Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Function Pointer Analysis for C Programs

submitted by

Stefan Stattelmann

on August 28, 2008

Supervisor

Prof. Dr. Reinhard Wilhelm

Advisor

Dr. Florian Martin

Reviewers

Prof. Dr. Reinhard Wilhelm
Prof. Dr. Bernd Finkbeiner

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, August 28, 2008

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, August 28, 2008

Abstract

Function pointers are a feature of the C programming language whose use obscures the control flow of a program and makes programs hard to analyze. Existing pointer analyses are able to resolve function pointers, but lack the capabilities to precisely distinguish function pointer variables within complex data structures.
The aim of this work is to develop a function pointer analysis which achieves this precision. It thereby allows a more precise analysis of programs with an intense usage of function pointers, as they are quite common in automotive software.

Contents

1 Introduction **1**
- 1.1 Function Pointers and Program Analysis . . . 1
- 1.2 Pointer Analysis . . . 1
- 1.3 Requirements . . . 2
- 1.4 Approach . . . 3
- 1.5 Overview . . . 3

2 Theoretical Foundations **4**
- 2.1 Program Analysis . . . 4
- 2.2 Data Flow Analysis . . . 5
- 2.3 Related Work . . . 6

3 Analysis Description **8**
- 3.1 Overview . . . 8
- 3.2 Memory Model . . . 9
- 3.3 Points-to Mapping . . . 10
- 3.4 Updating the Points-to Mapping . . . 12
  - 3.4.1 Assignments . . . 13
  - 3.4.2 Address-of Operator . . . 15
  - 3.4.3 Pointer Dereference . . . 15
- 3.5 Language Constructs requiring special handling . . . 16
  - 3.5.1 Pointer Arithmetic . . . 16
  - 3.5.2 Casting . . . 16
  - 3.5.3 Multidimensional Arrays . . . 17
  - 3.5.4 Unions . . . 17

4 Implementation **19**
- 4.1 Implementation Framework . . . 19
  - 4.1.1 Program Analyzer Generator (PAG) . . . 19
  - 4.1.2 ROSE Compiler Infrastructure . . . 20
  - 4.1.3 Static Analysis Tool Integration Engine (SATIrE) . . . 21
- 4.2 Required Extensions . . . 22
  - 4.2.1 Variable Renaming . . . 22

4.2.2 Control Flow Graph Refinement 22
4.2.3 External Functions 23
4.3 Further Extensions 24
4.3.1 Program Slicing 24
4.3.2 Optimistic Analysis 25

5 Results **27**
5.1 Test Programs 27
5.1.1 grep 27
5.1.2 diction 29
5.1.3 gzip 30
5.1.4 sim6809 30
5.1.5 Automotive software 30
5.2 Evaluation 31

6 Conclusion **33**
6.1 Summary 33
6.2 Outlook 33

References **35**

1 Introduction

1.1 Function Pointers and Program Analysis

Function pointers are a type of the C programming language that allow storing the address of a program routine within a variable and calling the routine through this variable. For programs that use function pointers, it cannot be determined directly which functions can be called during a run of the program because the runtime values of the function pointer variables are not known in general. This makes it quite hard to analyze such programs.

The information which function can be called at a given program point is necessary to construct the interprocedural control flow graph of the program. The control flow graph is required for a flow-sensitive interprocedural program analysis. Without precise knowledge about function pointers, it is not possible to construct an exact control flow graph, which will result in either imprecise or even wrong results of the analysis that uses the graph.

The effects of function pointers are not only relevant when analyzing a C program, they are also important for analyses on assembly level, since calls of function pointers in C are in general translated to computed calls on machine level by the compiler. In some sense, the effects of the computed calls that are induced by the use of function pointers in C are much more critical on assembly level. When doing a *worst case execution time* (WCET) analysis on an executable file for example, it is very hard to get precise results when the targets of a computed call are not known.

1.2 Pointer Analysis

A pointer analysis tries to determine the possible targets of pointer variables statically. Consider the C program in figure 1. For this example, a pointer analysis should find out that at `P1` `a` points to `c` and `b` points to `c` or `d`. We say that {`c`} and {`c, d`} are the *points-to sets* for `a` and `b` respectively,

```
void f (int x)
{
    int *a, *b, c, d;

    a = &c;

    if (x > 0)
        b = a;
    else
        b = &d;

    /* program point P1 */
}
```

Figure 1: Example program

as illustrated in figure 2. The result of a pointer analysis is *sound* if all actual runtime targets of a pointer variable are contained in its points-to set. Accordingly {`c`} would be an unsound points-to set for `b` because `b` might also store the address of `d` at `P1`, depending on the value of the function parameter `x`. A solution is *imprecise* if a points-to set contains variables whose addresses are never stored in the pointer during an actual execution of the program. Thus, {`c, d`} would be a sound, but imprecise solution for the points-to set of `a` in this example.

In general, it is not possible to get perfectly precise and sound points-to sets because the problem is undecidable ([Lan92], [Ram94]). Therefore, a pointer analysis has to do some form of approximation, e.g. by calculating sound but imprecise points-to sets.

1.3 Requirements

The goal of this work is to develop a program analysis that can determine possible values of function pointer variables in real-world C programs. The analysis is aimed at programs from an embedded context. Therefore it is assumed in this work that some features of the C programming language do not occur in the analyzed programs because they are not used in practice, as automotive software rarely uses dynamically allocated memory (MISRA-C).

Furthermore, the analysis should work on the analyzed programs directly, without having to do a lot of program transformations beforehand. This makes it easy to map the results back to the program so they can be used by other analyses, like a WCET analysis.

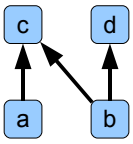

Figure 2: Points-to information

1.4 Approach

The basic idea behind the analysis presented in this work is to calculate a *points-to set* for every function pointer variable in the program. This set contains all memory locations whose address might be stored in the variable.

Since C allows indirect access to variables through their address, it is not sufficient to look only at function pointer variables. Instead, the analysis has to consider all variables that might modify a function pointer through some form of indirect memory access. As C is not a strongly typed language, it is in general not safe for an analysis to rely on type information. That is why the analysis described in the following does not use type information except for structural information, e.g. if a variable has a basic type or an aggregate type (structure or array).

The analysis information is stored in a recursive fashion so it is easy to handle function pointers that are stored within arbitrarily complex objects, like structures with arrays of function pointers.

1.5 Overview

The remainder of this work is organized as follows: chapter 2 gives an overview of program analysis in general and existing pointer analyses. A high-level description of the new analysis that is presented in this work can be found in chapter 3. Chapter 4 contains details about the implementation and the framework that was used for it. The application of the analysis to real-world programs is discussed in chapter 5. Chapter 6 gives a summary of the work and presents possible extensions to it.

2 Theoretical Foundations

This section is intended to give a short introduction to *program analysis* in general and to present some existing analyses for pointers in the C programming language. A more detailed description of the existing approaches to program analysis and the theory behind it can be found in [NNH99].

