Bachelor thesis
Computing Science

Radboud University

An implementation of Andersen-style pointer analysis for the x86 `mov` instruction

Author:
Charlotte Leuverink
s1009955

Supervisor/First assessor:
dr. F. Verbeek (Freek)
freek@vt.edu

Second assessor:
dr. ir. E. Poll (Erik)
e.poll@cs.ru.nl

June 26, 2020

Abstract

In the past decades, a great number of studies have proposed pointer analyses for source code written in high-level languages. Pointer analysis, sometimes referred to as points-to analysis, is commonly used in compilers for optimization or detecting security vulnerabilities. However, users often have no access to the (high-level) source code. Consequently, researchers have recently started advocating pointer analysis for lower-level code. When aiming to detect security vulnerabilities or for tasks such as binary re-optimization, there exists a need for analyzing lower-level code. In fact, in some cases, pointer analysis on assembly-level code can yield more accurate results than analyzing the higher-level equivalent [14]. Despite the many advantages of analyzing lower-level code, there is currently no tool available that allows one to perform pointer analysis on low-level code such as assembly or binary executables.

We present a prototype for assembly-level pointer analysis. Since the x86 `mov` instruction is Turing complete, the extensive x86 instruction set can be reduced to this one instruction. For this reason, we have designed a pointer analysis for the x86 `mov` instruction only, providing a proof of concept for assembly-level pointer analysis in general. Furthermore, our open-source implementation makes us the first to publish a tool for performing an inclusion-based pointer analysis on lower-level code.

Contents

Chapter 1

Introduction

Pointer analysis, also referred to as points-to analysis, is a static analysis that determines the set of memory locations a pointer variable can point to during runtime. This type of analysis is widely used in compilers to optimize code and to statically detect security vulnerabilities.

We present a prototype implementation of pointer analysis for lower-level code. To be specific, our tool performs pointer analysis on assembly programs consisting of only the x86 `mov` instruction. It has been proven by Stephen Dolan that the `mov` instruction is Turing complete [51]. This implies that the whole x86 instruction set can be reduced to this single instruction. Hence, an implementation of pointer analysis for the `mov` instruction functions as a proof of concept for assembly-level pointer analysis in general. Our implementation is based on Andersen's subset-based algorithm [49]. This algorithm provides relatively precise pointer information, whilst its flow-insensitivity makes it scalable as well.

Most existing pointer analyses are performed on high-level programs written in Java or C. However, in recent years, there has been a growing need for tools that analyze lower-level code. Pointer analysis on lower-level code can even yield more precise results than analysis of higher-level code [14]. Whereas some algorithms for lower-level pointer analysis have been proposed [14, 9, 11, 3]. There seems to be a general trend in the research field of pointer analysis where implementations of proposed algorithms are not available to the public. This makes that, to the best of our knowledge, there is currently no tool available that allows us to execute pointer analysis for lower-level programs.

The x86 instruction set for assembly consists of a great number of instructions. Still, no formal definition of its semantics exists. Besides, assembly lacks the concept of variables and type-information. This makes it challenging to statically determine which variables are pointers. Hence, a pointer analysis for assembly is forced to sacrifice some precision by overestimating the memory locations that are considered pointer variables.

In Chapter 2 we will elaborate on the theory behind pointer analysis. Chapter 3 provides a discussion on the current state-of-the-art pointer analyses. In Chapter 4 we extensively discuss the challenges associated with assembly-level analysis. In Chapter 5 we will explain our prototype implementation and its results. Finally, Chapter 6 consists of the conclusions and recommendations for future research.

Chapter 2

Background

2.1 What is pointer analysis?

Pointer analysis is a static code analysis technique that determines what memory locations or objects a pointer variable or expression can refer to. Because it is a static technique, the code is analyzed without executing it. The result of a pointer analysis is for each variable and expression, the set of objects it can point to during runtime. This result is represented as a set of 'points-to'-relations. For this reason, pointer analysis is sometimes referred to as 'points-to analysis' [20, 33]. The precise solution of a pointer analysis of the code in Listing 2.1 would be that the pointer variable `p` points to `x`. Throughout this work, we will use the notation `pts(p)={x}` to denote this type of points-to information.

```
1 int x;
2 int *p = &x;
```

Listing 2.1: C code with the use of pointers

2.1.1 Alias Analysis

A specific form of pointer analysis is alias analysis. It determines which variables or expressions can alias, meaning that they point to the same memory location. For an alias analysis, the result is all sets of pointer expressions that alias each other. The precise solution of an alias analysis of the example in Listing 2.1 would be that `x` and `*p` alias. This can be denoted as the alias set `<*p,x>`. Note that this alias set yields the same information as the points-to relation mentioned before. This is why the term alias analysis and pointer analysis are often used interchangeably. Accessing pointer addresses directly using the reference operator `'&'`, as done in Listing 2.1, is very common in the programming languages C and C++, but this is not possible in every language. Still, aliasing can occur in other languages, for example by accessing a specific element in an array or by referencing objects, which is common in Java. In Listing 2.2, we see an example in which aliasing can occur due to an array. Because `i` equals 3, `a[i]` and `a[3]` will alias and the value `12` will be overwritten by `13`.

```
1 int i = 3;
2 int a[10];
3 a[3] = 12;
4 a[i] = 13;
```

Listing 2.2: Pointer example in C

2.2 What is pointer analysis used for?

In general, pointer analysis is not used as a stand-alone task. The set of points-to relations resulting from a pointer analysis is usually the input for another client analysis [15]. This client analysis is often an interprocedural analysis of the whole program. We will discuss the most important uses of pointer analysis.

Optimization

Pointer analysis is an essential part of the optimization process in a compiler [8, 1]. A pointer analysis can detect ambiguous memory references to form a more accurate view of program behavior [15, 13]. For instance, a pointer analysis can be used to create a definition-use chain, which lists for each definition of a variable, all uses of that definition. This can be of use for minimizing the number of operations and memory accesses by detecting dead code [26]. Listing 2.3 shows us an example of code that can be optimized by performing a pointer analysis. We might want to write `y=1` instead of first `x=1` and then `y=x`. But as long as we are unsure if `p` is pointing to `x`, we cannot optimize this piece of code. A pointer analysis on the whole program might be able to determine that `p` does not point to `x`.

```
1 x = 1;
2 *p = 12;
3 y = x;
```

Listing 2.3: Code that can be optimized by a pointer analysis

Detecting security vulnerabilities

Another important purpose of pointer analysis is detecting security vulnerabilities in code [26, 15, 13, 42, 43]. For instance, it can be used for detecting buffer overflow or format string vulnerabilities. Especially for detecting security vulnerabilities, a static analysis is appropriate because errors can be detected early in the development process whereas fixing errors in later stages can be very expensive. The code in Listing 2.4 shows an example of a buffer overflow vulnerability. In C, array references are defined as pointers (`buff[n]` is defined as `*(buff + n)`). Therefore a pointer analysis can determine that when executing this code, `buffer[3]` points to `input[3]`. Because we know that the size of `buffer` is `3`, we know that `buffer[3]` is outside of our buffer and we should not write to this address.

```
1 void copyString(char *input) {
2     char buffer[3];
3     for (int i=0; i<=3; i++)
4         buffer[i] = input[i];
5     }
6 }
```

Listing 2.4: Code containing a buffer overflow vulnerability

2.3 How is a pointer analysis performed?

Pointer analysis is generally undecidable [24, 29]. Hence, pointer analysis algorithms try to approximate the outcome of an analysis. In this section, we will discuss in which ways pointer analysis can be performed. We discuss the dimensions that can affect accuracy and efficiency and we elaborate on the most fundamental algorithms for pointer analysis.

