχετικά με τον πηγαίο κώδικα του προγράμματος και στη συνέχεια εκτελεί διάφορους αλγόριθμους στατικής ανάλυσης υλοποιημένους στην γλώσσα datalog. Στην έκδοση 17 του LLVM framework λήφθηκε η απόφαση να μετακινηθεί από ένα αυστηρό σύστημα τύπων δείκτών σε αδιαφανείς τύπους δεικτών. Αυτή η αλλαγή είχε ως αποτέλεσμα πολλές πληροφορίες σχετικά με τους τύπους δείκτών στο LLVM-IR να αποκρυφθούν, πολλά εργαλεία στατικής ανάλυσης, συμπεριλαμβανομένου του Cclyzer-Soufflé βασιζόντουσαν σε αυτήν την πληροφορία για να λειτουργήσουν αποτελεσματικά. Αν και οι πληροφορίες για τους τύπους των δείκτων δεν είναι πλέον άμεσα διαθέσιμες στο LLVM-IR, μεγάλο μέρος αυτής της πληροφορίας παραμένει έμμεσα διαθέσιμη και μπορεί να συναχθεί μέσω μεθόδων στατικής ανάλυσης. Σε αυτή την εργασία χρησιμοποιούμε στατική ανάλυση για να ανακτήσουμε τις πληροφορίες που έχουν αποκρυφθεί σχετικά με τους τύπους δεικτών από το επίπεδο LLVM-IR καθώς και ενσωμάτωνουμε τους μηχανισμούς που αναπτύξαμε, στην αλυσίδα εργαλείων του Cclyzer-Soufflé.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Στατική Ανάλυση, Συναγωγή Τύπων, Datalog, LLVM-IR, Μεταγλωττιστές

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Yannis Smaragdakis for the invaluable insight and support he provided during the completion of this thesis. This work would not have been possible had it not been for his immense experience and deep understanding of static analysis concepts. I would also like to thank Konstantinos Triantafyllou for participating in the discussion process and providing insight on the problems at hand.

CONTENTS

LIST OF TABLES

1. INTRODUCTION

The LLVM Project [3] is a collection of modular and reusable compiler and toolchain technologies. It provides various tools for compiler developers, most notably, optimizer and code generation tools. Currently there are compilers utilizing LLVM as a backend for a number of languages such as C/C++ Rust Swift Scala and more. These compilers transform the original source code into LLVM-IR, leaving various optimizations and code generation to the LLVM backend.

The LLVM-IR is a Static Single Assignment [1] based instruction set language.The LLVM-IR is a strongly typed language, meaning that every virtual register is associated with an immutable type, which also as to be explicitly stated in every instance it is relevant. LLVM version 17 slightly relaxed the strict type system of the IR by completely replacing pointer types with a singular opaque pointer type. This fundamental change created compatibility issues with a lot of projects utilizing the LLVM toolchain.

CClyzer-Soufflé is one of the affected tools. Cclyzer-Soufflé is a reimplementation of the Cclyzer [2] static analysis tool in datalog, utilizing the Soufflé datalog engine. CClyzer-Soufflé parses the LLVM-IR generated by a languages native compiler and extracts information in the form of datalog relations. These facts about the target program are used by CClyzer-Soufflé to execute various static analysis algorithms including points to analysis and callgraph generation.

These analyses relied on information from the strong typing system in the LLVM-IR and suffered significant losses on precision but also completeness when the change to opaque pointer types was made. In this work we explain how we implemented a basic type inference system that recovers some of the missing pointer type information as well as how we hijack CClyzer-Soufflé's core analysis to propagate that information across the program.

This thesis is organized as follows:

- In Chapter 2 we will provide background for some of the technologies involved as well as explain the implications of opaque pointers in static analysis.
- In Chapter 3 we will describe the type inference datalog algorithm used to recover missing pointer type information from the LLVM-IR.
- In Chapter 4 we will explain how we integrated the aforementioned type inference with CClyzer-Soufflé's core analysis.
- In Chapter 5 we evaluate the precision and completeness of CClyzer-Soufflé's analysis and compare the results with previous versions.
- The final Chapter 6 accounts the conclusions we reached.