2.1 Program Analysis

The goal of a program analysis is to statically determine properties that hold for every possible execution of a given program. Since it is not possible to simulate every situation that might occur during the execution of a program, program analysis uses computable and *safe* approximations to describe the dynamic behavior of a program. An approximation is considered to be safe if and only if every possible program state is reflected in the analysis results. If the analysis is not able to find out something about the desired property, which means that everything is possible, this information also has to be present in the analysis result. This will sometimes result in *over-approximation*.

There are three main approaches to program analysis: data flow analysis, constraint based analysis and type based analysis. Data flow analyses require the control flow graph of the analyzed program and calculate the analysis information for every node in the graph. See the next section for a more detailed description. A constraint based analysis constructs a system of set constraints from a given program and then iteratively solves this system. In the basic approach, these systems do not make use of control flow information for the program, so the result is one solution of the system for the whole analyzed program. Type based analyses deduce the analysis information from a set of inference rules. This works very similar to the type checking in functional programming languages like SML.

Although all approaches in practice share many similarities, the focus of this

work will be on data flow analysis. It is thus closer looked at in the following.

2.2 Data Flow Analysis

A data flow analysis works on the control flow graph of a program, which contains a node for each statement. Edges in the graph represent the possible control flow of the program. Hence there is an edge from one node to another iff the statement corresponding to the second node can be executed immediately after the statement corresponding to the first node.

Data flow analyses compute an analysis result for each program point (each node in the control flow graph). This works as follows: the information the analysis wants to compute is represented by a data structure called the *carrier*. The carrier highly depends on the property that is analyzed. Furthermore, the analysis has to define a *transfer function* that models the effect a program statement has on the carrier. In other words, the transfer function describes the *operational semantics* of a program with respect to the carrier of the analysis.

Function calls can be handled in several ways by a data flow analysis. One way is to construct an interprocedural control flow graph, where there are additional edges for each function call from the call site of a function to the beginning of the code for the function and from the end of the function to the node after the function call. This has the effect that for each function call, the whole body of the function has to be re-analyzed.

Another way to handle function calls is the so called functional approach, which tries to calculate the effect of a function call on the carrier in order to reduce the number of times function bodies have to be re-analyzed.

To obtain a result, the analysis does a fixed point iteration on the control flow graph. When doing a *forward analysis*, the graph is traversed in the order of the actual control flow. During a *backward analysis* the order is reversed. In the beginning, some initial value is assigned to each node in the graph. Afterwards, the analysis goes over the graph and applies the transfer function to each node, combining the new result with the results from previous iterations. This is done until the result has stabilized for each node. The way the analysis selects nodes from the graph and combines old information with new information has a great impact on the performance of the analysis.

The details of how to design a data flow analysis, in particular how to make it terminating, efficient and correct (e.g. lattice theory and abstract inter-

pretation), have been omitted in this brief description. An in-depth study of all the underlying theory can be found in [NNH99] or [WS07].

2.3 Related Work

Many papers on pointer analysis have already been published, but at least for the analysis of function pointers, existing algorithms leave some things to be desired. Partly this is due to the fact that most pointer analyses were designed to be used in compilers to discover data dependences in order to be able to apply optimizing program transformations.

Pointer analyses can be partitioned into three groups:

- An *alias analysis* computes pairs of expression which might or must access the same memory location. This is the classical approach to resolve data dependences in a compiler.
- *Points-to analyses* calculate a points-to set for each pointer variable. The elements of this set represent the possible values of a pointer variable. However, although there are points-to analyses that try to handle data structures on the heap, generally they do not handle it very well.
- A *shape analysis* (e.g. [SRW02]) can not only be used to determine alias relations between memory locations, but it can also verify properties of data structures in memory, like the cyclicity of a list. Unlike the other analysis types, shape analyses are specifically designed to handle dynamically allocated memory.

Since this work describes a points-to analysis, most of the algorithms that are described below also fall into this category. An empirical study about most of these analyses can be found in [HP00].

Existing points-to analyses can be divided into flow-sensitive analyses, which use the control flow of a program for the analysis, and flow-insensitive analyses, which look at the statements of a program without considering a possible order of execution. However, there are also some hybrid approaches that combine both strategies.

The flow-insensitive analysis described by Steensgaard in [Ste96] calculates point-to sets by considering it as a typing problem. Shapiro and Horwitz tried to improve this idea [SH97].

Another flow-insensitive approach is to construct a system of set constraints over points-to sets and solve this system ([And94], [PKH07]).

Emami et al. presented a flow-sensitive analysis in [EGH94], which runs on the control flow graph that computes alias pairs but uses function inlining for every function call. If their analysis discovers a target for a call through a function pointer, the target function also gets inlined. This way, the control flow graph grows during the analysis. Obviously, the analysis does not perform very well for large programs or those with recursive functions.

The algorithm published by Wilson and Lam in [WL95] calculates points-to sets with a data flow analysis that calculates the effect on the data flow value for every function call (functional approach).

Among several other analyses, one can find an analysis in [HBCC99] that is flow-insensitive for the intraprocedural part of the analysis and flow-sensitive for the interprocedural part.

The application of pointer analysis to the construction of call graphs is studied in [CmWH99] and [MRR04].

3 Analysis Description

A high-level description of a general pointer analysis for C programs that do not use dynamic memory allocation is presented in this chapter.

3.1 Overview

To determine the possible targets for a function call through a function pointer in a given program, one needs to know all possible values this function pointer might have during a run of the program. Like any other pointer, a function pointer is a variable that stores the address of a memory location. Therefore, if a program analysis can compute the possible values of all pointer variables, it is also possible to resolve all function pointer calls with this information. Since C allows indirect memory access through pointers, any program analysis that wants to find out something about possible values of variables has to do some form of pointer or alias analysis anyway. So it seems quite natural to do a full pointer analysis on a program to resolve function pointers. That is why this chapter describes a general pointer analysis that will be used to implement the function pointer analysis described in the next chapter.

The idea behind the presented analysis is to calculate a *points-to mapping* for very program point of a given program using a data flow analysis. This mapping assigns a set of memory location to every memory locations containing a pointer variable. Those sets are also called *points-to sets* in the following. If the mapping calculated by the analysis assigns a points-to set to some memory location, this means that this location might store the address of the locations in the set.

3.2 Memory Model

In order to store information about memory locations in the points-to mapping, they have to be modelled in an abstract way. The approach chosen for this work is to preserve information about the relative position of memory locations that have to form consecutive blocks in memory during an actual execution of the program (e.g. elements of arrays and structures), whereas the model assumes nothing about the position of those blocks relative to each other. This means any variable in a program, no matter if it is of a basic or a complex type, is treated as a completely independent object in memory. A consequence of this is that some forms of pointer arithmetic cannot be expressed in this model, namely those that would leave the address range of some variable and enter that of another, as during an out-of-bound array access. Therefore this kind of operations are not allowed in the programs that serve as input for the analysis described here, although they might have a well-defined semantics in the compiler a given program was written for. Nevertheless, this seems to be a sound restriction since such programs are in general not portable because they use assumptions about the memory layout the target compiler generates.