2.3.1 Must and May analysis

A distinction is made between MUST and MAY points-to sets [25]. The former over-approximates actual program behavior while the latter reports points-to relations only when they are guaranteed to hold. Generally, pointer analysis is a MAY-analysis because MUST-analysis is more expensive [33].

Soundness

A sound MUST pointer analysis will return **only** those points-to relations that will definitely hold in each possible execution of the program [23]. This might yield an under-approximation of all the MUST point-to relations that hold. A MUST pointer analysis returning no points-to sets is still sound but very imprecise.

A sound MAY analysis reports **at least** all points-to relations that may occur. The drawback of this overestimation is that besides all true relations, some superfluous relations are returned that might be false [27]. For a MAY analysis, this means that every true MAY points-to relation is returned, but some extra points-to relations might be returned that will never occur in practice. In fact, a MAY analysis reporting that all variables point to all other variables is perfectly sound but too imprecise.

Thus, soundness for a MUST analysis means avoiding spurious inferences, whereas soundness for a MAY analysis means not missing true inferences [33]. The precision of an analysis is the degree to which it avoids such spurious results [27]. Pointer analyses are usually sound, so for the results to be useful, the results must be relatively precise as well.

An interesting result can be obtained by negating the result of a MAY pointer analysis (i.e., taking the complement of its output). This would result in a MUST-NOT point-to analysis [33]. However, one must be careful when drawing conclusions. These MUST-NOT point-to relations are true, but they are still an under-estimation. To be more precise, there might be more MUST-NOT point-to relations that are not determined because the MAY points-to relations overestimate the locations a pointer may refer to.

2.3.2 Dimensions affecting efficiency and accuracy

Because pointer analysis is undecidable, algorithms need to make a trade-off between efficiency and accuracy. There are several dimensions that can be considered in this trade-off, namely flow-sensitivity, path-sensitivity, context-sensitivity and field-sensitivity [31, 20]. In this section, we will explain the most important dimensions and the effect they have on the precision and efficiency of a pointer analysis algorithm. For a thorough exploration of existing pointer analyses and how they address these dimensions, we refer the reader to Chapter 3 on related work.

Flow-sensitivity

In a flow-sensitive pointer analysis, the possible pointer values are computed after each statement, that is after each assignment, function call or if-else statement. Whereas a flow-insensitive pointer analysis computes the set of values a pointer variable will have throughout the execution of the whole program. In flow-insensitive analysis, the order of statements is not important e.g., analysis of two statements S_1;S_2; will be the same as S_2;S_1;. Performing a flow-insensitive pointer analysis on the code example in Listing 2.5, would yield that `p` points to `a` and `b` during the execution of the program. A flow-sensitive pointer analysis would produce the more specific result that in program-point (4), `p` points to `a` and in program-point (5), `p` points to `b`.

```
1 int a;
2 int b;
3 int* p;
4 p = &a;
5 p = &b;
```

Listing 2.5: Example of flow-sensitivity

This makes flow-sensitive analysis usually more precise but less efficient than flow-insensitive analysis. Flow-sensitive pointer analysis is (traditionally) too expensive to perform for whole programs [49]. This is why whole-program analyses typically use flow-insensitive pointer analyses, although more recent research has shown certain optimizations that make flow-sensitive analysis possible for large programs as well [16].

In a flow-insensitive analysis, when a pointer variable is reassigned with a new value many times throughout the program, the points-to set of this variable can become very large. Accordingly, partial flow-sensitive algorithms exist, where a flow-insensitive approach is used to compute pointer values of a smaller piece of the program, for example within a method.

Path-sensitivity

A second dimension is path-sensitivity. A path-sensitive analysis considers the fact that paths are exclusive [40]. This implies that only one of the multiple paths can be chosen. When we take a look at Listing 2.6 for example, we see that a flow-insensitive analysis would conclude that `p` may point to either `a`, `b`, `c` or `d`. This result is sound but very imprecise. A path-sensitive analysis produces the result that `p` may point to `b`, `c`, or `d`, whereas a path-sensitive analysis would consider the fact that only one of two paths can be chosen and would result in the more precise solution that `p` may only point to `c` or `d`.

```
1  int a;
2  int b;
3  int c;
4  int* p;
5  p = &a
6  p = &b;
7  if (s){
8      p = &c;
9  }
10 else {
11     p = &d;
12 }
```

Listing 2.6: Example of path-sensitivity

Context-sensitivity

Another dimension that affects a pointer analysis' efficiency and precision is context-sensitivity. In a context-sensitive analysis, the caller of a function is considered when calculating results [33]. Consider the code example in Listing 2.7. The function `id` is called two times. Once with constant `1` as input and once with `2`. A context-insensitive analysis concludes that the function `id` might return either `1` or `2`. Hence, it will report that `c` can have values `2`, `3`, or `4`. A context-sensitive analysis, on the other hand, would distinguish between `id(1)` and `id(2)`. It yields the more precise solution that `c` is constant and must equal `3`. Note that this is also an example in which a pointer analysis can help the compiler to optimize the code by replacing `a+b` by `3`.

```
1  int a = id(1);
2  int b = id(2);
3  int c = a + b;
4
5  int id(int x){
6      return x;
7  }
```

Listing 2.7: Example of context-sensitivity

Field-sensitivity

Field-sensitivity means that the individual elements of an aggregate are distinguished, rather than regarding the aggregate as a whole. In general, an aggregate is a piece of data composed of smaller pieces that form one unit. We see this in object-oriented programming, where a class can contain multiple other classes. In such object-oriented programming languages like Java and C++, a class would be an aggregate. Similarly, in the programming language C a struct data type would be an aggregate since it is one piece of data that consists of multiple values. Other examples of aggregates are lists and arrays.

In a pointer analysis, these type of structures can be modeled in two different ways. Either the entire structure as one single object or each element in the structure as an individual memory location. The former is called field-insensitive while the latter is referred to as field-sensitive. In a field-sensitive analysis of a C program, each field of a struct is treated as a separate variable. That is why the concept of field-sensitivity is sometimes referred to as struct-modeling [21]. Since field-insensitivity simplifies the data structure, a field-insensitive analysis will gain efficiency but lose accuracy.

A third option is a field-based analysis, which distinguishes fields but only identifies points-to relations by the heap object's type and not its full identity. For instance, struct objects of the same name are merged but the fields are kept separate unlike in a field-insensitive analysis [33, 18]. We will discuss the resulting points-to sets of these different approaches to field-sensitivity the program snippet in Listing 2.8 to further clarify their differences.

```
1 struct point {
2     int x;
3     int y;
4 }
5 struct point p1, p2;
6 int * a, b, c;
7 p1.x = &a;
8 p1.y = &b;
9 p2.x = &c;
```

Listing 2.8: Example of field-sensitivity

A field-sensitive analysis would yield the precise result that `pts(p1.x)={a}`, `pts(p1.y)={b}` and `pts(p2.x)={c}`. A field-insensitive analysis would not distinguish between the different fields of the struct. It would, therefore, conclude that `pts(p1)={a, b}` and `pts(p2)={c}`. On the other hand, a field-based analysis distinguishes between the fields of the struct but is not able to tell apart different instances of the struct and would therefore yield `pts(x)={a,c}` and `pts(y)={b}`.

2.3.3 Fundamental pointer analyses

Although plenty of research has been done and many algorithms have been proposed, two pointer analyses remain as being the most fundamental and well-known approaches: Andersen's [49] and Steensgaard's [37]. Many pointer analyses base their algorithm on one of these fundamental algorithms. For a more elaborate description of how these algorithms are used as a basis for new algorithms, we refer to Chapter 3.

