Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# SSA Based Flow-Sensitive Points-To Analysis via Value Flow Graph Reachability using Binary Decision Diagrams

Bachelor-Thesis

Cyrill Gössi

October 1, 2014

Advisor: Prof. Dr. Thomas R. Gross

Laboratory for Software Technology
Department of Computer Science, ETH Zürich

# Acknowledgements

I would like to thank Professor Thomas Gross for having given me the opportunity to conduct a thesis under his direct supervision. The subject constituting this thesis was interesting but also demanding to investigate and I was grateful for the motivating and guiding words Professor Gross found when unexpected obstacles came up.

# Abstract

In 2011, Oracle Labs Australia Researchers Cristina Cifuentes et al. presented in [1] a flow-sensitive points-to analysis algorithm that, as a novelty at this time, reduces the analysis to a graph reachability problem. Performance evaluations conducted in [1] indicate that this new reduction can lead to significant runtime improvements over previously established methods. However, it was also found that for some benchmarks the space requirements of the new algorithm can be up to an order of magnitude higher.

In this thesis we implemented the algorithm presented in [1] into LLVM [2] and investigated the effect on the algorithms space and time requirements when C++ STL sets are replaced by Binary Decision Diagrams as representations of the points-to sets involved. We found that by using Binary Decision Diagrams, the algorithm experiences a runtime slowdown of 27.3% while the space requirements can be reduced up to 65.6% for large enough benchmarks.

# Contents

Chapter 1

# Introduction

A points-to analysis is a static program analysis used to compute the set of abstract memory locations pointer variables in a given source code file can possibly point to. Knowing whether two given pointer variables either always, never or sometimes point to the same memory location is a prerequisite for analysis and optimization problems like loop-invariant code motion [3], detecting memory safety violations [4] or memory race detection in multi-threaded applications [5]. As shown in [6], points-to analysis is generally undecidable. This has led to the emergence of a multitude of different approximation techniques, each trading precision for efficiency in an idiosyncratic manner and with time complexities ranging from almost linear [7] up to doubly exponential in the number of program variables [8].

The method underlying this thesis and presented in [1] simplifies the points-to analysis to a graph reachability problem and summarizes pointer information in a value flow graph. By this, [1] reaches the precision of approaches respecting the execution order of program statements while getting close to the runtime efficiency of order disrespecting approaches with a usually much lesser precision. The performance evaluation presented in [1] found the analysis to be up to 28x faster than [9], the previous state-of-the-art flow-sensitive analysis, and to be even competitive with [10], the current state-of-the-art flow-insensitive analysis. For some benchmarks however, the memory footprint of [1] is up to an order of magnitude higher than the one of the techniques [9][10] it was tested against.

In this thesis we implemented [1] into LLVM[1] [2] and investigated the use of Binary Decision Diagrams, which can be thought of as canonical representations of Boolean functions, when trying to optimize the space requirement deficiencies found for method [1].

[1] LLVM: llvm.org. Last accessed: September 29, 2014

The remainder of this thesis is structured as follows: chapter 2 presents the theoretical background by introducing notions and concepts from graph and compiler theory, by introducing basic alias analysis and by defining and characterizing Binary Decision Diagrams. Chapter 3 introduces the methodology of the analysis [1]. Chapter 4 presents the implementation of the analysis [1]. Chapter 5 evaluates the implementation and discusses the results and chapter 6 concludes the thesis and gives an outlook on possible future work.

Chapter 2

# Background

This chapter aims to provide enough knowledge about general graph and compiler theory, LLVM [2] and its intermediate code representation, points-to analysis as well as about Binary Decision Diagrams such that subsequent chapters detailing the methodology and algorithm of the Binary Decision Diagram using, SSA based flow-sensitive points-to analysis of [1] should be understandable.

It's also here where a small C application is introduced, shown in listing 2.1, that will be used throughout the thesis to illustrate the many theoretical concepts and ideas that are about to be put forward.

After line 11 in listing 2.1 was executed, the pointer variable $a$ points to array $A$, $b$ points to array $B$, $v$ to $a$ and $w$ to $b$. Nothing about this changes if the control flow reaches line 19 via the if-branch and the character '?' will be stored into array $A$. If the else-branch is taken, swap() will swap the values where $v$ and $w$ point to, i.e. the values of $a$ and $b$, and, as of line 17, $a$ will point to array $B$, ultimately leading to '?' being stored into array $B$.

## 2.1 Graph Theory

For the sake of brevity, this part of the theory is just presented as a sequence of definitions.

**Definition 2.1** A *directed graph* is a pair $G = (N, E)$ of sets where $E \subseteq N \times N$. Elements of $N$ are called *nodes* and elements of $E$ are called *edges*.

**Definition 2.2** Let a directed graph $G = (N, E)$ be given. $G$ is *acyclic* if no path through $G$ along the direction of the edges has the same node visited twice.

**Definition 2.3** Let the graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ be given. $G_1$ is a *subgraph* of $G_2$ iff $N_1 \subseteq N_2$ and $E_1 \subseteq E_2$.

```c
#include <stdio.h>
void swap(int *p, int *q){
    int tmp = *p;
    *p = *q; *q = tmp;
}
int main() {
    char A[] = { 'x' };
    char B[] = { 'x' };
    char *a, *b, **v, **w;
    int i;
    v = &a; w = &b; a = &A; b = &B;
    scanf("%d", &i);
    if(i)
        *a = 'A';
    else {
        swap(v, w);
        *a = 'B';
    }
    *a = '?';
    printf("%c\n", *A);
    return 0;
}
```

**Listing 2.1:** Running Example

**Definition 2.4** Graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ are *isomorphic* iff there is a bijection $f : N_1 \to N_2$ such that $(n_1, n_2) \in E_1 \Leftrightarrow (f(n_1), f(n_2)) \in E_2$.

**Definition 2.5** Let a directed graph $G = (N, E)$ be given. The set of *immediate predecessors* of $n \in N$ is $\{ n_0 \mid (n_0, n) \in E \}$. Note that this set may be empty.

**Definition 2.6** Let a directed graph $G = (N, E)$ be given. The set of *immediate successors* of $n \in N$ is $\{ n_0 \mid (n, n_0) \in E \}$. Note that this set may be empty.