The intuition behind the model used here is that memory accesses, examined on a low-level, usually start from some base address (like the address where an array or `struct` is located). They then move relative to this position, e.g. to select some element from a structure or to iterate through an array by adding a constant or a variable value to the base address. Since the analysis is meant to be compiler and platform independent, it does not assume anything about the absolute size of a variable. This is reasonable because a variable of type `int`, for example, might occupy 2, 4, or 8 bytes in memory, depending on the target architecture. The layout of structures in memory might also vary, depending on the padding used by the compiler. Therefore, parts of a larger data structure are described relative to the parent object. The idea behind this is that arrays always consist of elements of the same size and the position of an element within an array or struct never changes. So it is completely sound to just number the subobjects and refer to them via this numbering.

A memory location is described as a list of so called *offsets*. Those lists are constructed from the expressions in the program that might access the location. To do this, an offset, which is either a unique name or an integer constant, is assigned to each identifier in the expression. Then the offset list for the expression is calculated in a bottom-up fashion from the syntax tree of the expression. Offsets are assigned to identifiers as follows: variable and function names are mapped to themselves, whereas identifiers for structure

elements are mapped to their relative position in the structure, but without considering their size. To make this mapping work, *all* identifiers in an analyzed program have to be unique. From now on, it will be assumed that this holds for all input programs of the analysis. In an actual implementation, this can be achieved by an additional preprocessing phase of the analysis.

So, for simple expressions that only consist of a variable name, the offset list that is assigned to it only contains the variable name itself. The offset list that is assigned to an array index expression with a constant value is the list that only contains this constant value and therefore describes the relative position of this element within the array. Variables within an array index expression cannot be evaluated statically, because their value is not known in general. This is why the offset list describing such a subexpression only contains the *unknown offset* $\top$. A memory access with unknown offset is meant to span the whole parent data structure.

To get the offset list for more complex expressions, the offset lists of the subexpressions are concatenated. With this construction, it is possible to describe the memory locations which can be accessed by expressions that do not use any indirection through pointers. The construction of offset lists from expressions is shown in Figure 3.

Since an offset list only describes one consecutive block of memory, as soon as there are pointers involved it is not enough to describe the memory locations an expression might access. This is why *points-to sets* will be assigned to expressions instead of offset lists in the following. A points-to set is nothing but a set of offset lists, so for simple expressions the points-to set is just the singleton set containing the offset list described above. With the points-to mapping described in the next section, it is also possible to describe the memory locations that might be accessed by an expression containing a pointer dereference. The points-to sets of such expressions are not necessarily singletons, of course.

3.3 Points-to Mapping

The points-to mapping is a partial function that maps memory locations to points-to sets. In other words, it stores possible values for pointer variables, which is exactly what the analysis described in this document is meant to calculate.

The data structure that was chosen for the points-to mapping in the actual implementation is a recursive hash table. Basic variables like pointers or

```
struct
{
    int *p1;
    int *p2;
} s1;

struct
{
    char c;
    int a[4];
} s2;

int *arr[10];
int *p;
```

Identifier	Offset
p1	0
p2	1
s1	s1
c	0
a	1
s2	s2
arr	arr
p	p

Expression	Offset list
s1.p1	[s1, 0]
s1.p2	[s1, 1]
s1	[s1]
s2.c	[s2, 0]
s2.a	[s2, 1]
s2.a[1]	[s2, 1, 1]
s2.a[s2.c]	[s2, 1, $\top$]

Figure 3: Use of offsets to model memory locations

integers are mapped directly to a points-to set, whereas variables that are of an array or structure type are mapped to another hash table. Since the C standard allows conversion between pointer and integer variables ([C99], page 47), there is no real distinction between pointers and other variables that are large enough to store an address. To look up the points-to set for a given memory location, the concept of offset lists from the last section is used. Starting from the top-level points-to mapping, one has to check what is mapped to the head of the offset list in the mapping. If it is a points-to set, the process is finished, otherwise one has to continue with the tail of the offset list and the points-to mapping that was the result of the earlier lookup.

The points-to mapping can also be thought of as a tree. The leafs of the tree store the points-to sets and all edges are labeled with offsets. When looking up the points-to set for a memory location, the tree is traversed from the root along the edges which match the offsets in the offset list for the memory location.

To easily lookup points-to sets for arrays of pointers that are indexed with a variable, an entry for the *unknown offset* $\top$ is also stored for each of the mappings that are used to model array and `struct` variables. What is mapped to the unknown offset is a safe approximation of all entries stored in the mapping, so in general the set union of all points-to sets that are stored in the mapping. A pointer that has not been initialized is mapped to the empty points-to set, which is represented by $\bot$. Pointers that potentially point to any memory location are mapped to $\top$. This describes the points-to set which contains any possible list of offsets.

```
struct
{
    int *x;
    int *y;
} s;

int a[10];
int *z;

int function (bool flag)
{
    s.x = &a[1];
    s.y = &a[2];

    // do something with a

    if (flag)
        z = s.x;
    else
        z = s.y;

    return *z;
}
```

Figure 4: Example source code

An example of how the information computed by the analysis looks like is displayed in figure 5. It shows the control flow graph of the program presented in figure 4 and the calculated points-to mappings. The mapping which is valid after a program point is annotated at the edges leaving the respective node in the graph.

3.4 Updating the Points-to Mapping

During the transfer function of the analysis, two things have to be computed simultaneously when updating the points-to mapping:

- the memory locations which might be accessed by a given expression
- the points-to set of a given memory location

Both can be represented by a points-to set, but it is important to understand the difference. For a simple expression without some form of pointer dereference, the memory locations it might access can be derived directly by constructing its offset list and hence the points-to set representing the memory locations the expression can access would be a singleton. For expressions that employ indirection, this does not hold and therefore sets are required.

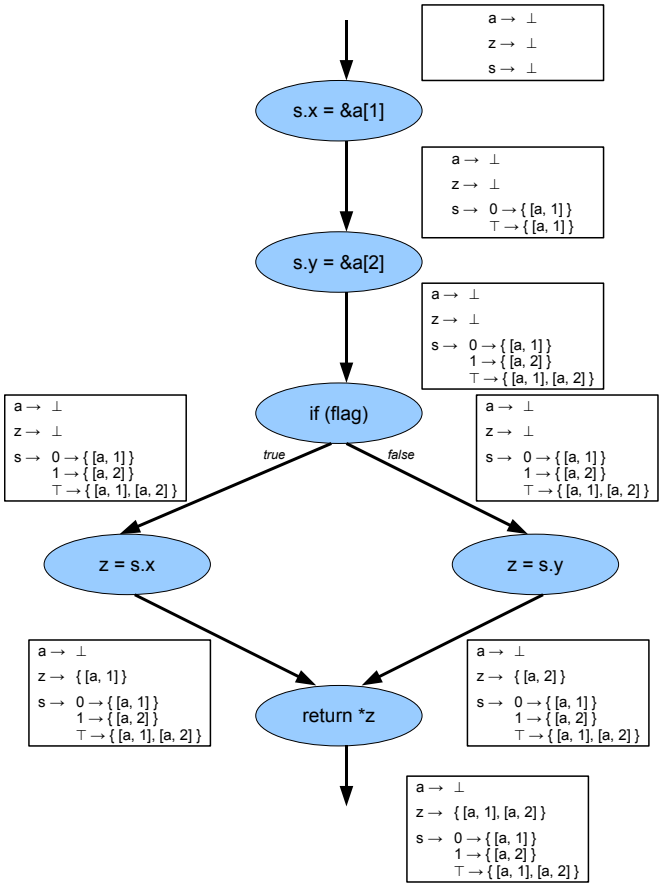

Figure 5: Control flow graph for code from figure 4 with analysis results

On the other hand, a points-to set describes the contents of a pointer variable which is the address of a memory location.