Steensgaard-style pointer analysis is known as unification-based and uses equality constraints, whereas Andersen's approach is inclusion-based and uses subset constraints. Both algorithms are flow-insensitive and context-insensitive and produce MAY points-to relations. Steensgaard's algorithm is less precise than Andersen's. However, its higher precision makes Andersen's approach less scalable [32]. Although Andersen's approach is more precise, both analyses yield sound results. In this section, we will explain Andersen's algorithm into more detail. For a more detailed explanation of Steensgaard's algorithm, including many examples and a comparison with Andersen's algorithm we refer to Smaragdakis' tutorial [33].

Andersen-style Analysis

Andersen-style analysis is a flow- and context-insensitive algorithm for pointer analysis [49]. The algorithm is represented by a set of rules of the form 'If `x` points to `y` then the points-to set of `x` must be a subset of the points-to set of `y`'. This set of translations from statement to subset constraint defines the algorithm. The following table shows which constrain is inferred from which C statement in Andersen's analysis. In this table `pts(a)` refers to the points-to set of `a`.

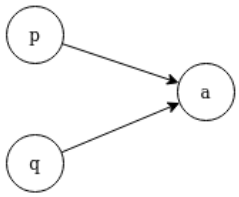
	C code	constraint
referencing	a = &b	{b}$\subseteq$ pts(a)
aliasing	a = b	pts(b) $\subseteq$ pts(a)
dereferencing read	a = *b	pts(*b) $\subseteq$ pts(a)
dereferencing write	*a = b	pts(b) $\subseteq$ pts(*a)

For each statement in the program, the resulting subset constraints are determined. These constraints are solved by representing them in an inclusion graph, where each node represents a variable. This means that the constraint `pts(b)` $\subseteq$ `pts(a)` results in an edge from node `b` to node `a`. This way, for each variable, its points-to set consists of the variables represented by its successive nodes in the graph. After going through the list of constraints and adding the resulting edges to the graph, the algorithm iteratively continues to solve the list of constraints until solving the constraints yields no new edges. At this point, the graph is finished and we can easily determine the resulting points-to sets of the variables by taking the adjacency list of the node corresponding to that variable. We apply an Andersen-style analysis on the following simple code example to demonstrate the algorithm.

```
1 p = &a; // Add edge from p to a
2 q = p;  // Add edge from q to all nodes p has outgoing edge to
```

Listing 2.9: Andersen-style analysis: precise results

Figure 2.1: Graph resulting from Andersen-style analysis



The constraint graph resulting from this piece of code is shown in Figure 2.1. Solving each constraint once will give us `pts(p)`=`pts(q)`={`a`}, meaning that `p` and `q` may alias. Iteratively solving the constraints does not change this outcome. In this case, the analysis yields a precise result. Note that since Andersen's algorithm is flow-insensitive, it does not typically yield fully precise results. Listing 2.10 provides another example to demonstrate this.

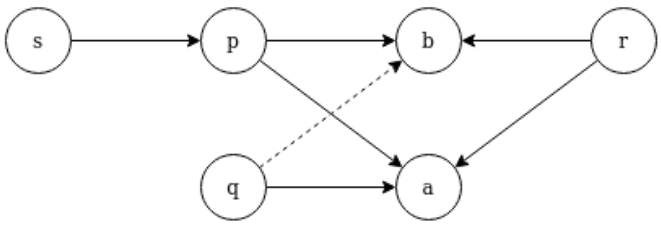
```
1 p = &a; // Add edge from p to a
2 q = p;  // Add edge from q to all nodes p has outgoing edge to
3 p = &b; // Add edge from p to b
4 s = &p; // Add edge from s to p
5 r = *s; // For each node x with incoming edge from s,
6 // add edge from r to all nodes x has outgoing edge to
```

Listing 2.10: Andersen-style analysis: imprecise results

This second example will result in the constraint graph shown in Figure 2.2. In this example, we need two iterations of the algorithm to finish. The single edge resulting from the second iteration is drawn as a dotted line in the figure. This result yields `pts(p)`=`pts(q)`=`pts(r)`={`a`,`b`}. In terms of aliasing, this would mean that we have the MAY alias set `<p,q,r>`. We can conclude this because `p`, `q` and `r` have equal points-to sets.

This result is an overestimation because, after line (3), `p` does not point to `a` anymore, so `r` will never point to `a`. This overestimation is caused by the flow-insensitivity of Andersen's algorithm. In this type of analysis, the order of the statements does not influence the result. This implies that there is no way to know if `p` points to `a` or `b` when statements `p=&b` and `r=*s` are executed. The precise result of a pointer analysis would be that `q` points to `a`, `p` points to `a` and `b`, `r` points to `b` and `s` point to `p`. So in practice, `q` and `r` will not alias.

Figure 2.2: Graph resulting from Andersen-style analysis



Andersen's algorithm can require $O(n^3)$ time (where n is the number of nodes in the inclusion graph). This is because, for a dereferencing statement such as `p=*q`, the algorithm might have to visit n^2 nodes. Specifically, `q` may point to n nodes, and these may again point to n other nodes. The algorithm adds n^2 edges from `p` to all of these nodes. We may have n of these statements, which yields a worst-case complexity of n^3. However, Sridharan and Fink have shown that, in practice, its runtime complexity is $O(n^2)$ [36].

Chapter 3

Current state-of-the-art pointer analysis

Many papers have been written about pointer analysis. Some of them date back more than twenty years. Nonetheless, the topic remains relevant today. Many papers have addressed the importance of accurate pointer analysis for detecting security issues [26, 15, 13, 42, 43]. Other papers discuss how we can use pointer analyses for optimization in compilers [8, 1]. In 2001, Hind extensively described the current status of research and the problems that remain in his paper "Pointer analysis: Haven't we solved this problem yet?" [20]. He categorizes a large number of different approaches and proposes improvements that can make pointer analysis results more useful. Now, almost twenty years later, numerous new papers have appeared, in this chapter, we evaluate the state of the current field of research.

3.1 Reproducibility

Although lots of work has been done on pointer analysis, the majority of papers do not provide an actual implementation of their algorithm. Although these papers formally describe their algorithm and include test results in which running times are compared with other algorithms, there is no source code available. Let alone there is a tool that can be downloaded easily to test and compare the algorithm. This is concerning because the results of scientific research are supposed to be reproducible. However, the solutions proposed by most pointer analysis papers can be implemented in many different ways that might affect the running time results. The problem has been addressed before by Hind, still, papers rarely contain an implementation of their analyses [20]. ACM has addressed the problem by assigning an 'artifact available' badge to papers that include an artifact that has been made permanently available. This encourages researchers to publish the implementation of their algorithm together with their paper.

Figure 3.1: The ACM 'artifact available' badge

3.2 Different approaches

The two most famous classic examples of approaches to pointer analysis are Andersen-style analysis [49], also referred to as inclusion-based analysis and Steensgaard-style analysis [37], referred to as unification-based analysis. Many papers propose improvements and optimizations to make one of these classical approaches more precise or faster [18, 30, 12, 32, 45, 16]. Other research proposes completely different and new pointer analysis algorithms. We will discuss a selection of these different approaches in this section.

Probabilistic pointer analysis

One example is probabilistic pointer analysis [46, 7, 6, 11]. Where a classic pointer analysis determines whether a points-to relation may or must exist, this type of analysis estimates the probability that a certain points-to relation holds. Da Silva and Steffan even show with their probabilistic analysis that many points-to relations that can occur are very unlikely to occur at runtime [7]. This is especially useful for compilers to determine if certain optimizations are necessary. However, it is less appropriate for finding security bugs since they might have a low probability of actually occurring.

BDD-based pointer analysis

Another completely different approach is binary decision diagram-based (BDD-based) pointer analysis [47, 4, 44]. In this type of analysis, the results are presented as a binary decision diagram. Binary decision diagrams are known to be very effective for compactly representing large sets and solving large state space problems. This approach is very suitable for scaling up analysis for large programs but generally does not increase precision.