2. BACKGROUND

2.1 LLVM-IR

The LLVM-IR is a strongly typed language. Each SSA register has a static type, that type is immutable and the IR syntax expects it to be stated every time the register is used. For example, a simplified syntax for the load instruction of the LLVM-IR before the change to opaque pointers can be seen in listing 2.1.

Listing 2.1: Load Instruction Syntax

```
<result> = load <ty>, <ty>* <pointer>
```

The load instruction loads a value of type <ty> from memory pointed to by the <ty>* type pointer. A usage example of the above can be seen in listing 2.2.

Listing 2.2: Load Instruction Example - Non-Opaque pointers

```
%reg = load i32, i32* %ptr
```

Where the i32 is the type for 32-bit integers and i32* is a pointer to a i32 value. In this version of LLVM-IR, it is trivial to acquire the type of a pointers pointee value as it is encoded in the pointers type name.

In LLVM-17 all pointer types were replaced with a generic opaque pointer type ”ptr”. With that change, the updated syntax for the load instruction is now:

Listing 2.3: Load Instruction Syntax - Opaque Pointers

```
<result> = load <ty>, ptr <pointer>
```

And the example in listing 2.2 would now become:

Listing 2.4: Load Instruction Example - Opaque pointers

```
%reg = load i32, ptr %ptr
```

In version 17 of LLVM-IR which has opaque pointers, the type of pointee values is no longer encoded in pointer types, nor maintained by the LLVM framework. From the perspective of static analysis tools, this change could be slightly beneficial, as the previous pointer type system had created necessity for a lot of typecasting instructions which could bloat up an analysis.

However, having explicit type information for every entity was extremely convenient for tools such as Cclyzer, as they relied on that information to implement type and field sensitivity components of their analysis. After the switch to opaque pointer types a lot of that information was obscured, effectively nullifying the benefits of field and type sensitivity, or downright breaking their core analysis algorithms.

In addition, opaque pointers are problematic when trying to parse the composition of user defined types. Pointers are very common components in user defined memory structures. Before the change to opaque pointers, it was possible to construct the type hierarchy of user defined types by recursively visiting each of its components. In LLVM-17 this is no longer possible as the connection with some components might be obscured behind an opaque pointer.

For example take a user defined type, StructA that has an integer field and a pointer to another user defined type, StructB.

Listing 2.5: Composition of struct types with non-opaque pointers

```
%struct.StructA = type { i32, %struct.StructB* }
%struct.StructB = type { i32 }
```

With non-opaque pointers the association between the two struct types is obvious. However, when using opaque pointers that connection gets obscured as can be seen in listing 2.6

Listing 2.6: Composition of struct types with opaque pointers

```
%struct.StructA = type { i32, ptr }
%struct.StructB = type { i32 }
```

Although various bits of information about pointer types are no longer directly present in the LLVM-IR, they can still be found indirectly by examining the usage of individual pointers.

2.2 CCLYZER-SOUFFLÉ AND DATALOG

Cclyzer-Soufflé [2] is a static analysis tool which utilizes the Soufflé datalog engine in order to analyze LLVM-IR code. In datalog, algorithms are defined declaratively, with an initial set of relations between objects and rules that monotonically ”prove” more relations until a fixpoint is reached.

An example of a Soufflé datalog algorithm can be seen in the listing below:

Listing 2.7: Example of a datalog program that calculates reachable nodes in a directed graph

```
//declarations for types and rules
.type Node <: symbol
.decl node(n:Node)
.decl edge(n1:Node,n2:Node)
.decl reachable(from:Node,to:Node)

//rule definitions
reachable(n1,n2):-
    edge(n1,n2).

reachable(n1,n2):-
    reachable(n1,n3),
    edge(n3,n2).

//input facts about our graph
node(”A”)
node(”B”)
node(”C”)
edge(”A”,”B”)
edge(”B”,”C”)
```

In the above program, the Soufflé engine ”proves” a relation `reachable("A","B")` using the first rule and the `edge("A","B")` relation. With the addition of `reachable("A","B")` in our knowledge base, in the next iteration the engine proves `reachable("A","C")` using the second rule by substituting n1 with Node A, n2 with Node C and n3 with Node B.