Intuitively, the memory locations which can be accessed by an expression describe the locations that will be used for read/write access, whereas the points-to set which is assigned to a memory location in the points-to mapping describes what is currently stored in these locations.

3.4.1 Assignments

Nodes in the control flow graph which contain an assignment expression modify the contents of the points-to mapping if it is an assignment between memory location whose content might be a memory address. To capture the

effect of an assignment like

```
a = b
```

where `a` and `b` represent arbitrary expressions, the first thing the analysis has to do is calculate the offset lists which describe the memory locations that can be accessed by the left and the right hand side of the assignment. Since `a` and `b` are arbitrary expressions, the memory location they describe might not be unique. For all memory locations described by the right-hand side of the assignment, the analysis looks up what is stored for those locations in the points-to mapping and builds the union of the results from these lookups. This new set has to be stored in the points-to mapping for the memory locations described by `a`. The intuition behind this is that if `b` stores an address, `a` stores the same address after the assignment.

As already stated, `a` might not describe one unique memory location. This can happen when a subexpression of `a` contains a dereference of a pointer whose points-to set is not a singleton or accesses an array with an index that is not known statically. In order to safely approximate the result of the assignment in this case, the information which is already stored in the points-to mapping for the left-hand side of the assignment must not be overwritten. Therefore the points-to set is updated with the union of the already stored points-to set and the newly calculated set from the right-hand side. This is called a *weak update* because it does not remove any previous information and only makes the points-to set grow.

On the other hand, if `a` describes a unique memory location, the assignment is guaranteed to write to this memory location. Therefore all information stored in the points-to mapping for this location can be safely replaced by the new information. Thus it is possible to do a *strong update* in the mapping and only store the newly calculated set as points-to information for the location described by `a`. It is desirable to do strong updates whenever possible because they can produce smaller points-to sets and hence provide more precise information.

Updating the points-to mapping happens by evaluating expressions in a bottom-up fashion. So in the example above, the expressions `a` and `b` are evaluated before the assignment is evaluated by the analysis. The evaluation of subexpression in turn might modify the points-to mapping. This recursive strategy takes care of modeling side effects and indirect memory accesses in an easy way.

A special case that has to be considered when updating the points-to mapping is that a pointer to an aggregate object and a pointer to the first element of the object describe the same memory location ([C99], page 47). The analysis handles this by directly updating the zero element of the points-to mapping for the aggregate object in this case, e.g. if an array is just dereferenced instead of using the subscript operator `[]` with index 0.

3.4.2 Address-of Operator

The use of the address-of operator `&` is the operation which is mainly used to access the address of a variable instead of its value. Other methods for this, like direct assignments of an array variable or a function to a pointer variable, can be reduced to an expression that uses the address-of operator or might be handled as a special case in an actual implementation of the analysis.

An application of the address-of operator is usually encountered during the evaluation of an assignment when the analysis tries to calculate the memory location described by the right-hand side of an assignment. To model the memory location described by an expression of the form

```
&e
```

a temporary variable is added to the points-to mapping. The temporary is mapped to the memory locations that can be accessed through `e`. In fact, this is a direct translation of the semantics of the address-of operator to the analysis, although the introduction of a temporary variable might seem odd at first. During a further evaluation of an expression like

```
p = &e
```

the lookup in the points-to mapping for the right-hand side of the assignment will yield the memory locations which can be accessed by `e`. So, assuming `p` and `e` are simple variables, `p` will point to `e` in the mapping and the temporary variable can be discarded afterwards.

3.4.3 Pointer Dereference

In order to determine which memory locations might be accessed by a pointer dereference expression, the information stored in the points-to mapping has to be used. So to handle the expression

```
*p
```

whereas `p` might be an arbitrary complex expression, the analysis has to determine which memory locations can be accessed by `p` and then look up the points-to sets stored in the points-to mapping for all of those locations. The union of all these sets describes the memory locations which might be accessed by the dereference expression.

3.5 Language Constructs requiring special handling

3.5.1 Pointer Arithmetic

The memory addresses stored in pointers can be manipulated directly with pointer arithmetic, but this operation is only well defined for pointers to elements of an array ([C99], page 83). Principally adding n to a pointer to the mth element of an array yields the $(n + m)$th element, subtraction is defined similarly.

Since the offsets which are used to describe the elements of an array store the relative position in the array and not the absolute size, it is fairly easy to model the effect of an arithmetic operation on the points-to set of a pointer by just adding the integer operand to the last entry of each offset list in the points-to set. If the actual value of the integer operand is not known statically, the result of the addition is not known statically either and hence will result in the unknown offset. This approach works very nicely for standard applications of pointer arithmetic, e.g. iterating through an array by pointer arithmetic, but there are special cases like casting of pointers to array elements and multidimensional arrays that require special handling.

3.5.2 Casting

Casting pointers might modify the size of the base type of a pointer. If a pointer is casted to a pointer with a smaller base type, pointer arithmetic and the dereference operator still can be applied to the pointer of the smaller base type. Hence it is possible to write data to subparts of the objects the original pointer pointed to, e.g. it is possible to access the elements of an array of pointers bitwise by casting a pointer to an element to a pointer to `char`. This has to be considered when updating points-to sets. Nevertheless it must be possible to cast back and forth between different pointer types without information loss as long as no pointer arithmetic is used.

The solution that was chosen for the presented analysis was to add a casting tag to points-to sets which are copied between pointers of different base type sizes. If pointer arithmetic is applied to a points-to set with this tag, the result is undefined ($\top$). All other operations on sets with this tag work as they do on sets without it.

3.5.3 Multidimensional Arrays

In C it is possible to construct multidimensional arrays, which are basically arrays whose elements are arrays again. Pointer arithmetic on pointers into multidimensional arrays is different from pointer arithmetic on pointers into simple arrays. For simple arrays, the pointer is not allowed to leave the array bounds, at least if it is used for a dereference operation, and the definition above implicitly assumes this. On the other hand, it is quite common to use one single pointer to iterate through all elements of a multidimensional array. Since the analysis uses recursive points-to mappings to model multidimensional arrays, using the definition for simple arrays would result in only iterating over the first subarray. To make this work for multidimensional arrays, too, the analysis must detect whenever the address of an element of a multidimensional array is taken. In this case, the offsets used for indexing into the array are modified to $\top$ for all offset lists in the points-to set of the pointer, making the pointer cover the whole multidimensional array. As a result of this, when the analysis evaluates pointer arithmetic there is no need for a special handling of pointers into multidimensional arrays. Nonetheless, standard operations (like assignments to elements) on multidimensional arrays can be represented in a precise way by calculating the offset lists as usual for expressions which do not involve pointers.