Demand-driven analysis

In his paper from 2001, Hind recommends demand-driven pointer analysis [20]. Lots of demand-driven analyses have been proposed in response to this [39, 35, 43, 26, 15, 42, 45, 17]. Demand-driven pointer analysis is based on queries. It computes only the points-to information for the queried variables, instead of returning points-to sets for all variables. This makes demand-driven analysis faster and optimal in the sense that it does not compute unnecessary information [17]. This type of analysis is not appropriate for calculating a whole points-to graph since it would be much slower than the alternative approaches.

3.3 Different programming languages

Existing analyses differ greatly in the programming language they target. Most pointer analyses are proposed for the language C. This is because C, due to its lack of type safety [2], is prone to security vulnerabilities such as buffer overflows, format string exploits, and memory leaks [41].

Other languages such as C++ and Java make use of pointers as well. In Java, these pointers are mostly references to objects whereas in C and C++, pointers can also arise from indirect memory accesses by using the dereference operator '*' [38, 5, 42]. There are multiple pointer analyses for Java and C++ too [10, 35, 28].

3.3.1 Lower-level pointer analysis

While most Pointer Analyses are designed for higher-level languages such as C, C++, and Java. There have been a made a few efforts to develop lower-level pointer analysis as well [14, 9, 11, 3, 50]. Multiple researchers advocate performing pointer analysis on lower-level code such as assembly or executable code over higher-level languages. Unfortunately, an implementation or tool that allows us to perform pointer analysis on lower level code does not yet exist.

Guo et al. show that computing pointer information at the lower level yields more precise results compared to propagating this information form a high-level pointer analysis through subsequent code transformations. This is caused by transformations performed by the compiler that conservatively propagate dependence relations. We refer the reader to the paper by Guo et al. [14] for a more elaborate explanation of this phenomenon.

Moreover, since pointer analysis is a type of static analysis, high-level pointer analysis is only applicable when the source code is available. Therefore, tasks such as binary re-optimization, link-time optimization, and run-time optimization cannot benefit from it. Similarly, some resource-poor computers such as smart cards or embedded processors in small devices do not support programs written in higher-level languages, meaning that all programs should be written in assembly and no high-level source code is written.

Finally, lower-level pointer analysis can be used to analyze low-level software such as binary executables. According to Balakrishnan and Reps, there has been a growing need for tools that analyze executables in recent years [3]. This can be useful to decipher the behavior of worms and virus-infected code. Security vulnerabilities in binary executables are often hard to detect, pointer analysis can improve this process.

3.4 Comparing pointer analyses

The vast differences between pointer analyses and the unavailability of implementations make it difficult to compare different pointer analyses. Hind and Pioli propose some dimensions that should be considered to conduct a useful comparison of multiple pointer analyses [21]. They mention flow-sensitivity, context-sensitivity and field-sensitivity. In another paper, they use these metrics to compare six context-insensitive analyses [22]. Although these dimensions help us categorizing the different types of analyses, qualitative comparison remains difficult. One of the reasons for this is that analyses are inconsistent in their returned information and query format [34].

Unavailability of benchmark suites

Another part of the problem is that there are no benchmarks suites that are commonly used in pointer analysis research. We see that multiple researchers have developed own benchmark suites to test and compare their algorithms. But since these benchmark suites are again most of the time unavailable for download, others cannot use their suites to compare new algorithms. Some researchers have made efforts to create tools to compare pointer analyses [48, 38].

Recently, Zyrianov et al. have developed SrcPtr, a framework to support the implementation and comparison of pointer analysis algorithms [48]. The framework is based upon SrcML, a tool that generates an XML representation for C++, Java, and C code. SrcPtr takes this XML representation as input and performs pointer analysis on the XML file. Even though this tool is one of the few tools that can be downloaded easily and allows us to test different pointer analyses, it has some limitations. It doesn't support all types of analyses. For example, performing a flow-sensitive analysis is not possible. Neither does it support all programming languages, making it impossible to test many pointer analyses that do not meet these requirements using this tool.

Chapter 4

Challenges for assembly-level pointer analysis

In this chapter, we will discuss some of the challenges in developing a pointer analysis for assembly. Note that throughout the remainder of this work, when referring to 'assembly', we specifically target the Intel x86 instruction set.

4.1 Identifying pointers in assembly

4.1.1 Lack of variables

By definition, pointer analysis computes for each variable, the memory location it points to. However, the concept of variables in assembly is very different from most higher-level programming languages. High-level languages make it easy for the programmer to access memory by using variable names. Instead of 'put the value 6 in memory at location `0x83f4110`', we can just write '`int number = 6`' and we can use the alias '`number`' to later refer to this memory location. This way the programmer does not need to worry about the complicated memory address `0x83f4110`.

4.1.2 Lack of type information

Moreover, the statement '`int number = 6`' tells us something about the type of the value stored in the memory location, namely that it is an integer. Similarly, when we write `int* p` we know that on the memory location `p` is referring to, an address will be stored, which makes `p` a pointer.

4.1.3 Registers and memory

Even though we do not have actual variables in assembly, we can store values in memory and later read back the contents of this memory location. Assembly distinguishes between registers and memory. A register is a small storage location that can be accessed quickly by the CPU. Registers are typically used to store values that the processor needs quick access to, such as the instruction that is currently being processed and its operands. Memory, on the other hand, is a slower but larger storage space where programs and data are stored. Both a register and a memory location can hold a memory address, that is 'point to' a location in memory. So in a pointer analysis for assembly, we have to consider both registers and memory location as 'pointer variables'. But how do we know whether a register holds a memory address or just an arbitrary value? We will explain the three different manners in which we can make a register or memory location hold an address.

1. Accessing the heap with the `malloc` function

First of all, we can use the `malloc` function. Even though this is not an assembly instruction, `malloc` is a function that is commonly called from assembly. It is used to allocate memory on the heap. For reference, the heap is a specific part of memory that can be dynamically allocated during runtime. Calling `malloc` will allocate a specific amount of bytes in memory on the heap and returns the starting address of this space in eax (on 32-bit systems). This function is relevant for a pointer analysis because after calling `malloc` we know that register `eax` contains an address.

Since memory is dynamically allocated during runtime, this memory must be 'freed' again when it is no longer needed. As long as memory is not freed, it can't be used by other processes. We call this a memory leak. We can free the allocated memory by calling the `free` function, this way it can be allocated for a different purpose again. Note that, even though this memory is freed, we still have a pointer to that memory address on the heap and it is still possible to write to it. We want to avoid this behavior since it might cause a program to crash. This is where a pointer analysis for assembly could come in useful. In the following example, we see how a pointer analysis can detect references to freed memory.

```
1 mov edi, 40              ; number of bytes to allocate
2 extern malloc
3 call malloc              ; rax points to the allocated memory
4 mov DWORD PTR [rax],7;   ; write a constant into the memory
5 mov rdi,rax              ; allocated memory address is argument
6                            for free
7 extern free
8 call free                ; memory is freed
9 mov [rax],8              ; write a constant into the freed memory
```

Listing 4.1: Accessing freed memory on the heap

A pointer analysis can conclude that after the execution of this program, both `rdi` and `rax` point to the same memory location on the heap. This result helps us notice that writing to or reading from either `rdi` or `rax` after `free` is potentially harmful.

2. Accessing the stack with stack and base pointer

A second way in which we can interact with memory in assembly is by accessing the stack. The stack is a part of memory in which local variables are stored. In x86, two registers are used to keep track of the memory address of the stack. Register `esp` holds the stack pointer, which is the address of the current top of the stack. Register `ebp` holds the base pointer, the bottom of the stack. Note that the stack grows downwards so the address of the bottom will be higher than the address of the top of the stack. We can access values in between these two pointers, that is, values on the stack, by accessing addresses relative to the stack or base pointer.