## 2.2 Compiler Theory

This section starts off with defining some most elementary notions of compiler theory and then advances to introducing the Static Single Assignment form representation of source code.

### 2.2.1 Elementary Notions

**Definition 2.7** A Basic Block (BB) is a sequence of source code statements that is maximal (no more statements can be added) and that in the order

implied by the sequence has either all or none of its statements executed [11].

**Definition 2.8** The Control Flow Graph (CFG) of a program $P$ is a directed graph in which the nodes represent the BBs from $P$ and the edges represent the control flow paths in $P$ [11].

**Definition 2.9** Let a CFG $G = (N, E)$ with $x, y \in N$ be given. *$x$ dominates $y$* if every path from the root of $G$ to $y$ goes through $x$.

**Definition 2.10** Let a CFG $G = (N, E)$ with $x, y \in N$ be given. *$x$ strictly dominates $y$* if $x$ dominates $y$ and $x \neq y$.

**Definition 2.11** Let a CFG $G = (N, E)$ with $x, y \in N$ be given. $x$ is the *immediate dominator* of $y$ if $x$ strictly dominates $y$ and for no $z \in N$ does $z$ strictly dominate $y$ while $x$ strictly dominates $z$.

**Definition 2.12** Let the CFGs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ with $G_1$ a subgraph of $G_2$ as well as $x, y \in N_2$ be given. $x$ is the *most immediate dominator* of $y$ in $G_1$ if $x \in N_1$ and $x$ strictly dominates $y$ in $G_2$ and for no $z \in N_1$ does $x$ strictly dominate $z$ in $G_1$ while $z$ strictly dominates $y$ in $G_2$. In other words: $x$ is the first node of $N_1$ to be met if one starts walking up the dominator tree from $y \in N_2$.

**Definition 2.13** Let a CFG $G = (N, E)$ with $x, y \in N$ be given. $y$ is in the *dominance frontier* of $x$, written as $y \in DF(x)$, if $x$ does not strictly dominate $y$ but $x$ dominates some immediate predecessor of $y$. The *dominance frontier* of $S \subseteq N$ is defined as $DF(S) = \bigcup_{s \in S} DF(s)$.

**Definition 2.14** Let a CFG $G = (N, E)$ be given. The *iterated dominance frontier* of $S \subseteq N$, written as $IDF(S)$, is the limit of the following sequence:

$$IDF_0 = DF(S)$$
$$IDF_{i+1} = DF(S \cup IDF_i)$$

### 2.2.2 Static Single Assignment

A program is in Static Single Assignment (SSA) [12] form if every variable is assigned exactly once. Redefinitions of a variable are replaced by definitions of new variables. If those newly introduced definitions happen to be on different CFG paths, a $\varphi$ node representing the combination of these definitions is placed at the joint point in the CFG where these paths merge again. $\varphi$ nodes are purely virtual in the sense that no machine instructions exist to execute them. As such, $\varphi$ nodes will be turned into a sequence of assignment instructions inserted into all BBs belonging to the immediate predecessor set of the joint point BB.

Since variables can be indirectly defined through pointers, see line 4 in listing 2.1 for ex., building SSA requires pointer information. This leads to an unlucky circular dependency as most pointer analyses build upon SSA. To overcome this difficulty, modern compilers such as GCC[1] and LLVM [2] employ *partial* SSA where only those variables are represented in SSA whose addresses are never taken.

## 2.3 LLVM

LLVM [2], the compiler infrastructure used in this thesis to implement the points-to analysis [1], has an Intermediate Representation (IR) rich in instructions out of which, and together with the general basics of LLVM IR, only the small handful-sized subset that proved to be relevant for the implementation of [1] will be discussed next.

### 2.3.1 LLVM IR

The LLVM IR[2] is a strongly typed language and uses SSA for all non-memory located variables. As LLVM allocates every address-taken variable in memory, the LLVM IR employs *partial* SSA in the sense introduced above. Furthermore, LLVM defines for each global variable a region of memory allocated at compilation time and lets the corresponding LLVM IR global variable point to it. Accessing the contents of global variables in LLVM therefore happens via one level of pointer indirection.

Allocating, reading and writing memory in LLVM is done through the three instructions described below.

#### alloca - Allocating Memory

The generic syntax to allocate memory in LLVM is:

*result* = alloca *type*[, *ty NumElements*][, align *alignment*]

This will yield *result* to be of type *type** and to point to a memory location of size sizeof(*type*) x *NumElements*, where *NumElements* defaults to 1 if not otherwise specified.

#### load - Reading from Memory

The generic syntax to load content from memory is:

[1]GNU Compiler Collection: gcc.gnu.org. Last accessed: September 29, 2014

[2]LLVM IR Documentation: llvm.org/docs/LangRef.html. Last accessed: September 29, 2014

*result* = load *ty** *pointer*[, align *alignment*]

This instruction will load the content of the memory location pointed to by *pointer* of type *ty** into *result* of type *ty*.

#### store - Writing to Memory

The generic syntax for writing to memory is:

store *ty value*, *ty** *pointer*[, align *alignment*]

This will store the value of *value* of type *ty* to the memory location pointed to by pointer *pointer* of type *ty**.

Two more instructions prove to be essential for implementing the analysis [1]; The first deals with type conversion while the second is about accessing elements of aggregate data structures.

#### bitcast - Converting Types

The generic syntax for converting types without changing any bits is:

*result* = bitcast *ty value* to *ty2*

This will convert the value of *value* of type *ty* to type *ty2*. If the type *ty* is a pointer type then *ty2* must be a pointer type too.