3.5.4 Unions

Unions in C offer a possibility to store many different types in the same memory block. This has to be considered by a pointer analysis because in some cases it is allowed to read data from the elements of one type that has been written there through the elements of another type. An example for this is provided in figure 6. Although this is only allowed for structures in the union that declare elements of the same type at the beginning ([C99], page 73), assuming that elements of the union are accessed only through their respective selector might yield wrong results. Hence it would be incorrect to treat a `union` like a `struct` variable.

```
union {
    struct {
        void (*callback)(int);
        char data[10];
    } s1;

    struct {
        void (*callback)(int);
        int id;
    } s2;
} u;

u.s1.callback = foo;
u.s2.id = 1;

u.s2.callback (u.s2.id);
```

Figure 6: Valid use of a union containing structures

Again, the description of memory locations by offset lists offers a very elegant solution to this problem. Since the elements of structures in the `union` that coincide must have the same size and ordering, their offset will be the same. That is why it is possible to discard the selector of the union elements from the offset list, which basically lets everything contained in the union overlap. Nevertheless, this is completely safe because for elements where mixed access to the union elements is allowed, this will provide correct result. For other accesses, the semantics of the input program for the analysis would not be well-defined, so the output of the analysis also does not have be.

To further illustrate how the analysis handles unions, with the definition from figure 6, the offset list for the expressions `u.s1.callback` and `u.s2.callback` both will be `[u, 0]`. So in the memory model of the analysis both expressions will access the same memory location, which is exactly what would happen during an actual execution of the program.

4 Implementation

The analysis described in the last section was implemented using an existing toolchain for generating static program analyzers for C programs. In addition to what was described in the last chapter, several other tasks had to be handled by the actual implementation, namely the generation of unique identifier names in the analyzed program and the construction of a mapping from those names to the offsets used by the analysis. It is also necessary to extend the control flow graph of the analyzed program during the analysis as targets for calls through function pointers are found by the analysis. Furthermore, the implementation offers the possibility to only consider nodes of the control flow graph that have an effect on function pointer variables or to resolve function pointers in an optimistic way.

The output of the analysis implementation is a complete control flow graph of the analyzed C program, at least if all function pointers could be resolved. This CFG can be reused by other analyses that are implemented in the same framework. Furthermore, the analysis implementation can create an annotation file for the AbsInt aiT Worst Case Execution Time Analyzer. For each position in the analyzed source code that is a function pointer call, this file stores which functions might be called at this program point. This provides aiT with the necessary information to compute the CFG for the compiled version of the program on assembly level, which is required to do the worst case execution time analysis.

4.1 Implementation Framework

4.1.1 Program Analyzer Generator (PAG)

The Program Analyzer Generator ([PAG]) is a tool to automatically generate data flow analyzers from a specification in a functional programming language. It was originally developed at Saarland University and is now

maintained by AbsInt Angewandte Informatik GmbH. The process of developing a data flow analysis is greatly simplified by PAG, because it provides means to easily implement the domain of an analysis, iterate the control flow graph of the analyzed program and it takes care of the fixed point computation. For each programming language one wants to generate analyzers for, it is necessary to implement a front-end for PAG that is responsible for parsing programs and generating their control flow graph as well as supplying an interface to access it.

The specification of an analysis is translated to C source code by PAG. It is also possible to directly implement parts of an analysis in C instead of generating it. The actual analyzer is created by compiling the generated code with the front-end for the analyzed language.

4.1.2 ROSE Compiler Infrastructure

The ROSE compiler framework ([ROS]) is a source-to-source infrastructure which has been developed at the Lawrence Livermore National Laboratory (LLNL). ROSE offers an easy way to parse C/C++ programs and access their abstract syntax tree (AST), e.g. for program analysis or program transformations. Like a traditional compiler, ROSE is divided into a front-end, a mid-end and a back-end. The Edison Design Group ([EDG]) front-end is used by ROSE to parse C/C++ programs, but the internal representation of the EDG front-end, which is C-style, is translated to an object-oriented representation of the AST on which ROSE works internally. The mid-end allows modifications of the AST in an easy fashion. The neat thing about the approach chosen by ROSE is that program transformations of C/C++ programs can be specified in C++ directly, so it is not necessary to learn a special transformation language to work with ROSE. Moreover, ROSE provides a back-end that can unparse the AST back to C/C++. That is why everything that was present in the original source code has to be represented in the AST, even whitespace or comments.

ROSE is not limited to C/C++, Fortran for example is also supported, but since only C is considered here, the other parts are of no interest for this work. Nevertheless, ROSE offers many interesting possibilities which go beyond what it is used for in the implementation of this function pointer analysis.

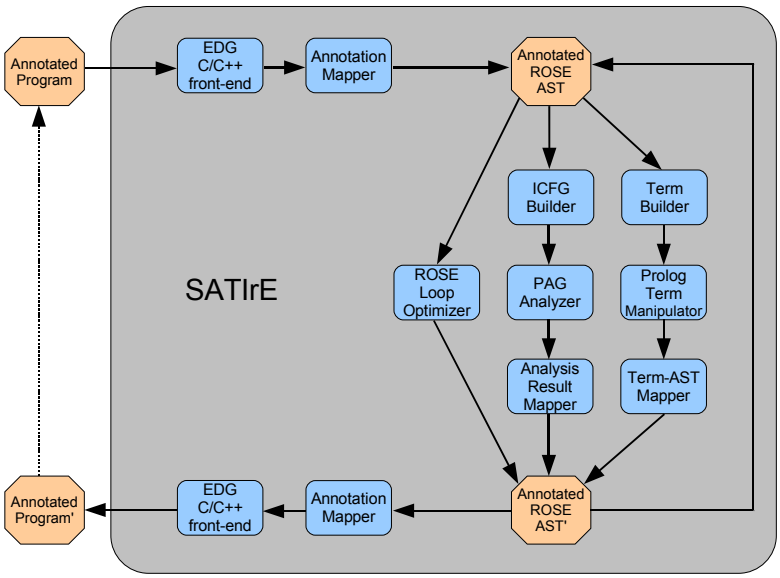

Figure 7: SATIrE Implementation overview from [Sch07]

4.1.3 Static Analysis Tool Integration Engine (SATIrE)

The Static Analysis Tool Integration Engine (SATIrE) is an architecture to combine different program analysis and transformation tools ([Sch07]) for the usage with C and C++ programs. It is being developed at the Vienna University of Technology and it heavily relies on the ROSE infrastructure.