Consider the simple C code in Listing 4.2, in which we declare three local variables and assign values to them. Listing 4.3 contains a translation of this C function into assembly. We see that in the assembly program, we first equal the base pointer to the stack pointer such that it points to the top of the stack. Then we decrease the stack pointer by 12. Accordingly, there is now 12 bytes of space between the base and stack pointer, which is perfect for three 4 byte integers. Consecutively, we can write values to these locations by writing them to the locations relative to the base pointer. `mov [ebp-4], 10` takes the address in `ebp` and subtracts 4, then writes the constant 10 in the resulting address.

A pointer analysis of this assembly code would yield that `[esp]` and `[ebp-12]` alias. Note that when we would perform pointer analysis on the C code, we would not find any aliases. This makes a good example of a pointer analysis on assembly giving us more information on what is happening in memory than the information we would get from analyzing the C code.

```
1 void foo() {
2   int a, b, c;
3   a = 10;
4   b = 5;
5   c = 2;
6 }
```

Listing 4.2: Declaring and assigning variables in C

```
1 push ebp             ; save the value of the current base pointer
2 mov ebp, esp         ; copy value of esp into ebp
3 sub esp, 12          ; allocate space on stack for local variables
4 mov [ebp -  4], 10   ; location of variable a
5 mov [ebp -  8], 5    ; location of b
6 mov [ebp - 12], 2    ; location of c
```

Listing 4.3: Declaring and assigning variables on the stack in assembly

3. Absolute memory addresses

The third way to make a register or memory location hold an address is by moving an absolute address as a constant into that register or memory location using the `mov` instruction. The following example shows how we can use the `mov` instruction to write an address into a register directly.

```
1 mov eax, 0x0000FFFF                ; place '0x0000FFFF' into eax
2 mov ebx, DWORD PTR [eax]           ; copy 32-bits in eax to ebx
3 mov ecx, DWORD PTR ds:0x0000FFFF   ; copy 32-bits in 0x0000FFFF to ecx
```

Listing 4.4: Memory addresses as constants

In the third line of code, the notation `ds:0x0000FFFF` is used. In this notation, the prefix to the address `ds:` specifies the segment register, and `0000FFFF` is the offset from the beginning of that segment register, in this case, the data segment. Segment registers are used for memory segmentation, a technique in which the physical memory is separated into segments. Memory in these segments can then be accessed by specifying the segment together with an offset. This way a 16-bit system could use more than 2^{16} bytes of memory. In modern operating systems, this mechanism is not used anymore. Operating systems now use paging for virtual memory management. Still, it is supported for backwards compatibility. Segment registers are usually set to zero so it does not change the meaning of the code. In this case, and throughout this work, the reader can assume that all segment registers are set to zero. The only difference between `0x0000FFFF` and `ds:0x0000FFFF` remains that the former is just a constant address, and the latter accesses the contents of that memory location. This is similar to putting square brackets around a register.

In general, programmers writing in assembly will not use absolute addresses like this in their code. When writing the code, we are not sure which specific memory locations we can access. This virtual address space is determined by the linker. When a program is compiled and linked, the memory accesses in the program will be linked to specific memory locations. Hence, when we disassemble a compiled and linked executable binary program, we will see many accesses to absolute (virtual) addresses. We can extract the exact range of virtual addresses assigned to the executable from the binary. On Linux systems, for example, this can be done using the `readelf` command.

As discussed in Section 3.3.1, analyzing disassembled binary files is an important benefit of performing pointer analysis on lower-level code. A pointer analysis that can recognize such absolute memory addresses can be very useful for this purpose.

4.2 The x86 instruction set

The x86 instruction set is an instruction set architecture developed by Intel. In general, x86 refers to the 32-bit variant whereas the 64-bit variant is specified as x86-64. As a so-called complex instruction set computer (CISC) architecture, x86 consists of many instructions. The majority is complex and executes more than one lower-level operation at once. The extensive size of the instruction set together with the absence of a formal definition of its semantics has been criticized by many [19][52]. In fact, in 2013, Stephen Dolan has concluded that the whole x86 instruction set can be reduced to only the `mov` instruction by proving this instruction to be Turing complete [51]. Christopher Domas has supported this proof by developing a compiler called 'Movfuscator' that allows the user to compile C code to valid x86 assembly consisting of `mov` instructions only[1].

4.2.1 The `mov` instruction

The move instruction, denoted as `mov`, is used to copy data between registers and memory. In a computer, a register holds the data that CPU is currently processing whereas the memory holds program instruction and data that the program requires for execution. With the move instruction, data can be moved from one register to another register, from a register to memory or from memory to a register. Note that because x86 is a load-store architecture, copying from memory directly to another location in the memory is not allowed. The following table contains a snippet of the x86 reference explaining the `mov` instruction and its syntax[2].

`mov` <reg>, <reg>	Copy value from register to register
`mov` <reg>, <const>	Store constant in register
`mov` <reg>, <mem>	Copy value from register to memory
`mov` <mem>, <reg>	Copy value from memory to register
`mov` <mem>, <const>	Store constant in memory

[1] Christopher Domas. Movfuscator. `github.com/xoreaxeaxeax/movfuscator`
[2] Microsoft Macro Assembler (MASM). `docs.microsoft.com/cpp/assembler/masm.`

Chapter 5

Prototype assembly-level pointer analysis

The main contribution of this thesis is a proof-of-concept implementation of assembly-level pointer analysis. The purpose of this implementation is to provide a prototype for assembly-level pointer analysis, rather than an extensive tool that can be used for every assembly program. It is meant to show the type of results one could expect from assembly-level pointer analysis and how to conquer the challenges discussed in Section 4. It also describes, for the first time, an approach for translating the constraints in an Andersen-style pointer analysis into constraints for assembly. In this chapter, we will discuss the design decisions that we made to implement this pointer analysis. Furthermore, we will discuss the experimental results of our tool and the limitations that our current implementation brings.

5.1 Movlang

From Section 4.2.1 we know that `mov` is Turing complete. In practice, we can even use Domas' Movfuscator [1] to compile each C program into valid x86 assembly consisting of `mov` instructions only. This means that a pointer analysis that works for the `mov` instruction can, in theory, analyse every C program. For this reason, we decided to limit our scope to the x86 `mov` instruction. We have defined a small language named 'Movlang', consisting of only this instruction. A definition of its grammar can be found in Appendix A1. We have used the tool ANTLR [2] in our implementation to generate a parser from this grammar. See Appendix A2 for the full implementation of our pointer analysis using the parser generated by ANTLR.

5.2 Syntax

The x86 assembly language has two main syntax branches: Intel and AT&T syntax. Multiple assemblers are using the Intel syntax. Two common assemblers are the Netwide Assembler (NASM) and the Microsoft Macro Assembler (MASM). Even though these two assemblers both use the Intel syntax, the syntax of the input assembly differs greatly between the two. For instance, NASM is case sensitive whereas MASM is not. It is beyond the scope of this thesis to go into further detail about the differences between the two. That is why in this work, we will stick to x86 (32-bit) MASM syntax. This is mainly because Movfuscator generates MASM syntax so this enables a user of our tool to convert any MASM assembly program to a `mov`-only program using Movfuscator, allowing this `mov`-only program as input for our pointer analysis.

[1] Christopher Domas. Movfuscator. `github.com/xoreaxeaxeax/movfuscator`.
[2] ANTLR. `www.antlr.org`

5.3 Implementation

We base our pointer analysis on Andersen's algorithm (see Section 2.3.3). The algorithm itself is the same but the difference is in the definition of the constraints. In Andersen's analysis, constraints were based on statements in the programming language C. We have redefined these constraints for the `mov` instruction.