Cclyzer-Soufflé focuses on points-to analysis and pointer analysis, meaning it tries to determine information about a pointers set of potential values. In order to achieve sufficient precision within realistic compute times for realistic programs, Cclyzer-Soufflé employs auxiliary techniques such as context sensitivity and field sensitivity [5]. Field sensitivity especially, relied on information about pointer types to produce more accurate points to sets for compound types. With that information now no longer available in the LLVM-IR, the performance of Cclyzer-Soufflé's analysis suffered greatly.

3. TYPE INFERENCE

3.1 Inference Core Logic

As we mentioned in Chapter 2, a lot of information obscured by the introduction of opaque pointers is still indirectly present in the LLVM-IR and can thus be inferred with the use of static analysis.

The LLVM-IR remains a strongly typed language, by examining the type information found in instructions that involve a pointer we can infer facts about that pointer. Consider again the example in listing 2.4. Although the type of the pointer the load instruction loads from is ”ptr”, the strongly typed design of the LLVM-IR language requires the actual type being loaded to be present in the instruction. Since the type of the value loaded from that pointer is i32, that pointer can be an i32* type pointer. Similar inferences can be made using information from instructions such as store, getelementptr and alloca.

3.2 Inference Implementation

In order to source type information from the aforementioned LLVM-IR instructions we introduce a new datalog relation, `pointer_has_type`. This relation records all pointer types associated with a pointer.

Listing 3.1: pointer_has_type Declaration

```
.decl pointer_has_type(pointer:symbol , ptr_type: PointerType)
```

3.2.1 Load Instructions

Listing 3.2: Load Instruction Syntax

```
<result> = load <ty>, <ty>* <pointer>
```

As mentioned previously in section 2.1 load instructions carry information about the type of the value being loaded from a pointer [4]. We visit each instruction with the following datalog rule and extract the type of the variable being assigned to. Finally we construct the appropriate type for the pointer by appending a * symbol to the extracted type.

Listing 3.3: pointer_has_type definition for load instructions

```
pointer_has_type(pointer , ptr_type) :-
    //indentify load instruction
    load_instruction(instr),
    load_instruction_address(instr,pointer),
    instruction_assigns_to(instr,var),
    //build type for pointer
    variable_has_type(var,var_type),
    ptr_type = cat(var_type,"*").
```

Inversely, for pointers that we have made inferences for from other instructions, we can make inferences about the type of the value being loaded.

Listing 3.4: pointer_has_type definition for load instructions

```
pointer_has_type(var , as(var_type,PointerType)) :-
    //indentify load instruction
    load_instruction(instr),
    load_instruction_address(instr,pointer),
    instruction_assigns_to(instr,var),
    //build type for var
    pointer_has_most_specific_type(pointer,ptr_type),
    pointer_type_has_component(ptr_type,var_type).
```

3.2.2 Store Instructions

Store instructions load a value to a memory address pointed to by a pointer. Similarly with load instructions, store instructions contain type information about the value being stored.

Listing 3.5: Store Instruction Syntax

```
store <ty> <value>, ptr <pointer>
```

From the above, we can infer that the type of <pointer> is <ty>*. We supplement the `pointer_has_type` datalog rule with this information.

Listing 3.6: pointer_has_type definition for store instructions

```
pointer_has_type(pointer , ptr_type) :-
    //identify store instructions
    store_instruction(instr),
    store_instruction_address(instr,pointer),
    store_instruction_value_type(instr,var_type),
    //build type for pointer
    ptr_type = cat(var_type,"*").
```

In addition, for pointers we have already made inferences for its type, we attempt to make inferences for the type of the value being stored.