The combination of various independent analysis techniques is achieved by transforming input programs to a common intermediate representation (IR). The input program might already contain annotations of analysis results from previous runs of different analyses which are also represented in the high-level IR. For each integrated tool, the high-level IR must be translated to the tool-specific intermediate representation before the tool can start working on the program. After a successful run, the tool-specific IR can be translated back to the common IR, which can also store the results of the run. The use of the ROSE back-end also allows it to translate the high-level IR back to source code. It is thereby possible to annotate analysis results directly to the source code of the analyzed programs. Figure 7 gives an overview of the current implementation of the SATIrE architecture.

Although SATIrE can be used for other things (like it is possible to translate C/C++ to a Prolog representation, work with the Prolog term and then

translate it back to C/C++ source code), for this work SATIrE only serves as front-end for an analysis generated with PAG. In this role, SATIrE uses ROSE to get the AST for the program that is meant to be analyzed and translates the AST to an interprocedural control flow graph that can be accessed by PAG. This involves some non-trivial program transformation because some basic operations of C, like parameter passing, are not directly supported by PAG. These transformations are also achieved by using the ROSE infrastructure.

4.2 Required Extensions

4.2.1 Variable Renaming

The general pointer analysis described in chapter 3 requires the possibility to assign a non-ambiguous offset to each identifier in the analyzed program. Since identifiers are not guaranteed to be unique, some form of variable renaming has to be done before the analysis can start. Although SATIrE offers the possibility to number identifiers in a program according to their scope, so that variables with the same name but a different scope can be distinguished, this is not good enough to construct the offset mapping for the analysis. This is the case because names of `struct` and `union` elements have to be distinguished from ordinary variable names. Therefore a renaming phase was added to the analysis implementation before the control flow graph is constructed from the ROSE AST. This phase not only generates a unique name for each identifier in the analyzed program, but it also calculates a mapping that assigns an offset to those unique identifiers. Since this mapping is calculated in advance, the analysis can just use it when constructing offset lists for expressions during calculations in the transfer function. Fortunately, ROSE makes variable renaming and constructing the offset mapping relatively unproblematic because it provides an easy to use interface to traverse the AST and match certain language constructs.

4.2.2 Control Flow Graph Refinement

Without doing a function pointer analysis, the control flow graph of a program that uses function pointers must be incomplete because the functions that are called by a call through a function pointer are not known. Therefore the control flow graph, as it is generated by SATIrE for such programs, lacks the call and return edges for those call sites in the program. This is prob-

lematic because a data flow analysis on an incomplete CFG might produce incorrect results because it misses realizable control flow paths. So obviously it is necessary to add the appropriate call and return edges to the control flow graph whenever the analysis discovers a target for a function pointer call. Since modification of the CFG is not supported by PAG during the analysis, the approach that was chosen for the implementation is to let the analysis run to completion on the incomplete graph. Afterwards the call and return edges are added for those functions that are in the points-to set of the expressions that form a function pointer call and the analysis is restarted. This process is repeated until no new targets are found.

The stepwise extension of the CFG might add too many call and return edges because the analysis information at a function pointer call site can change from one iteration of the analysis to another. This is very unlikely though, because it would require that the value of a function pointer is changed in a function that is called through a function pointer, and even then the edges are never removed. Nevertheless the results of the analysis are still sound, although they might be less precise. Additional edges in the CFG only mean that more unrealizable paths are considered by the analysis, but unrealizable paths might be there anyway because of the approximative nature of data flow analysis. In the final iteration of the analysis no new targets have been found. Hence, the CFG the analysis works with is a superset of the minimal correct CFG of the analyzed program if and only if all function pointer calls could be resolved in previous iterations. Therefore the analysis results are correct, if it was possible to find a target for each function pointer call site. Otherwise the current CFG might still be a subset of the actual CFG and thus the analysis results are not guaranteed to be correct.

4.2.3 External Functions

Quite often it is not possible to analyze all functions that are called by a program because their source code is not available. This happens almost exclusively for functions from program libraries. In many cases, these functions will not affect the information calculated by the pointer analysis. Nevertheless this is possible and must be considered to get correct results. Therefore the option was added to annotate the effect an external function has on the points-to information, of which there are three possible:

- *None*, which means the external function call does not influence any pointers of the program.

- *Local destructive*, which has the effect that after the call the return value and all memory locations reachable through the arguments of the function are assigned to $\top$ in the points-to mapping.
- *Global destructive*, meaning that all information gathered so far becomes invalid after the function call.

The annotations are stored independently from the analyzed program and hence must be provided only once for each library used by any number of analyzed programs.

4.3 Further Extensions

4.3.1 Program Slicing

To improve the run-time of the function pointer analysis, an additional analysis phase was added that determines at which program points the values of function pointers might be modified directly or through aliases. In the actual pointer analysis, the transfer function is only applied to those nodes.

The preprocessing analysis to determine nodes in the CFG that are *interesting* for the function pointer analysis works by looking for interesting identifiers in the expressions of a program. In principal this works by doing a backward data flow analysis which applies the following rules to the CFG until a fixed point is reached:

- all identifiers in an expression that is a function pointer call are interesting
- an identifier is interesting if it occurs within an expression that contains an identifier which is known to be interesting
- a node in the CFG which contains an expression with an interesting identifier is interesting

Nodes in the CFG are only marked instead of actually removing nodes from the graph because with this approach it is possible to reuse the CFG for other analyses without recalculating it. This way, despite slicing, it is still possible to let subsequent analyses work with the complete CFG created by the function pointer analysis. An exhaustive overview of the different analysis stages is shown in figure 8.

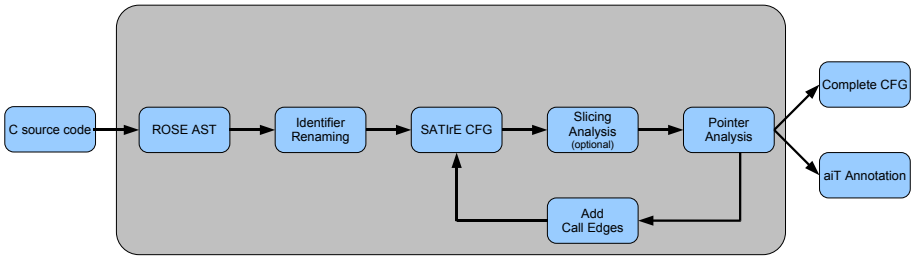

Figure 8: Analysis stages

4.3.2 Optimistic Analysis

If a completely safe pointer analysis reaches a program point at which a value is assigned to a memory location the analysis does not know, it must discard all information it has gathered so far. This is due to the fact that such an assignment could change the content of any memory cell and therefore none of the calculated points-to sets can be guaranteed to be safe. This case might occur for various reasons, e.g. because a pointer is dereferenced that was manipulated by a complicated form of pointer arithmetic or an array of pointers is indexed with a variable but only some of the pointers were initialized previously.