#### getelementptr - Accessing Elements of Aggregates

The generic syntax to get the address of a subelement of an aggregate is:

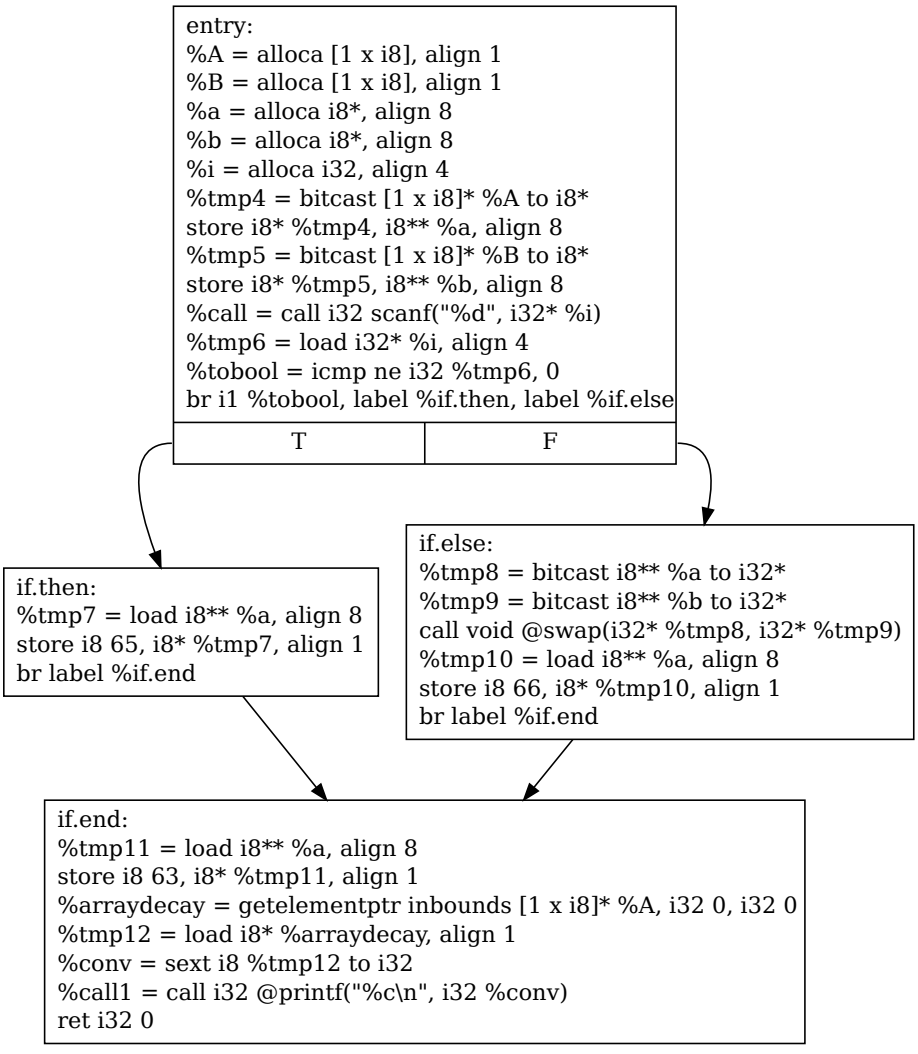
*result* = getelementptr inbounds *pty** *ptrval*{, *ty idx*}*

This will access a subelement of the aggregate pointed to by the pointer *ptrval* of type *pty**. The subelement, or possibly a subelement of a subelement, whose address is to be returned, is specified by the sequence of indexes following the pointer.

The CFGs of the LLVM IR produced for the sample code of listing 2.1 are shown in listings 2.2 and 2.3. The CFGs are slightly simplified such that only code sections essential for the upcoming chapters are included. With the theory presented so far, similar points-to reasoning as performed on listing 2.1 in the introduction to this chapter should now be possible on the IR versions of the C code.

```
entry:
%tmp0 = load i32* %p, align 4
%tmp1 = load i32* %q, align 4
store i32 %tmp1, i32* %p, align 4
store i32 %tmp0, i32* %q, align 4
ret void
```

**Listing 2.2:** IR CFG of swap(int *p, int *q)




**Listing 2.3:** IR CFG of main()

## 2.4 Points-to Analysis

A points-to analysis computes the set of memory locations a pointer-typed variable can point to during the execution of a program. Ideally those sets are exact. However, points-to analysis is undecidable in nature [6] and in all generality solutions can only be approximated. Such an approximation either respects the execution order of program statements and computes different points-to sets for different points in a program, then said to be *flow-sensitive*, or disrespects the execution order and is then said to be *flow-insensitive*. Examples of *flow-insensitive* approaches are [13][7][10], while examples of *flow-sensitive* ones are [14][9] as well as the paper underlying this thesis ([1]).

In the next section, we describe the basic principles and techniques of points-to analysis, ranging from memory abstractions to the formal background of *flow-sensitive* approaches. The full details about the methodology of paper [1] and the algorithms used for implementing it will be elaborated on in chapter 3 and chapter 4, respectively.

### 2.4.1 Memory Model

As points-to analyses are statical analyses, i.e. no information about the execution of the program or the runtime environment is available, memory has to be abstracted and the algorithm has to work with abstract memory locations. A widely used memory abstraction was introduced by Lars Ole Andersen in [13] and identifies abstract memory locations by the label of the source code instruction allocating the location. As most memory is byte-addressable, and if no further distinction is made, having an instruction allocating space for an object larger than one byte will lead to one abstract memory location representing multiple concrete memory locations. Depending on whether such a distinction is made, i.e. subelements (also called fields) of allocated objects have their own abstract memory location, a points-to analysis is categorized into being *field-sensitive* or *field-insensitive*, with analysis [1] being *field-insensitive*.

### 2.4.2 Relevant Statements for Analyzing C Code

Assuming that nested pointer dereferences were eliminated by introducing auxiliary variables, a points-to analysis analyzing C code has to consider only 5 types of statements [13]. These statements, together with the constraints they impose on the points-to sets involved are listed in table 2.1 and briefly discussed thereafter.

The *Base* type statement, reflecting the Andersen style variant of naming abstract memory locations, has the constraint that the points-to set of the

| Statement | Set Constraint | Type |
|---|---|---|
| *i: x = alloc()* | $o_i \in pts(x)$ | Base |
| $x = \&y$ | $loc_y \in pts(x)$ | Reference |
| $x = y$ | $pts(y) \subseteq pts(x)$ | Assign |
| $x = {}^*y$ | $\forall y_p \in pts(y) : pts(y_p) \subseteq pts(x)$ | Load |
| ${}^*x = y$ | $\forall x_p \in pts(x) : pts(y) \subseteq pts(x_p)$ | Store |

Table 2.1: Statements needed for C code analysis

assigned variable, $x$ in this case, has to be extended by a new abstract location identified through the label of the *Base* type instruction. In a *Reference* the points-to set of $x$ gets extended by the storage location of the right-hand side variable. The constraint of an *Assign* states that every abstract location that is pointed-to by the right-hand side variable $y$ is also pointed-to by the left-hand side variable $x$. A *Load* is only then of interest if the right-hand side $y$ points-to a variable $y_p$ whose points-to set then will be included into the points-to set of the left-hand side variable. Similar reasoning applies to the constraint of a *Store*.