Listing 3.7: pointer_has_type definition for store instructions

```
pointer_has_most_specific_type(variable,var_type):-
    //identify store instructions
    store_instruction(instr),
    store_instruction_address(instr,pointer),
    store_instruction_value(instr,variable),
    pointer_has_most_specific_type(pointer,pointer_type),
    //get type for variable
    pointer_type_has_component(pointer_type,var_type).
```

3.2.3 GetElementPtr Instructions

The getelementptr instruction performs address calculations, given a memory address pointed to by a pointer the type of the value stored in that address and various offsets.

Listing 3.8: Getelementptr Instruction Syntax

```
<result> = getelementptr <ty>, ptr <ptrval>{, <ty> <idx>}*
```

The <ty> type used as a basis for the address calculations can be used to infer the type of the <ptrval> pointer, which is <ty>*. The datalog relation that implements this logic follows in listing 3.9

Listing 3.9: pointer_has_type definition for getelementptr instructions

```
pointer_has_type(pointer , ptr_type) :-
    //indentify GEP instructions
    getelementptr_instruction(instr),
    getelementptr_instruction_base(instr,pointer),
    getelementptr_instruction_base_type(instr,base_type)
    //build type for pointer
    ptr_type = cat(base_type,"*").
```

3.2.4 Alloca Instructions

Alloca instructions allocate stack memory for a value of a given type, and return a pointer to that memory.

Listing 3.10: Alloca Instruction Syntax

```
<result> = alloca <type>
```

We can infer the type of the <result> pointer to be <type>*. We supplement the `pointer_has_type` relation as follows in listing 3.11

Listing 3.11: pointer_has_type definition for alloca instructions

```
pointer_has_type(pointer , ptr_type) :-
    //indentify alloca instruction
    alloca_instruction(instr),
    alloca_instruction_type(instr,var_type),
    instruction_assigns_to(instr,pointer),
    //build type for pointer
    ptr_type = cat(var_type,"*").
```

3.3 Global Variables

In LLVM-IR all global variables are memory constructs that live in compile-time allocated memory. As all memory constructs in LLVM, they are accessed through pointers.

Listing 3.12: Global variable declaration example

```
@G = external global i32
```

In the above listing , the SSA value @G is a pointer representing a global variable of type i32. But since @G is a pointer, it is used in LLVM-IR instructions with the ”ptr” type. We add the type information found in global variable declarations to our `pointer_has_type` relation.

Listing 3.13: pointer_has_type definition for global variables

```
pointer_has_type(pointer , ptr_type) :-
    global_variable(pointer),
```

```
    global_variable_value_type(pointer, value_type),
    global_variable_has_type(pointer, ptr_type).
```

3.4 Type Hierarchy

Because the LLVM-IR is strongly typed, each pointer has a unique type. However, during type inference multiple types for a given pointer may be inferred. A simple example of this is demonstrated with a double pointer in listing 3.14.

Listing 3.14: Example of pointer with multiple inferred types

```
%1 = alloca ptr, align 8
%2 = load ptr, ptr %1
%3 = load ptr, ptr %2
%4 = load i32, ptr %3
```

From the last instruction we infer the type of the %3 register to be i32*, and from the second to last instruction we also infer the type of %3 to be ptr*. Although both of those two types are semantically correct, the i32* type is much more specific and provides us with the most information. Since our goal is to provide accurate types for our static analysis tool, we need to implement a type hierarchy system that will rank our inferred types based on how much information they provide. We implement this system using three datalog relations.

The `more_specific_type` relation expresses that a type is more ”specific” than another type. The more information about the type of the pointee value a pointer type holds, the more specific it is. We define pointer ”specificity” using the following 2 rules:

1. The generic ”ptr” pointer type is the least specific pointer type.
2. A pointer ”A” type is more specific than a pointer type ”B” if A’s component type is more specific than B’s component type. A pointer’s component type is the type of its pointee value.