Such effects might influence the analysis results quite considerably, especially when global arrays of function pointers are used. That is why an optional *optimistic* mode was added to the analysis, which differs in the following points from the standard analysis:

- For global variables that are declared with an initializer, each time the analysis enters a function body it checks if this information has been overwritten with $\top$. In this case, the information from the initializer is added to the points-to mapping again.
- The unknown offset in a points-to mapping for an array or a `struct` variable is never mapped to $\top$. Instead, the points-to set of the unknown offset only collects entries of the points-to set for the other entries in the mapping which are not mapped to $\top$. This makes it possible to handle arrays of pointers in which some of the entries are set to `NULL`.
- When resolving targets of calls through function pointers, the analysis also uses targets which occur while the analysis is still running and

the fixed point has not yet been reached. Also when looking up what is stored in the points-to mapping for the points-to set of a function pointer call expression, the information stored for the unknown offsets is used if the exact items in the offset list are mapped to $\top$.

Obviously this approach is not guaranteed to provide correct results in general. Nonetheless, under the restriction that no complicated pointer arithmetic is used to modify function pointer variables and uninitialized memory is never used, this optimistic strategy seems to work quite well in practice.

5 Results

The analysis implementation was tested with a set of programs that use function pointers and whose source code is publicly available. All programs were analyzed with the different analysis modes offered by the implementation. The test programs and the analysis results are presented and evaluated in the following sections. Some basic information about the test programs and the results for the standard analysis mode which does not use any optimistic assumptions can be found in figure 9. The results for the various other analysis modes are displayed in figure 10. Columns labeled *Resolved* contain the number of function pointer calls for which the analysis was able to determine which functions are called. The *Targets* column lists the average size of the points-to set of a function pointer used for a call.

5.1 Test Programs

5.1.1 grep

The well-known Unix tool `grep` uses function pointers in a way that shows the advantages the analysis described in this work offers compared to other

		Characteristics		Standard Results	
	Version	Lines	Ind. Calls	Resolved	Targets
diction	0.7	2037	3	0	0
grep	2.0	12417	3	0	0
gzip	1.2.4	8163	4	1	1
sim6890	0.1	3290	97	2	6

Figure 9: Analysis results for the test suite using the standard mode of the analysis implementation

	Optimistic Results		Slicing Results		Optimistic + Slicing	
	Resolved	Targets	Resolved	Targets	Resolved	Targets
diction	1	1	3	1	3	1
grep	2	12.5	0	0	3	9
gzip	1	1	2	1	4	3.5
sim6890	97	10	2	6	97	10

Figure 10: Analysis results for the test suite using the slicing and optimistic mode of the analysis implementation

analyses, but also illustrates the problems these kind of analyses have to handle when they are applied to real-world programs.

The modes of operation that are offered by `grep` are stored in a global array of structures which contains function pointers to the functions that are appropriate for the respective mode. According to the mode that is selected at the startup of the program, global function pointers that determine the behavior of `grep` are set during the initialization phase in the function `setmatcher()`. The parts of the `grep` source code responsible for this are shown in figure 11.

Since the global function pointer variables are initialized from distinct structure elements within the global array of structures, an analysis that does not compute points-to information for the elements of objects that have an aggregate type will calculate points-to sets that are too large. In this case, this would result in equal points-to sets for the global variables `compile` and `execute`. Hence, this would result in an interprocedural control flow graph which is too large if this information was used to add edges for function pointer calls to the graph.

On the other hand, the use of a recursive points-to mapping to store the points-to information for objects that have an aggregate type can handle this application of function pointers. What is problematic is that the initialization of the global variables takes place within a loop. Since we know nothing about the loop bounds statically, it has to be assumed that every element of the array can be accessed, including the last element which is initialized with zeroes in this example. As a pointer with value zero has no well-defined target, the analysis conservatively assumes that the target might be anything. Hence the calls through the pointers `compile` and `execute` (which are omitted in figure 11) can not be resolved by the analysis implementation, at least not in its standard mode. Nevertheless, the optimistic modes of the analysis described in section 4.3 can resolve these calls because it assumes that the

```
/* grep.h */
extern struct matcher
{
  char *name;
  void (*compile)(char *, size_t);
  char *(*execute)(char *, size_t, char **);
} matchers[] = {
  { "default", Gcompile, EGexecute },
  { "grep", Gcompile, EGexecute },
  { "ggrep", Gcompile, EGexecute },
  { "egrep", Ecompile, EGexecute },
  { "posix-egrep", Ecompile, EGexecute },
  { "gegrep", Ecompile, EGexecute },
  { "fgrep", Fcompile, Fexecute },
  { "gfgrep", Fcompile, Fexecute },
  { 0, 0, 0 },
};
...

/* grep.c */
static void (*compile)();
static char *(*execute)();

int setmatcher(char *name)
{
  int i;

  for (i = 0; matchers[i].name; ++i)
    if (strcmp(name, matchers[i].name) == 0)
      {
        compile = matchers[i].compile;
        execute = matchers[i].execute;
        return 1;
      }
  return 0;
}
...
```

Figure 11: Parts of the grep source code

accesses within the array only go to elements that store valid information. For the example shown, this is completely safe.

5.1.2 diction

GNU diction offers two command line tools, `diction` and `style`, that can check text files for diction and language style. These programs share parts of their source code. The use of a function pointer occurs for the call of a function that is responsible for parsing sentences in the input file. The parameters of the function contain a function pointer to the function that is responsible to do the actual processing of the sentence. For `diction`,

this pointer always stores the address of a single function. The analysis implementation is able to detect this and hence can construct the complete and minimal interprocedural control flow graph for `diction`.

5.1.3 gzip

The file compression tool `gzip` uses a global function pointer variable to decide how input files are processed, meaning which compression algorithm is used. The initialization and the application of this function pointer take place in different functions called by the `main` function of the program. Therefore the results of the analysis in figure 9 seem to be quite imprecise as the analysis is not able to resolve all of the function pointer calls. The explanation for this is that `grep` makes use of Unix signal handlers. Hence the analysis assumes that the values of the function pointers might be changed at any time by the signal handlers and therefore cannot resolve the function pointer calls. In spite of that the function pointer can be resolved if the analysis uses some optimistic assumptions.

5.1.4 sim6809

`sim6809` is a simulator for the Motorola 6809 microprocessor developed by Jérôme Thoen ([Tho98]). A large array of function pointers is used in this simulator to select the function which is used to simulate the current instruction by (almost) directly indexing into this array with the numeric value of the instruction. As the binary representation of 6809 instruction set does not seem to be dense, this array contains many entries with value zero and hence the same problems as for the analysis of `grep` arise. Nevertheless the analysis implementation is able to resolve the calls under the assumptions that those `NULL` pointers are never used for a call.

5.1.5 Automotive software

An attempt was made to test the implementation with several large real-world automotive software projects, but this was not successful for several reasons. At first, it is not trivial to analyze large pieces of software at all since the analysis framework has to access the whole source code at once. This requires manual examination of the project hierarchy because the source code is usually spread over several directories and existing project management

files (e.g. makefiles) cannot be reused for the analysis because those existing files are mostly used to compile only parts of the whole project.

Another problem that occurred was that some of the programs used special language keywords because they were developed using compilers for target architectures with a small word size, but these keywords were not recognized by the analysis framework. These problems could by solved with manual intervention, too.