5.3.1 Defining a pointer in Movlang

As discussed in Section 4.1.1, the concept of variables in assembly differs greatly from that of higher-level languages. This makes it particularly difficult to statically recognize if the value in a register or memory location is a valid address or just a value. Hence, to preserve simplicity, we will only consider a memory location or register to be a pointer, whenever an absolute address is written to that location. Our tool allows the user to provide a range of values that will be considered valid addresses. For a disassembled binary executable, the range of virtual addresses can be found by using the `readelf` command as described in Section 4.1.3. If this range is not provided by the user, the tool will by default consider all hexadecimal values valid addresses which will yield some overestimation in the points-to sets.

5.3.2 Defining subset-constraints for Movlang

We have redefined Andersen's subset-constraints such that they cover the syntax and semantics of the `mov` instruction. The following table shows how the constraints are defined.

x86 assembly statement	constraint
`mov` <reg_a>, < reg_b >	$\text{pts}(\text{reg}_b) \subseteq \text{pts}(\text{reg}_a)$
`mov` <reg>, <const>	$\text{pts}(\text{const}) \subseteq \text{pts}(\text{reg})$
`mov` <reg>, <mem>	$\text{pts}(\text{mem}) \subseteq \text{pts}(\text{reg})$
`mov` <mem>, <reg>	$\text{pts}(\text{reg}) \subseteq \text{pts}(\text{mem})$
`mov` <mem>, <const>	if const is a valid address, $\text{const} \in \text{pts}(\text{mem})$
	else $\text{pts}(\text{const}) \subseteq \text{pts}(\text{mem})$

Andersen's subset-constraints are meant for the language C, in which dereferencing a pointer and storing it in another variable can happen in one statement. For the `mov` instruction, that would take multiple moves. Since complicated statements are split up, the constraints for our pointer analysis are more simple than the original constraints.

5.4 Experimental results

In this section, we will discuss some experimental results of our pointer analysis. All code examples discussed in this section are included in the tool as described in Appendix A2. The results will demonstrate the in- and output that can be expected from our tool, as well as some important characteristics of our analysis. A simple example to demonstrate our the results of our analysis is given in Listing 5.1. The algorithm performs the following steps after each program point. In the first iteration:

1. 0xFFFFFF08 $\in$ pts(0xFFFFFF00)
2. pts(0xFFFFFF00) $\subseteq$ pts(eax)
3. pts(eax) $\subseteq$ pts(0xFFFFFF04)
4. 0xFFFFFF0C $\in$ pts(0xFFFFFF00)

In the second iteration:

1. No changes to graph
2. pts(0xFFFFFF00) $\subseteq$ pts(eax): add edge from eax to 0xFFFFFF0C

The rest of the steps have no impact on the graph anymore. The resulting points-to set are shown in Listing 5.1.

```
Address range provided: 0xFFFFFF00 - 0xFFFFFFFF
Input:
mov ds:0xFFFFFF00, 0xFFFFFF08
mov eax, DWORD PTR ds:0xFFFFFF00
mov ds:0xFFFFFF04, eax
mov ds:0xFFFFFF00, 0xFFFFFF0C

Output:
Points-to set of 0xFFFFFF00 is
 [ 0xFFFFFF08 0xFFFFFF0C ]
Points-to set of eax is
 [ 0xFFFFFF08 0xFFFFFF0C ]
Points-to set of 0xFFFFFF04 is
 [ 0xFFFFFF08 0xFFFFFF0C ]
```

Listing 5.1: Aliasing pointers in Movlang

The input code of this example is similar to the C code in Listing 5.2, where pointer variables `p` and `q` would refer to the 32-bits at memory addresses `0xFFFFFF00` and `0xFFFFFF04` respectively. Similarly the integer variables `a` and `b` would refer to the 32-bits at memory addresses `0xFFFFFF08` and `0xFFFFFF0C`

```
p = &a;
q=p;
p = &b;
```

Listing 5.2: Aliasing pointers in C

An Andersen-style alias analysis on this C code would result in the points-to relations `pts(p)={a,b}` and `pts(q)={a,b}`, meaning that `p` and `q` alias. Note that, when we compare the results of pointer analysis on the assembly

code with the results of the same analysis of C code, the assembly-level analysis results in one more points-to set, namely `pts(eax)={0xFFFFFF08}`. As we know from Section 4.2.1, the `mov` instruction cannot copy data directly from one memory location to another. Consequently, a C statement `q=p;` is separated into two assembly statements. The value is first copied from one memory location into a register and then copied from that register to the other memory location.

5.4.1 Flow-insensitivity

As we know from Section 2.3.3, Andersen-style analysis is flow-insensitive. We can compare the following code example to the example in Listing 5.1 and see that executing the same statements in a different order results in the same points-to set.

```
1  Address range provided: 0xFFFFFF00 - 0xFFFFFFFF
2  Input:
3  mov eax, DWORD PTR ds:0xFFFFFF00
4  mov ds:0xFFFFFF04, eax
5  mov ds:0xFFFFFF00, 0xFFFFFF0C
6  mov ds:0xFFFFFF00, 0xFFFFFF08
7
8  Output:
9  Points-to set of 0xFFFFFF00 is
10  [ 0xFFFFFF08 0xFFFFFF0C ]
11 Points-to set of eax is
12  [ 0xFFFFFF08 0xFFFFFF0C ]
13 Points-to set of 0xFFFFFF04 is
14  [ 0xFFFFFF08 0xFFFFFF0C ]
```

Listing 5.3: Executing the statements in Listing 5.1 in a different order

Registers as intermediate storage location

In Listing 5.1 we have already seen an example in which the single C statement `q=p` is translated into the following two separate assembly statements: `mov eax , DWORD PTR ds:0 xFFFFFF005` and `mov ds:0xFFFFFF04 , eax`. Since x86 is a load-store architecture, registers are used continuously to store values temporarily. The reason for this is that copying directly from memory to memory is not allowed. Throughout execution of a program, one register may have many values. Our algorithm iteratively updates the points-to sets of all nodes pointing to a register by adding all values that specific register has had to those points-to sets. This can cause the result that every variable to which something has been moved from a register once, now points to every address that has been stored in that register. Accordingly, the algorithm might in the worst case conclude that everything points to everything.

The following example shows how many imprecise points-to aliases arise from using register `eax` as intermediate storage. This is a sound solution but it is obviously very imprecise. In small programs, overestimates are usually relatively small. But in larger programs, we cannot avoid using the same register multiple times. This means we will see more superfluous variables in the points-to sets. Hence, our pointer analysis performs better on small code fragments in which a register doesn't change value too often.

```
1  Address range: 0xFFFFFF00 - 0xFFFFFF30
2  Input:
3  mov ds:0xFFFFFF10, 0xFFFFFF00
4  mov ds:0xFFFFFF14, 0xFFFFFF04
5  mov ds:0xFFFFFF18, 0xFFFFFF08
6  mov eax , DWORD  PTR ds:0xFFFFFF10
7  mov ds:0xFFFFFF20 , eax
8  mov eax , DWORD  PTR ds:0xFFFFFF14
9  mov ds:0xFFFFFF24 , eax
10 mov eax , DWORD  PTR ds:0xFFFFFF18
11 mov ds:0xFFFFFF28 , eax
12
13 Output:
14 Points-to set of 0xFFFFFF10 is
15  [ 0xFFFFFF00 ]
16 Points-to set of 0xFFFFFF14 is
17  [ 0xFFFFFF04 ]
18 Points-to set of 0xFFFFFF18 is
19  [ 0xFFFFFF08 ]
20 Points-to set of eax is
21  [ 0xFFFFFF00 0xFFFFFF04 0xFFFFFF08 ]
22 Points-to set of 0xFFFFFF20 is
23  [ 0xFFFFFF00 0xFFFFFF04 0xFFFFFF08 ]
24 Points-to set of 0xFFFFFF24 is
25  [ 0xFFFFFF00 0xFFFFFF04 0xFFFFFF08 ]
26 Points-to set of 0xFFFFFF28 is
27  [ 0xFFFFFF00 0xFFFFFF04 0xFFFFFF08 ]
```

Listing 5.4: Registers as intermediate storage

5.4.2 Time complexity

In Section 2.3.3 we have discussed that Andersen's algorithm for C programs has a theoretical runtime complexity of $O(n^3)$. This possible $O(n^3)$ complexity is caused by C statements such as `p=*q`. Similar to what is discussed earlier about the statement `p=q` in Listing 5.1 and Listing 5.2, translating such a C statement into assembly will yield multiple `mov` statements. This causes the set of constrains of our implementation to only contain 'simple' constraints, in contrasts to Andersen's complex constraints for dereferencing. As a result, the runtime complexity of our implementation is only $O(n^2)$. However, effectively there will not be a significant difference between runtimes of the two since Sridharan and Fink have shown that, in practice, Andersen's analysis' runtime complexity is $O(n^2)$ as well [36].