Listing 3.15: Datalog implementation of the more_specific_type relation

```
.decl more_specific_type(st:Type, t:Type)

more_specific_type(st, "ptr") :-
    type(st),
    st != "ptr".

more_specific_type(st, t) :-
    pointer_type_has_component_type(st, st_ele),
    pointer_type_has_component_type(t, t_ele),
    more_specific_type(st_ele, t_ele).
```

The `pointer_has_more_specific_type` relation is a per-pointer version of the `more_specific_type` relation, and finally `pointer_has_most_specific_type` is used to select the most specific pointer type we could infer for a given pointer.

Listing 3.16: Datalog implementation of the pointer_has_more_specific_type and pointer_has_most_specific_type relations

```
.decl pointer_has_more_specific_type(pointer:Variable, st:Type,
t:Type)

pointer_has_more_specific_type(ptr, st, t) :-
    pointer_has_type(ptr, t),
    pointer_has_type(ptr, st),
    more_specific_type(st, t).

.decl pointer_has_most_specific_type(pointer:Variable, ty:Type)

pointer_has_most_specific_type(ptr, st) :-
    pointer_has_type(ptr, st),
    !pointer_has_more_specific_type(ptr, st, _).
```

Due to the nature of the SSA form and the limited scope of the inference algorithm thus far, the `pointer_has_most_specific_type` set will only contain a single type for each pointer. Realistically however, this is not always the case as pointers can be used to reference objects of different types. For example a C void pointer can be used as a reference to arbitrary memory. In order to make complicated inferences like that we need our analysis to consider the global scope of the program.

4. INTEGRATION WITH CCLYZER

4.1 Basic integration with variable_has_type

The first step to integrating our inference mechanism with Cclyzer-Soufflé is to export the results back into its relations. More specifically the results of our `pointer_has_most_specific_type` are added into Cclyzer-Soufflé's `variable_has_type` relation.

Listing 4.1: Integration of pointer_has_most_specific_type

```
variable_has_type(as(pointer, Variable),type):-
    pointer_has_most_specific_type(pointer , type).
```

4.2 Integration with Cclyzer-Soufflé's interprocedural analysis

4.2.1 The need for interprocedural flow analysis

This process we have described thus far is adequate to reliably infer types for pointers that are directly referenced in the instructions we source the type information from, as well as account for some intraprocedural data flow. However, due to the nature of the SSA representation and the general complexity of realistic programs, more sophisticated methods are required to track the flow of type information through the program. In the following example we demonstrate the need for a global value flow analysis.

The @foo function takes an integer pointer as argument and does something with it. The @main function allocates stack memory for an integer value and an integer pointer and calls @foo.

```
define i32 @foo(ptr %0){
    %2 = alloca ptr, align 8
    store ptr %0, ptr %2
            .....
}

define i32 @main(){
    %1 = alloca i32
    %2 = alloca i32
    %3 = alloca ptr
    store i32 0, ptr %1
    store ptr %2, ptr %3
    %4 = load ptr, ptr %3
    %5 = call i32 @foo(ptr %4)
    ret i32 %5
}
```

Our inference algorithm would be able to infer that the `%4` register in @main is an i32* type pointer, but would not be able to correlate that type with the type of the `%0` argument register in @foo. Tracking the data flow of our program would allow us to associate the inferred type of the function arguments with the function call operands as well as propagate

our results across the program. Dataflow and interprocedural analyses are already implemented in our tool, Cclyzer-Soufflé. In order to receive the benefits of those analyses, we integrate our inference mechanism by allowing them to draw information from it.

4.2.2 Type indication mechanism

Before the change to opaque pointers, Cclyzer-Soufflé was able to reliably know the type of every stack or global allocation object by reading that information off of the respective alloca instruction or global variable declaration. However, unlike stack and global allocations, heap allocations require the invocation of allocator functions (e.g., malloc) found in libraries instead of language built-in heap allocation mechanisms. In addition, as most allocator functions only require the allocation size in bytes as an argument and are agnostic towards the specific type of the data being allocated. As a result it is not possible to infer the type of the allocation just by looking at the invocation of the allocator.