### 2.4.3 Flow-Sensitivity

Standard *flow-sensitive* points-to analysis, formally introduced in [14], is embedded into the classic iterative dataflow analysis framework and has every node $k$ in the CFG associated with the following pair of functions used to compute its output set of points-to sets $OUT_k$ from its input set of points-to sets $IN_k$:

$$IN_k = \bigcup_{x \in pred(k)} OUT_x$$
$$OUT_k = GEN_k \bigcup (IN_k - KILL_k)$$

$GEN_k$ and $KILL_k$ are sets of pointer information generated and killed, respectively, by CFG node $k$ and the analysis iteratively computes the above two functions for all nodes in the CFG until convergence is reached, i.e. for all CFG nodes $k$, $IN_k$ and $OUT_k$ do not change anymore.

By iteratively computing the points-to sets according to the above two functions, pointer information generated by CFG node $k_0$ is only considered by CFG node $k_1$ if $k_1$ is reachable from $k_0$ in the CFG. Thus the execution order of program statements is respected and precision of the points-to sets is increased.

However, the iterative computation of the above two dataflow equations makes the *flow-sensitive* approach both time-consuming as well as memory-

consuming since no knowledge about variable definitions and usages are available so far and the analysis must therefore propagate all points-to information available in one node to all of its successors.

To conclude this sub-chapter, the difference in precision of the computed points-to sets when either a *flow-insensitive* or a *flow-sensitive* analysis is applied to the running example, introduced in listing 2.2, is illustrated in listings 2.4 and 2.5 below.

$$
\begin{aligned}
pts(A) = pts(tmp4) &= \{\, alloc_A \,\} \\
pts(B) = pts(tmp5) &= \{\, alloc_B \,\} \\
pts(a) = pts(tmp8) = pts(p) &= \{\, alloc_a \,\} \\
pts(b) = pts(tmp9) = pts(q) &= \{\, alloc_b \,\} \\
pts(i) &= \{\, alloc_i \,\} \\
pts(tmp0) = pts(tmp1) &= \{\, alloc_A, alloc_B \,\} \\
pts(tmp7) = pts(10) = pts(tmp11) &= \{\, alloc_A, alloc_B \,\}
\end{aligned}
$$

**Listing 2.4:** Flow-Insensitive Analysis of Running Example

$$
\begin{aligned}
pts(A) = pts(tmp4) = pts(tmp0) = pts(tmp7) &= \{\, alloc_A \,\} \\
pts(B) = pts(tmp5) = tmp(1) = tmp(10) &= \{\, alloc_B \,\} \\
pts(a) = pts(tmp8) = pts(p) &= \{\, alloc_a \,\} \\
pts(b) = pts(tmp9) = pts(q) &= \{\, alloc_b \,\} \\
pts(i) &= \{\, alloc_i \,\} \\
pts(tmp11) &= \{\, alloc_A, alloc_B \,\}
\end{aligned}
$$

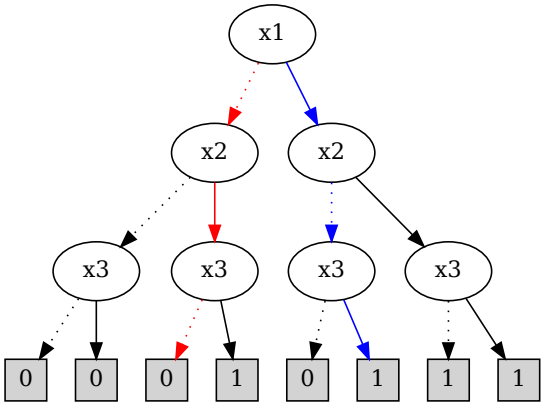
**Listing 2.5:** Flow-Sensitive Analysis of Running Example

## 2.5 Binary Decision Diagrams

A Binary Decision Diagram (BDD), introduced in [15] and extensively studied in [16][17], represents a Boolean function $f : \{\, 0,1 \,\}^n \rightarrow \{\, 0,1 \,\}$ as a rooted directed acyclic graph where the terminal nodes of out-degree zero are labeled 0 or 1 and where every non-terminal node is associated with a specific index into the $n$-tuple representation of the domain of $f$ and has an out-degree of two, with successors called 0-*successor* and 1-*successor*.

As $f$ indirectly defines a specific set of binary strings of length $n$, the set $\{ x \in \{ 0,1 \}^n \mid f(x) = 1 \}$, a BDD therefore can equivalently be thought of as being a representation of a set. The membership of $x$ in the set defined by $f$ can be determined by starting at the root of the BDD representing $f$ and if the bit of $x$ at the index position associated with the root is 0 the process proceeds to the 0-*successor*, else it proceeds to the 1-*successor*. Recursively traversing the BDD in this manner eventually leads to reaching a terminal node whose label indicates the set-membership of $x$.

In order to illustrate the concepts introduced above, consider the Boolean function $f : \{ 0,1 \}^3 \to \{ 0,1 \}$ naturally defined through a propositional logic sentence as $f(x_1, x_2, x_3) \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. One possible BDD for $f$ is depicted in figure 2.6.