5.5 Limitations

In this section, we discuss the limitations of our implementation. In addition, we suggest improvements to solve current problems. These suggestions can form a basis for future research to transform our prototype implementation into a complete tool that can be widely used on x86 assembly programs.

5.5.1 The x86 `mov` instruction only

Our prototype analysis is only meant for the x86 `mov` instruction. Naturally, this limits the set of real-world assembly programs can be analyzed. However, we can think of some examples in which we can find security vulnerabilities in code consisting of the `mov` instruction only. See the example in Listing 5.5.

```
1 mov DWORD PTR [eax], 0
2 mov DWORD PTR [ebx], 1
3 mov rdi, DWORD PTR [rax]
4 mov DWORD PTR[rsp-4+4*rdi], 0x11111111
```

Listing 5.5: Overwriting the return address

In this example, we assume registers eax and ebx are pointers. If `eax` and `ebx` point to different locations, the instruction in line (4) just writes the address `0x11111111` to its stack frame (at `rsp-4`). However, if `eax` and `ebx` alias, the return address, which is usually stored in `rsp`, is overwritten. Note that our pointer analysis itself does not compute the mathematical operations in line (4) yet, so it will not recognize that `rsp-4+4*rdi` is equal to `rsp` or `rsp-4`. But, as a part of a more extensive whole program analysis, our pointer analysis can help in detecting these type of vulnerabilities.

5.5.2 Testing on a larger scale

There is no library or benchmark suite written in only `mov` instructions. Hence, it is difficult to properly test our analysis on a larger scale and to compare it to others. We should first extend the analysis to work for a larger set of instructions. Alternatively, we could translate C programs into `mov` instructions and test our analysis on the translation. In Section 4.2.1 we have shortly discussed Christopher Domas' 'Movfuscator' [3], a tool that can be used to convert C code into x86 assembly containing only `mov` instructions. If Domas' tool would provide a correct translation, that is, the result of each translated instruction is exactly the same as the original instruction. Then, taking C programs, using Movfuscator to convert them to assembly and then performing our pointer analysis on it could yield interesting results. However, Movfuscator does not claim to give a correct translation. Extra operations are used on a memory location holding a state, to make the execution of the program work. This obfuscates the result of the pointer analysis.

[3]Christopher Domas. Movfuscator. `github.com/xoreaxeaxeax/movfuscator`

5.5.3 Registers and flow-insensitivity

Because precise pointer analysis is undecidable, our pointer analysis results in an overestimation of the actual points-to relations. Just like Andersen's algorithm for pointer analysis, our analysis is flow-insensitive. Flow-insensitivity can lead to very large points-to sets when the same pointer variable is reassigned with a new value many times throughout the code. In assembly, it is common that a register holds many different values during a program execution. As discussed in Section 5.4.1, this can cause a large overestimation.

The precision of the algorithm can be improved by making the algorithm flow-sensitive, or at least party flow-sensitive for the registers. In practise, this means that for each new assignment of the value in a register, this register is considered a new 'variable'. As an example, we will show what the results of an analysis of the code in Listing 5.4 would be if we would make it partly flow-sensitive for the registers, see Listing 5.6.

```
1  Address range: 0xFFFFFF00 - 0xFFFFFF30
2  Input:
3  mov ds:0xFFFFFF10, 0xFFFFFF00
4  mov ds:0xFFFFFF14, 0xFFFFFF04
5  mov ds:0xFFFFFF18, 0xFFFFFF08
6  mov eax , DWORD  PTR ds:0xFFFFFF10  ; new 'variable' eax0
7  mov ds:0xFFFFFF20 , eax
8  mov eax , DWORD  PTR ds:0xFFFFFF14  ; new 'variable' eax1
9  mov ds:0xFFFFFF24 , eax
10 mov eax , DWORD  PTR ds:0xFFFFFF18  ; new 'variable' eax2
11 mov ds:0xFFFFFF28 , eax
12
13 Output:
14 Points-to set of 0xFFFFFF10 is
15  [ 0xFFFFFF00 ]
16 Points-to set of 0xFFFFFF14 is
17  [ 0xFFFFFF04 ]
18 Points-to set of 0xFFFFFF18 is
19  [ 0xFFFFFF08 ]
20 Points-to set of 0xFFFFFF20 is
21  [ 0xFFFFFF00 ]
22 Points-to set of 0xFFFFFF24 is
23  [ 0xFFFFFF04 ]
24 Points-to set of 0xFFFFFF28 is
25  [ 0xFFFFFF08 ]
```

Listing 5.6: Partly flow-sensitive analysis for Movlang

It is clear that the points-to sets are significantly more precise compared to the example in Listing 5.4. Besides the shown output, this partial flow-sensitivity would create multiple separate points-to sets for each register, in this case `eax0`, `eax1` and `eax2`. Note that we have omitted the points-to sets of these registers in the output because these points-to sets are not useful in itself. We are not interested in the values of the registers as they are constantly overwritten.

Chapter 6

Conclusions

In the last 20 years, extensive research has been done in the field of pointer analysis for code in high-level languages. However, lower-level pointer analysis has the benefit of being able to analyse software of which no source code is available, such as binary executables [3]. In fact, research on pointer analysis for lower-level code has shown that in some cases, pointer analysis for assembly or executable code can provide more accurate results than analysis of higher-level programs [14].

In Chapter 4, we have identified that the key challenges of applying pointer-analysis to assembly code are the lack of variables, lack of typing information and the complexity of the CISC style of the x86 instruction set.

Our contribution, a pointer analysis for the x86 `mov` instruction only, is the first available open-source tool that performs Andersen-style [49] pointer analysis on x86 assembly. As described in Chapter 5, our implementation functions as a proof of concept for assembly-level pointer analysis in general. The results of our analysis are relatively precise for small programs. For larger programs in which registers change value multiple times throughout the program, results are less precise but sound.

Our proof-of-concept implementation opens up the field for future researchers to extend this prototype in many ways. As discussed in Section 5.5, extending our implementation to support the complete x86 instruction set would greatly enlarge the set of real-world assembly programs can be analyzed. Furthermore, making the algorithm flow-sensitive would greatly increase precision and avoid overestimates caused by registers having many different values throughout the program.

Bibliography

[1] Darren C. Atkinson and William G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 46–55. ACM, 1998.

[2] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, pages 332–341, 2005.

[3] Gogul Balakrishnan and Thomas Reps. Analyzing Memory Accesses in x86 Executables. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[4] Marc Berndl, Ondřej Lhoták, Lhot Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to Analysis using BDDs. *ACM SIGPLAN Notices*, 38(5):103–114, 2003.

[5] Barry W. Boehm and Philip N. Papaccio. Understanding and Controlling Software Costs. *IEEE TRANSACTIONS ON SOFTWARE ENGlNEERlNG*, 14(10):1462–1477, 1988.