In order to overcome this problem Cclyzer-Soufflé uses the concept of type indications to infer the type of those heap allocations. In Cclyzer's logic, type indications are bitcast instructions following a heap allocation that typecast the pointer returned by the allocator function to the appropriate type. Cclyzer-Soufflé records the information found in those bitcast instructions and constructs new allocation objects with the type information and integrates them with the core analysis.

Listing 4.2: LLVM-IR example of bitcast instructions used as type indications

```
define i32 @main(){
    %1 = alloca i32
    %2 =  alloca i32*
    store i32 0, i32* %1
    %3 = call noalias i8* @malloc(i64 4)
    %4 = bitcast i8* %3 to i32*
    store i32* %4, i32** %2
    %5 = load i32*, i32** %2
    %6 = load i32, i32* %5
    ret i32 %6
}
```

The pointer returned from the malloc call in the above example is immediately casted into the intended type using a bitcast instruction. Cclyzer-Soufflé creates an additional allocation object for that allocation with the appropriate type.

In the opaque version of LLVM, not only can we not reliably know the exact type of some stack allocation objects, but the type indication system for heap allocation also breaks apart as the bitcast instructions it relied upon are now redundant and omitted by the compiler. To remedy these problems we redesign Cclyzer-Soufflé's type indication system to additionally draw information from the `pointer_has_type` relation of our type inference mechanism. This change supplies Cclyzer-Soufflé with information about both stack and heap allocations and allows it to propagate that information across the entire program through it's points-to-analysis.

Listing 4.3: LLVM-IR example in version 17

```
define i32 @main() {
    %1 = alloca i32
```

```
    %2 = alloca ptr
    store i32 0, ptr %1
    %3 = call ptr @malloc(i64 4)
    store ptr %3, ptr %2
    %4 = load ptr, ptr %2
    %5 = load i32, ptr %4
    ret i32 %5
}
```

As it can be seen in the above example, in version 17 of the LLVM-IR the bitcast instructions are omitted due to the existence of the opaque pointer type. In order to recover the type of the allocation object, we must rely on the results of the `pointer_has_type` relation from our analysis. For the above code snippet we are able to infer the type of the `%3` register with the following steps:

- `%5 = load i32, ptr %4` → `%4` is of type i32*
- `%4 = load ptr, ptr %2` → `%2` is of type i32**
- `store ptr %3, ptr %2` → `%3` is of type i32*

With the appropriate pointer type inferred by our analysis we implement new `type_indication` rules as follows.

Listing 4.4: Type indication rules that draw information from the inference mechanism

```
type_indication(?type, ?aCtx, ?alloc) :-
    untyped_allocation(?alloc),
    var_points_to(?aCtx, ?alloc, _, ?pointer_var),
    pointer_has_type(?pointer_var,?ptrType),
    pointer_type_has_component(?ptrType,?type).

type_indication(?PointerAllocType, ?aCtx, ?PointerAlloc) :-
    untyped_allocation(?PointerAlloc),
    ptr_points_to(?PointeeCtx,?PointeeAlloc, ?aCtx, ?PointerAlloc),
    type_indication(?PointeeAllocType, ?PointeeCtx, ?PointeeAlloc),
    pointer_type_has_component(?PointerAllocType,?PointeeAllocType).
```

The first rule serves to associate an allocation object with the pointers that point to it. A pointer that has been observed to have a type T* and points to an allocation object acts as a indication that the allocation object is of type T.

The second rule covers the case where an allocation object A acts as a pointer to another allocation object B. If there is a type indication that object A is of type T* and object A points to object B, then this acts as a type indication that object B is of type T.

With the inference mechanism's information integrated into Cclyzer-Soufflé's type indication mechanism, the tool has recovered the majority of the information lost due to the opaque pointers change.

5. EVALUATION

We evaluate the performance of our analysis on programs from GNU's coreutils package and wherever it is applicable, we compare the results of Cclyzer-Soufflé plus the inference mechanism, with the results of previous versions of Cclyzer-Soufflé.