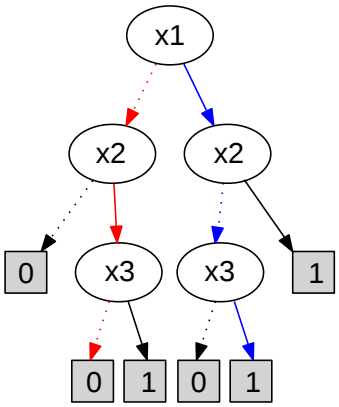



**Listing 2.6:** Maximally sized BDD for $f$

Dashed lines in listing 2.6 indicate edges leading to the 0-*successor* while full lines are the directed edges leading to the 1-*successor*. The path in red leading from the root to the third terminal node from the left is the path we would follow when tracking the set membership of $x = (0,1,0)$ and the terminal node label 0 at the end of this path indicates that $x$ is not a member of the set defined by $f$. The blue path is the path for tracking $x = (1,0,1)$ and by the label of the terminal node at the end of the path we see that $x$ is a member of the set defined by $f$ (and therefore a satisfying assignment of the propositional logic sentence defining $f$).

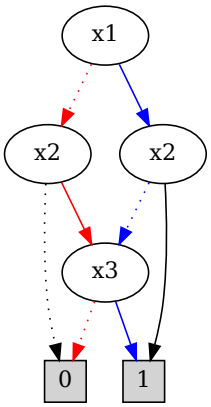
If we take a second look at the BDD in listing 2.6 we see that some branches are of no importance when deciding a set membership. Consider $x = (0,0,1)$. Starting from the root we follow the dashed arrow to the left $x2$. As at $x2$ we once more take the dashed arrow all the subsequent paths will lead to a terminal node labeled 0. This means that after evaluating the first two bits of $x$ we already know that $x$ is not a member of the set defined by $f$.

The same reasoning applies for the outermost path to the right in the BDD, with the difference to leading to a terminal node labeled with 1 than with 0. The BDD from listing 2.6 therefore can be transformed into a smaller graph, with respect to the number of nodes and edges it contains, while still representing the same function $f$. The BDD resulting from the considerations above is depicted in listing 2.7.




**Listing 2.7:** Simplified BDD for $f$

Even though the size of the BDD is already reduced a great deal there is still some potential for further reductions. To see this, note that in listing 2.7 the subgraphs rooted at both nodes labeled with $x_3$ are isomorphic. If we redirect all incoming edges of the left subgraph rooting in $x_3$ to the right subgraph rooting in $x_3$ we can completely remove the left subgraph while still keeping the BDD representing the original function $f$. The resulting BDD after those transformations were made and after all terminal nodes with the same label were unified into one node is depicted in listing 2.8.




**Listing 2.8:** Fully reduced BDD for $f$

The BDD presented in listing 2.8 is reduced in the sense that no two distinct nodes in the graph associated with the same domain-tuple index have equal 0-*successors* and 1-*successors* and furthermore no node in the graph has a 0-*successor* identical with its 1-*successor*. A BDD satisfying those properties is called a Reduced Binary Decision Diagram (RBDD). Another property the RBDD of listing 2.8 satisfies is that all paths from the root to the terminals respect the order $x_1 < x_2 < x_3$. A BDD satisfying this property is called an Ordered Binary Decision Diagram (OBDD). Therefore, the BDD of listing 2.8 really is a Reduced Ordered Binary Decision Diagram (ROBDD).

Stated next is the fundamental key property about the canonicity of ROBDDs.

**Theorem 2.15** For any Boolean function $f : \{0,1\}^n \to \{0,1\}$ there is exactly one ROBDD with variable ordering $x_1 < \cdots < x_n$ such that the Boolean function represented by the ROBDD is $f$.

**Proof** The proof is by induction on the number of arguments of $f$ and can be found in [16]. □

Two more properties about ROBDDs are stated as short remarks with references to more detailed literature describing each remark in more depth:

- ROBDDs are highly sensitive to the linear order imposed on the variables. Depending on the order, the size of a ROBDD can range from linear in the number of variables to exponential [16][17].
- Algorithms performing logical operations (such as $\wedge$ or $\vee$) on two ROBDD have been devised with runtime complexities only depending on the sizes of the ROBDDs involved and not on the number of variables making up the represented Boolean expressions [16].

As the second remark above indicates, ROBDDs not only represent propositional logic sentences but can also be handled and manipulated as if they indeed were.

The next two sections will show how sets and relations can be modeled in propositional logic and will allow to conclude that ROBDDs can be used to represent these mathematical concepts.

### 2.5.1 Representing Mathematical Sets

Let the universe of discourse be $U = \{0,1\}^n$. A set $S \subseteq U$ is denoted by its *characteristic Boolean function* $\chi_S : \{0,1\}^n \to \{0,1\}$, expressed as a propositional logic sentence, with $x \in S \Leftrightarrow \chi_S(x) \coloneqq \bigvee_{s \in S} \bigwedge_{1 \leq i \leq n} x_i \overline{\oplus} s_i = 1$, where $\overline{\oplus}$ represents the complement of the *Exclusive-Or* operation. Now, set operations can be implemented through the *characteristic Boolean functions* as follows:

- $\chi_{S \cup T} = \chi_S \vee \chi_T$
- $\chi_{S \cap T} = \chi_S \wedge \chi_T$
- $\chi_{S \setminus T} = \chi_S \wedge \overline{\chi_T}$

### 2.5.2 Representing Mathematical Relations

Again, we assume the universe of discourse to be $U = \{0,1\}^n$. As a *k-arry* relation is defined as a subset of $U_1 \times \cdots \times U_k$, a relation is basically just a set of *k-tuples*. Thus, representing relations can also be done using *characteristic Boolean functions*. To show an example representation, consider $R \subseteq U_1 \times U_2$. $R$ has the following *characteristic function*:

$$\chi_R(x,y) = \bigvee_{u_1 \in U_1} \bigvee_{\substack{u_2 \in U_2 \\ (u_1,u_2) \in R}} \left( \left( \bigwedge_{1 \leq i \leq n} x_i \overline{\oplus} u_{1_i} \right) \wedge \left( \bigwedge_{1 \leq i \leq n} y_i \overline{\oplus} u_{2_i} \right) \right)$$

For the remainder of this thesis it holds that whenever we speak about BDDs we actually refer to ROBDDs.

Chapter 3

# Methodology of the Analysis

The analysis presented in [1] proposes several new methods for flow-sensitive points-to analysis; Sparse analysis and sparse iterative dataflow computation is enabled by summarizing all definition-usage information of pointer variables in a Value Flow Graph (VFG). Flow-sensitivity is guaranteed by ensuring that only those variable definitions can flow to a usage in the VFG whose definitions can reach the usage in the CFG without being overwritten by another definition. The flow-sensitive points-to analysis then is boiled down to a reachability problem in the VFG.