Currently the analysis framework also still seems to have problems with some constructs of the C programming language (e.g. valid source lets it crash). Understanding these crashes is extremely difficult because there is no precise error message, only the information that something is wrong in the abstract syntax tree. Hence it is just not feasible for larger programs to manually modify the pieces of code that are responsible for the crash because it is impossible to find them. For this reason it was not possible to apply the analysis to real automotive code, although it was available.

5.2 Evaluation

The results of the application of the analysis to real-world programs seem disappointing at first because the analysis is not able to resolve a significant number of function pointer calls without further assumptions. On the other hand, the results that were achieved using the optimistic extensions to the analysis described in section 4.3 are very good. A manual inspection of the test programs convey the impression that these results are pretty close to an optimal solution, although this cannot be guaranteed in general. Nonetheless, the slightly optimistic approach seems sufficient from a practical point of view because there is just no rational reason to modify the values of function pointers with complicated operations like excessive pointer arithmetic. For the construction of a complete control flow graph, or at least a more precise graph than one that discards all function pointer calls, it does not seem relevant that there might be the possibility that a function pointer with value zero might be used for a call. This is a matter of program correctness which is not the primary focus of this analysis. Hence the standard version is too conservative to provide usable information, while the optimistic assumptions are weak enough to hold for real-world programs.

Using a recursive points-to mapping to model the contents of aggregate objects in memory provides more precise results than a model that does not consider the subelements of complex data structures. The additional expressiveness of this model in relation to previous works is depicted in figure 12

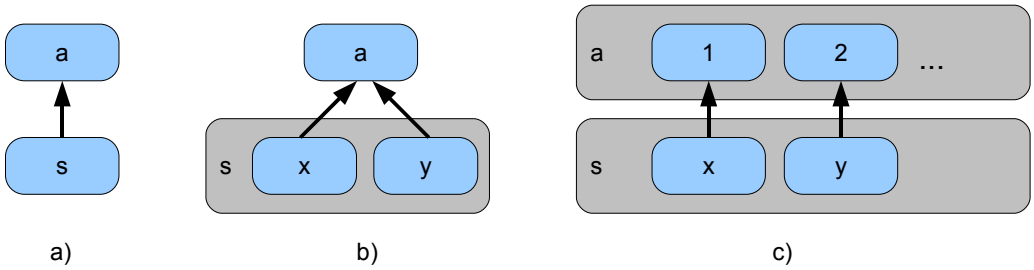

Figure 12: Better precision compared to other analyses
a) Aggregate objects treated like basic objects
b) Only calculate points-to sets for `struct` (e.g. like [PKH07])
c) Recursive points-to mapping

using the source code example from figure 4. This approach might be useful for other analyses as well, for example a flow-insensitive pointer analysis.

Flow-insensitive analyses are used more often in the literature and the results presented here seem to support the view that this method is sufficient. The reason for this seems to be that a flow-insensitive analysis can be conservative without too much loss of precision because it is less precise than a flow-sensitive analysis in the first place. The effect of the conservative approach on the precision of the result is much bigger in the flow-sensitive case because a flow-sensitive analysis must discard all its information whenever there is a memory access that cannot be resolved. Hence the flow-sensitive approach seems to deliver less precise results for a pointer analysis in practice although one would expect the opposite.

In its current state, the framework used for the analysis implementation makes it rather difficult to analyze programs without user intervention. That is why it does not seem feasible at the moment to use the analysis for the automatic extension of control flow graphs for programs or the automatic creation of annotations for analysis tools that do not depend on the source code because an user action would be required anyway.

6 Conclusion

6.1 Summary

This work described a flow-sensitive function pointer analysis for C programs. The intended area of application for this analysis is automotive software. This kind of software makes extensive use of function pointers but rarely uses dynamic memory allocation. Therefore the analysis was designed to derive as much information as possible about function pointers on the stack without considering the possibilities and problems that can arise when dynamically allocated memory is used.

The new concept presented in this thesis is the use of a recursive points-to mapping to store the calculated information about pointers. This approach makes it possible to model the contents of the stack in a precise abstract way. It is an extension of the ideas used in [WL95] but it is more precise and less platform dependent.

Results for the application of the analysis to real-world programs are twofold. Storing pointer information in a recursive mapping seems to be a very good method to store precise information very easily. On the other hand, the existing implementation of the analysis does not provide sufficiently precise results without further optimistic assumptions about the analyzed programs. Furthermore, the framework used for the implementation is not yet able to handle all realistic C programs. Nonetheless the results are promising and it is probably not too hard to solve the remaining problems.

6.2 Outlook

An implementation of the analysis would probably be able to calculate much better information about array elements if more precise information about accesses to the elements were available. It thus seems like a natural extension to place some form of *constant propagation* or *interval analysis* in front of the

pointer analysis to calculate information about the values of integer variables and use this knowledge to evaluate index expressions of arrays more precisely. As the results of the pointer analysis might affect the results of the value analysis, one would have to iterate this process until the result stabilizes.

The results that were achieved with the analysis implementation indicate that some optimistic assumptions are necessary to get useful results. In its current state these assumptions are not optimal because they are not guaranteed to hold in any case. Further research into this direction could provide better or safer results, e.g. by adding additional analyses to guarantee that things that are assumed not to happen really never happen.

Modelling points-to information with a recursive mapping and the description of memory locations with an offset list offers an advantage over the methods that have been used so far. The application of these methods to existing flow-insensitive pointer analyses might improve the results of these analyses considerably and therefore should be investigated.

Only simple forms of dynamic memory allocation can be handled by the presented analysis because it was designed for programs that do not use dynamically allocated memory. The use of cyclic data structures might eventually cause divergence of the analysis. It would be a valuable extension to the analysis if it was possible to analyze function pointers in dynamic data structures or at least detect and avoid a possible divergence.

References

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[C99] C99. *JTC 1/SC 22/WG 14 - ISO/IEC 9899:1999 - Programming Languages - C*. American National Standards Institute, 11 West 42nd Street, New York, New York 10036.

[CmWH99] Ben-Chung Cheng and Wen mei W. Hwu. An empirical study of function pointers using SPEC benchmarks. In *Languages and Compilers for Parallel Computing*, pages 490–493, 1999.

[EDG] Edison design group. http://www.edg.com.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1:323–337, 1992.

[MRR04] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for C programs with function pointers. *Automated Software Engg.*, 11(1):7–26, 2004.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[PAG] PAG - The Program Analyzer Generator - user's manual. AbsInt Angewandte Informatik GmbH. http://www.absint.com/pag.

[PKH07] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[ROS] LLNL-ROSE. http://www.rosecompiler.org.

[Sch07] Markus Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions, 2007. http://www.complang.tuwien.ac.at/markus.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[Ste96] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Computational Complexity*, pages 136–150, 1996.

[Tho98] Jérôme Thoen. Sim6809, Motorola 6809 simulator, 1998. http://membres.lycos.fr/jth/6809.html, as of August 10, 2008.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[WS07] Reinhard Wilhelm and Helmut Seidl. *Übersetzerbau: Programmanalyse und Optimierung*, March 2007. Draft, handed out in lecture "Program Analysis and Transformation", Saarland University, summer term 2007.