The first metric we employ is the percentage of pointer variables that we could infer a specific type for. Since the purpose of our analysis is to recover the pointer types that were obscured by the switch to opaque pointers, this metric serves as an indication of how much information we were able to actually recover.

Table 5.1: Pointer variables with specific inferred types.

Program Name	N of pointers with recovered type	N of total pointers	% Recovered
rm.bc	2554	3107	82.2%
join.bc	1179	1527	77.2%
kill.bc	515	714	71.1%
cat.bc	617	818	75.4%

Our analysis was able to recover a considerable amount of the obscured information, up to 82% in some cases, and around 76% on average across the GNU Coreutils codeset.

We employ a similar approach with allocations, measuring the percentage of allocation objects with specific inferred types. For this metric, we compare the results of Cclyzer-Soufflé's + type inference mechanism against the LLVM-17 version of Cclyzer-Soufflé, but without the inference mechanism.

Table 5.2: Allocation objects with specific inferred types.

	LLVM-17 + inference	LLVM-17 no inference
rm.bc	86.5%	69.4%
join.bc	89.5%	71.8%
kill.bc	88.8%	76%
cat.bc	87.6%	74.4%

Another metric we employ in order to evaluate our analysis is the percentage of pointer variables that are not involved in any `var_points_to` relation. Meaning that our analysis could not deduce anything about the objects those variables may point to.

We compare the results against:

1. The LLVM-14 version of Cclyzer-Soufflé, before the switch to opaque pointers.
2. The LLVM-17 version of Cclyzer-Soufflé, but without the inference mechanism.

Table 5.3: Percentage of pointer variables without a points-to set.

	LLVM-17 + inference	LLVM-14	LLVM-17 no inference
rm.bc	16.8%(522 / 3107)	13.2%(384 / 2907)	55.8%(1625 / 2907)
join.bc	23.3%(356 / 1527)	16.2%(224 / 1378)	47.8%(652 /1362)
kill.bc	32.3%(231 / 714)	29.0%(174 / 600)	44.6%(268 / 600)
cat.bc	33.4%(274 / 818)	28.7%(201 / 699)	45.3%(317 / 699)

Comparing the results of the LLVM-17 + inference and LLVM-17 without inference we observe significant decrease in the percentage of pointers without points-to sets, which signifies that the analysis was improved in terms of completeness.

Since in the LLVM-14 version all pointer type information is directly available, our inference mechanism cannot outperform it. We use the results of version LLVM-14 as the optimum our inference could reach. By comparing the results of LLVM-17 + inference with LLVM-14 we observe on average 3% less pointers with a points-to set. This discrepancy indicates that there is information present in the LLVM-14 version that could not be recovered by the inference mechanism. We believe that the main reason for this discrepancy is the fact that our inference mechanism does not reconstruct compound types that include pointers in their fields. An example of such case can be seen in listing 2.5 and 2.6. It is possible that a similar mechanism to our type inference system could be used to reconstruct those types, but it is left for future work.

6. CONCLUSIONS

In this thesis we introduced some core concepts surrounding the LLVM Project, specifically the LLVM-IR and how it serves as an excellent target for static analysis tools to run their analyses on. We then explained the changes that came with the introduction of opaque pointers in LLVM version 17 as well how those changes affected the performance of static analysis tools such as Cclyzer-Soufflé. The core purpose of this thesis was to showcase a type inference mechanism that could recover the obscured information in a program through the use of static analysis. We evaluated the type inference mechanism we implemented and we found that it was capable of recovering up to 88% of the obscured information on various test programs from the GNU Coreutils package.

ABBREVIATIONS - ACRONYMS

SSA	Single Static Assignment
LLVM	Low Level Virtual Machine
LLVM-17	LLVM version 17
LLVM-14	LLVM version 14

BIBLIOGRAPHY

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, techniques, and tools. pages 369–370. Addison Wesley, 2006.

[2] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*, pages 84–104. Springer, 2016.

[3] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[4] LLVM Project. Llvm-17 language reference. https://llvm.org/docs/LangRef.html.

[5] Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.