On all of this as well as on the challenges imposed by requiring precision and completeness of the points-to sets is detailed in the following chapters.

## 3.1 Value Flow Graph

[1] uses a VFG to summarize all the definition-usage information of pointer variables assembled during the execution of the points-to analysis. Nodes in the VFG represent pointer variables and memory locations while edges between them represent flows of addresses. After the points-to analysis converged to a final result, the points-to set of a pointer variable is the set of memory objects that can reach the pointer in the VFG. The basic rules to build up the VFG are summarized in table 3.1.

| IR Instruction | Type | Value Flow | Edge Type |
|---|---|---|---|
| $x = alloc\_o$ | Base | $alloc\_o \rightarrow x$ | Direct |
| $x = y$ | Assign | $y \rightarrow x$ | |
| *store a, x* | Store | $a \rightarrow$ *store a, x* | |
| *store a, x*<br>$b = load\ y$ | Store<br>Load | $\exists\ alloc\_o : alloc\_o \rightarrow x \wedge alloc\_o \rightarrow y$<br>$\Rightarrow$ *store a, x* $\rightarrow b = load\ y$ | Indirect |

Table 3.1: Basic rules to build up the VFG

The fourth construction rule in table 3.1 highlights a key property of the analysis [1]. It states that variable $a$ can flow to variable $b$ indirectly via pointer dereferences only if $a$ is stored to a memory object first and $b$ is loaded from the same memory object afterwards.

By summarizing all the pointer information in the VFG according to the rules introduced, the time-consuming and memory-consuming propagation of this pointer information in the CFG, as required by traditional flow-sensitive approaches, can be prevented.

The points-to analysis then terminates if no more direct or indirect flows are left to be added to the VFG.

## 3.2 Flow-Sensitivity

In order to get a flow-sensitive analysis, the value flows need to be computed with respect to the execution order of the program instructions. Due to the properties of partial SSA we can insert all direct value flows resulting from Base instructions and Assignment instructions into the VFG without further ado. Inserting indirect value flows into the VFG, connecting Stores with Loads, is more involved. First, both instruction targets must point to a same memory object. Second, the Load must be reachable from the Store in the CFG. Computing the reachability becomes challenging when *strong updates* are involved where a Store instruction can be *killed* by another Store and the killed Store cannot flow further to any Load. The *strong update* rule is defined as follows:

**Definition 3.1** Let the Store instruction $S = store\ a,\ x$ be given. If $x$ points to one single memory location $alloc\_o$ then all previous Store instructions to $alloc\_o$ will be *killed* by $S$.

Therefore, a Load is only reachable from a Store if this Store can reach the Load in the CFG without being *killed*.

## 3.3 Challenges

As described above, the analysis summarizes pointer information in a VFG. During the progress of the analysis, this VFG is dynamically updated according to the rules in table 3.1. However, as the pointer information at this stage is still incomplete, indirect edges might falsely be inserted, leading to imprecise points-to sets, or indirect edges might falsely be not inserted, leading to incomplete points-to sets.

### 3.3.1 Incomplete Points-To Sets

Incomplete points-to sets result from Stores being considered *strong updates* while actually being not and therefore possibly preventing another Store to reach a Load. As no indirect edge will be added to the VFG, the points-to set of the variable defined through the Load will be incomplete.

This can be handled by updating and recomputing the indirect flows of the memory object as soon as a Store referring to it no longer is a strong update.

### 3.3.2 Imprecise Points-To Sets

Imprecision of the points-to sets is introduced if an indirect edge is inserted into the VFG even if the Store, for which the indirect edge was inserted, cannot actually reach the Load when reachability under the *strong update* rule is considered. This happens if a Store in between the original one and the Load is not known to point to the same memory location but the Store really would be a strong update on this location. The first Store then falsely reaches the Load while, if all pointer information was available, actually being killed by the second one.

An easy way to deal with imprecision is to conservatively assume that every Store with unknown points-to information kills all other Stores. However, this solution is computationally inefficient as every memory object could have its indirect flows recomputed as soon as a Store with previously unknown pointer information gets updated.

A second way to deal with imprecision is to only compute the indirect flows of a memory object if its pointed-to-by set is complete. This is achieved by following an *escape order* $\lhd$ when computing the indirect flows.

**Definition 3.2** Memory object $alloc\_a$ escapes to memory object $alloc\_b$ if there exists an instruction *store* $X$, $Y$ with $alloc\_a \rightarrow X$ and $alloc\_b \rightarrow Y$, denoted as $alloc\_b \lhd alloc\_a$.

**Theorem 3.3** Precision is achieved by computing indirect flows of memory objects in escape order.

**Proof** Consider the memory object $M = alloc\_a$ and the set of memory objects $S = \{\, alloc\_o \mid alloc\_o \lhd M \,\}$. The indirect flow of $M$ is computed only if the indirect flows of all elements in $S$ were computed. The address of $M$ flows to other pointer variables only by direct flows or indirect flows of elements in $S$. Hence the pointed-to-by set of $M$ is complete and all strong update Stores will be recognized. □

If there still are Store instructions with unknown pointer information when computing the indirect flow of a memory object, which is possible because

computing the escape order itself requires pointer information and this information might be incomplete during the analysis, we can always fall back to assuming the Store to kill all other Stores and precision is guaranteed.

Chapter 4

# Implementation of the Analysis

This chapter starts with first introducing the notation used later on and then proceeds to algorithmically describing the implementation of the analysis [1]. The pseudo-codes used for the description are biased towards the actual implementation of analysis [1] into LLVM [2] and are therefore not completely compiler and architecture agnostic. However, the LLVM specific parts are rare in number and marked so porting the algorithms to a different setting should be doable without much difficulty. To ease picturing how the analysis works, much of the algorithmic description will be accompanied by illustrations based on the example presented in listing 2.1. The last section of this chapter then covers technical details about how this thesis integrated the analysis [1] into LLVM and how BDDs were integrated into the analysis implementation.