[6] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Ju, and Jenq Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *Sigplan Notices - SIGPLAN*, 38:25–36, 10 2003.

[7] Jeff Da Silva and J Gregory Steffan. A Probabilistic Pointer Analysis for Speculative Optimizations. *ACM SIGPLAN Notices*, 2006.

[8] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Static Analysis: 8th International Symposium*, pages 260–278, Paris, 2001.

[9] Saumya Debray, Robert Muth, and Matthew Weippert. Alias Analysis of Executable Code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, 1998.

[10] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems: 13th Asian Symposium*, pages 465–484, Pohang, 2015.

[11] M Fernandez and R Espasa. Speculative alias analysis for executable code. In *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*, pages 222–231, 2002.

[12] Jeffrey Foster, Manuel Fähndrich, and Alexander Aiken. Flow-Insensitive Points-to Analysis with Term and Set Constraints. *University of California at Berkeley*, 1997.

[13] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the Importance of Points-To Analysis and Other Memory Disambiguation Methods For C Programs. *ACM SIGPLAN Notices*, 36:47–58, 2001.

[14] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. Practical and accurate low-level pointer analysis. In *Proceedings of the 2005 International Symposium on Code Generation and Optimization*, pages 291–302, 2005.

[15] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. In *Science of Computer Programming*, 2005.

[16] Ben Hardekopf and Calvin Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 289–298, 2011.

[17] Nevin Heintze and Olivier Tardieu. Demand-Driven Pointer Analysis. *ACM SIGPLAN Notices*, 2001.

[18] Nevin Heintze and Olivier Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 254–263, 2001.

[19] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 13-17-June-2016, pages 237–250. Association for Computing Machinery, 6 2016.

[20] Michael Hind. Pointer analysis: Haven't we solved this problem yet? *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[21] Michael Hind and Anthony Pioli. Which Pointer Analysis Should I Use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, page 113–123, 2000.

[22] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, 2001.

[23] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. An efficient data structure for must-alias analysis. In *CC 2018 - Proceedings of the 27th International Conference on Compiler Construction, Co-located with CGO 2018*, volume 2018-February, pages 48–58. Association for Computing Machinery, Inc, 2 2018.

[24] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[25] William Landi and Barbara G Ryder. Pointer-induced Aliasing: A Problem Classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 93–103, New York, NY, USA, 1991. Association for Computing Machinery.

[26] Benjamin Livshits and Monica S Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. *ACM SIGSOFT Software Engineering Notes*, 28(5):317–328, 2003.

[27] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Möller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *Communications of the ACM*, 58(2):44–46, 1 2015.

[28] Prakash Prabhu and Priti Shankar. Field Flow Sensitive Pointer and Escape Analysis for Java Using Heap Array SSA. In Alpuente María and Germán Vidal, editors, *Static Analysis*, pages 110–127, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[29] G Ramalingam. The Undecidability of Aliasing. In *ACM Transactions on Programming Languages and Systems*, 1994.

[30] Atanas Rountev and Satish Chandra. Off-line Variable Substitution for Scaling Points-to Analysis. *ACM SIGPLAN Notices*, 35, 5 2000.

[31] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2622:126–137, 2003.

[32] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 12 1996.

[33] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–80, 2015.

[34] Johannes Späth, Fraunhofer Iem, and Karim Ali. Context-, Flow-, and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. In *Proceedings of the ACM on Programming Languages*, volume 3, 2019.

[35] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *Leibniz International Proceedings in Informatics, LIPIcs*, 2016.

[36] Manu Sridharan and Stephen J. Fink. The complexity of Andersen's analysis in practice. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5673 LNCS, pages 205–221, 2009.

[37] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time Constraint-based Analysis. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[38] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. *Proceedings of CC 2016: The 25th International Conference on Compiler Construction*, pages 265–266, 2016.

[39] Yulei Sui and Jingling Xue. Value-Flow-Based Demand-Driven Pointer Analysis for C and C ++. *IEEE Transactions on Software Engineering*, 14(8):1–23, 2018.

[40] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA. In *Programming languages and systems. 9th Asian symposium*, pages 155–171, Kenting, 2011.

[41] David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of Network and Distributed Systems Security*, 2000.

[42] Ji Wang, Xiao Dong Ma, Wei Dong, Hou Feng Xu, and Wan Wei Liu. Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis. *Journal of Computer Science and Technology*, 24(2):347–356, 2009.

[43] Yuexing Wang, Guang Chen, Min Zhou, Ming Gu, and Jiaguang Sun. TsmartGP: A tool for finding memory defects with pointer analysis. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 1170–1173, 2019.

[44] John Whaley and Monica S Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, 2004.

[45] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by Level: Making Flow- and Context-Sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *Proceedings of the 8th International Symposium on Code Generation and Optimization*. ACM, 2010.

[46] Yu-Min Lu and Peng-Sheng Chen. Probabilistic Alias Analysis of Executable Code. *International Journal of Parallel Programming*, 39(6), 2011.

[47] Jianwen Zhu and Silvian Calman. Symbolic Pointer Analysis Revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157, Washington, DC, USA, 2004.

[48] Vlas Zyrianov, Christian Newman, Drew Guarnera, Michael Collard, and Jonathan Maletic. SrcPtr: A framework for implementing static pointer analysis approaches. *IEEE International Conference on Program Comprehension*, 2019-May:144–147, 2019.

Technical reports

[49] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, 1994. Accessed on Jun 24 2020. Available: `ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z`.

[50] David Brumley and James Newsome: Alias analysis for assembly. Technical report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, December 2006. Accessed on Jun 24 2020. Available: `reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-180R.pdf`.

[51] Stephen Dolan. mov is Turing-complete. Technical report, University of Cambridge, 2013. Accessed on Jun 24 2020. Available: `stedolan.net/research/mov.pdf`.

[52] Christopher Domas. Breaking the x86 ISA. Technical report, 2017. Accessed on Jun 24 2020. Available: `blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf`.

Appendix

A1. Movlang grammar

```
grammar movlang;

program : statement+;

statement : regToReg | regToMem | memToReg | conToReg | conToMem ;

regToReg : 'mov' REG ',' REG;
regToMem : 'mov' mem ',' REG;
memToReg : 'mov' REG ',' mem;
conToReg : 'mov' REG ',' constant;
conToMem : 'mov' mem ',' constant;

mem : '[' location ']' | 'BYTE PTR [' location ']' | 'WORD PTR ['
    location ']' | 'DWORD PTR [' location ']' | address | 'BYTE PTR '
    address | 'WORD PTR ' address | 'DWORD PTR ' address ;
location : REG | REG '+' constant | REG '-' constant | REG '+' REG |
    REG '-' REG | REG '+' REG '*' constant | REG '*' constant '+'
    constant;
address : 'ds:' HEX_NUMBER ;
constant : HEX_NUMBER | DEC_NUMBER ;

REG : 'eax'| 'ebx' |'ecx' | 'edx' | 'esi' | 'edi' | 'esp' | 'ebp' | '
    ax' | 'bx' | 'cx' | 'dx' | 'ah' | 'bh' | 'ch' | 'dh' | 'al' | 'bl'
     | 'cl' | 'dl' ;
DEC_NUMBER : DIGIT+ ;
HEX_NUMBER : '0x' HEX_DIGIT+ | HEX_DIGIT+;
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
DIGIT : ('0'..'9') ;
WS : [ \n\t]+ -> skip ;
```

Listing 6.1: The Movlang grammar in the format required by ANTLR

A2. Pointer analysis implementation

The full implementation of our pointer analysis can be found at `https://github.com/charbella/Movlang`. All programs in the Section 4.2.4 are included in this tool in the folder 'examples'. We refer the reader to the README for a user guide of our tool.