## 4.1 Notation

The analysis [1] has for every memory object *alloc_o* a set of pointer variables pointing to it associated. This is the *pointed-to-by* set and is denoted as *pted(alloc_o)*. Furthermore, every memory object *alloc_o* has a set of memory objects associated to which *alloc_o* can escape to. This set is denoted as *esp(alloc_o)*. Finally, every pointer variable *p* has a set of memory objects associated to which *p* can point to. This is the *points-to* set of *p* and is denoted as *pts(p)*.

## 4.2 The Big Picture

Algorithm 1 presented below depicts a high-level view into the internals of the implementation of the SSA based, VFG using and flow-sensitive points-to analysis of [1].

**Algorithm 1** SSA based and VFG using Points-To Analysis - Big Picture

```
1: procedure BUILDPOINTSTOVFG
2:     processedList = ∅
3:     waitList = ∅
4:     updateList = { all Memory Objects }
5:     DIRECTVALUEFLOWCOMPUTATION()
6:     while updateList ≠ ∅ ∨ waitList ≠ ∅ do
7:         for every object alloc_o in updateList do
8:             COMPUTEPTEDSET(alloc_o)
9:         end for
10:        // Addressing the Incompleteness Challenge
11:        for every Store S not being a strong update anymore do
12:            updateList = updateList ∪ pts(S)
13:        end for
14:        for every object alloc_o in updateList do
15:            COMPUTEESPSET(alloc_o)
16:        end for
17:        updated = false
18:        toUpdateList = ∅
19:        // Try Computing the Indirect Flows in Escape Order
20:        for every object alloc_o in updateList do
21:            updateList = updateList \ { alloc_o }
22:            if esp(alloc_o) ∩ ( updateList ∪ waitList ) ≠ ∅ then
23:                waitList = waitList ∪ { alloc_o }
24:            else
25:                updated = true
26:                processedList = processedList ∪ { alloc_o }
27:                INDIRECTVALUEFLOWCOMPUTATION(alloc_o, toUpdateList)
28:            end if
29:        end for
30:        if updated = false then
31:            // No Flows Computed. Handle non-scalar Objects first.
32:            for every non-scalar object alloc_o in waitList do
33:                updated = true
34:                processedList = processedList ∪ { alloc_o }
35:                waitList = waitList \ { alloc_o }
36:                INDIRECTVALUEFLOWCOMPUTATION(alloc_o, toUpdateList)
37:            end for
38:        end if
39:        if updated = false then
40:            // Still No Flows Computed. Handle one scalar Object.
41:            for one scalar object alloc_o in waitList do
42:                waitList = waitList \ { alloc_o }
43:                INDIRECTVALUEFLOWCOMPUTATION(alloc_o, toUpdateList)
44:            end for
45:        end if
46:        updateList = updateList ∪ toUpdateList
47:    end while
48: end procedure
```

The lists of memory objects declared and initialized in lines 2 to 4 are used to compute the indirect flows of memory objects in escape order. The *processedList* is a list of objects whose indirect flows have been computed, the *updateList* is a list of objects whose indirect flows are yet to be computed and the *waitList* is a list of objects whose indirect flows cannot be computed yet because they escape to objects whose indirect flows have not been computed yet. After line 5 initialized the VFG with all direct edges, the main loop is entered where indirect flows are computed until all memory objects were processed and the VFG is complete. In lines 7 to 9, every memory object has its *pted* set and *pts* set computed. Lines 11 to 13 address the incompleteness challenge discussed in chapter 3.3.1. Next, the set of memory objects that a memory object escapes to is computed in lines 14 to 16. In lines 20 to 29, all objects from the *updateList* next in the escape order have their indirect flows computed and are put into the *processedList* while all other objects not yet ready to have their indirect flows computed are put into the *waitList*. Lines 30 to 38 handle the case when no indirect flows were computed and, assuming that primarily non-scalar objects were responsible for circular escapes, compute all indirect flows for all non-scalar objects. If there nevertheless were only scalar objects in the *waitList* then lines 39 to 45 select one of those scalar objects and compute its indirect flows.

## 4.3 Direct Value Flow Computation

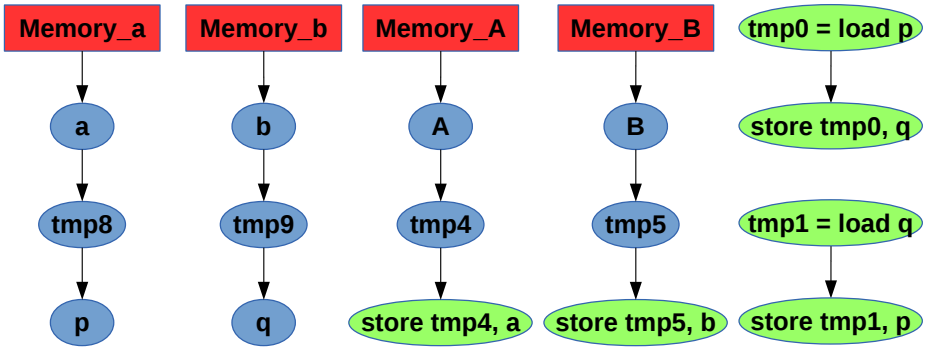
The VFG is initialized by inserting all edges resulting from direct value flows. Numerous code constructs can lead to a direct flow and how to extract them is shown in algorithm 2.

Lines 2 to 4 of algorithm 2 are LLVM specific and handle the fact that LLVM, as described in 2.3.1, turns every global variable into a pointer to the content of the original variable and allocates the space needed for this content in memory. As a result global variables can be handled as if they were mere allocation instructions. In lines 6 to 13, for every call site of every function the direct flows resulting from argument to function parameter passage are extracted as well as the direct flows from return statements in the function to the call site receiver variable. The rule found in lines 14 to 16 shows that the implementation conservatively assumes that every call to an external function with a pointer return type will lead the assigned variable, $r$ in this case, to point to one single dedicated memory object. Lines 17 to 21 show that every variable getting into a $\varphi$ node will flow to the one and same left-hand side variable. The subsequent rules concerning allocations, *Base* instructions, *Assign* instructions and *Store* instructions have already been introduced and explained in chapter 3.1. The last rule is again LLVM specific and shows that *bitcast* instructions can be handled like *Assign* instructions.

**Algorithm 2** Direct Value Flow Computation

1: **procedure** DIRECTVALUEFLOWCOMPUTATION
2: **for** every global variable $p$ **do** ▷ LLVM Specific
3: Add $o_p \rightarrow p$ to VFG
4: **end for**
5: **for** every function $f(p_1, ..., p_n)$ **do**
6: **for** every call site $l : r = f(a_1, ..., a_n)$ **do**
7: **for** every pair $(a_i, p_i)$ with $a_i$ a pointer **do**
8: Add $a_i \rightarrow p_i$ to VFG
9: **end for**
10: **for** every pointer type return variable $retV$ of $f$ **do**
11: Add $retV \rightarrow r$ to VFG
12: **end for**
13: **end for**
14: **for** every call $l : r = g(a_1, ..., a_m)$ in $f$ to external $g$ and $r$ a pointer **do**
15: Add $o_{Zero} \rightarrow r$ to VFG
16: **end for**
17: **for** every phi node $l : r = \varphi[r_1, ..., r_n]$ in $f$ with $r$ a pointer **do**
18: **for** every $r_i$ **do**
19: Add $r_i \rightarrow r$ to VFG
20: **end for**
21: **end for**
22: **for** any memory allocation $l : r = allocation(a_1, ..., a_m)$ in $f$ **do**
23: Add $o_l \rightarrow r$ to VFG
24: **end for**
25: **for** every *Base* instruction $l : p_0 = \&p_1$ in $f$ **do**
26: Add $p_1 \rightarrow p_0$ to VFG
27: **end for**
28: **for** every *Assign* instruction $l : p_0 = p_1$ in $f$ **do**
29: Add $p_1 \rightarrow p_0$ to VFG
30: **end for**
31: **for** every *Store* instruction $l : *p_0 = p_1$ in $f$ with $p_1$ a pointer **do**
32: Add $p_1 \rightarrow p_0$ to VFG
33: **end for**
34: **for** every $l : p_0 = bitcast\ p_1$ in $f$ with $p_0, p_1$ pointers **do** ▷ LLVM Specific
35: Add $p_1 \rightarrow p_0$ to VFG
36: **end for**
37: **end for**
38: **end procedure**

The VFG resulting from the application of algorithm 2 to the running example introduced in listing 2.1 is depicted in listing 4.1.




**Listing 4.1:** VFG of running Example after Initialization

## 4.4 Value Flow Graph Traversal

As explained in chapter 3.1, the VFG is used to summarize all the pointer information computed so far. The analysis itself makes use of this information when computing the *pted*, *pts* and *esp* sets, as done in lines 8 and 15 of algorithm 1. How the implementation computes the *pted* sets and *pts* sets is shown in algorithm 3.

**Algorithm 3** Computing Pted and Pts Sets

```
1: procedure computePtedSet(alloc_o)
2:     pts(alloc_o) = ∅
3:     pted(alloc_o) = ∅
4:     Let P be all VFG nodes reachable from alloc_o
5:     for every node p in P do
6:         pted(alloc_o) = pted(alloc_o) ∪ { p }
7:         pts(p) = pts(p) ∪ { alloc_o }
8:     end for
9: end procedure
```

Computing all reachable VFG nodes, as required in line 4 of algorithm 3, can be done through a simple breadth-first search in the VFG, starting from the memory location node $alloc_o$ in this case. By definition, every node p reachable from $alloc_o$ is in the *pted* set of $alloc_o$ and on the other hand $alloc_o$ is in the *pts* set of p. This is reflected in lines 6 and 7 of algorithm 3.

How the *esp* sets are computed is shown in algorithm 4. If $alloc_o$ can flow to a Store then $alloc_o$ can escape to all memory objects where the Store's desti-

**Algorithm 4** Computing Esp Sets

```
1:  procedure computeEspSet(alloc_o)
2:      esp(alloc_o) = ∅
3:      for every Store p in pted(alloc_o) with d as the destination of p do
4:          if pts(d) = ∅ then
5:              esp(alloc_o) = esp(alloc_o) ∪ { allMemoryObjects }
6:          else
7:              esp(alloc_o) = esp(alloc_o) ∪ pts(d)
8:          end if
9:      end for
10: end procedure
```

nation pointer points to. This reasoning is expressed in line 7 of algorithm 4. However, if it's not known where the destination pointer of the Store points to then the *esp* set of $alloc_o$ equals to all memory objects. This will subsequently be noted in line 22 of algorithm 1 and $alloc_o$ won't have its indirect flows computed.

## 4.5 Indirect Value Flow Computation

Computing the indirect value flows of a memory object happens in lines 27, 36 and 43 of algorithm 1 and its purpose is, as explained in chapter 3.2, to connect a Store with all Loads it can reach in the CFG without being *killed*.

The driver algorithm for computing the indirect value flows is shown in algorithm 5.

**Algorithm 5** Driver for Computing Indirect Value Flows

```
1:  procedure indirectValueFlowComputation(alloc_o, toUpdateList)
2:      indirectFlows = computeIndirectFlows(alloc_o)
3:      for every S → L ∈ indirectFlows do
4:          if S → L ∉ VFG then
5:              Add S → L to VFG
6:              toUpdateList = toUpdateList ∪ pts(S)
7:          end if
8:      end for
9:  end procedure
```

First, algorithm 5 computes all indirect flows, line 2, and subsequently augments the VFG with all newly computed flows. This changes the pointed-to-by sets of all memory objects in the points-to set of the Store that was added and therefore those memory objects are added to the toUpdateList, whose

elements are later on added to the updateList of algorithm 1.

The actual computation of the indirect flows happens in line 2 and the corresponding procedure to compute them is shown in algorithm 6.

**Algorithm 6** Compute Indirect Value Flows

```
1: procedure computeIndirectFlows(alloc_o)
2:     indirectFlows = ∅
3:     preprocessGlobalFlow(alloc_o)
4:     for every function f referencing alloc_o do
5:         indirectFlows = indirectFlows ∪ computeLocalFlow(f, alloc_o)
6:     end for
7:     return indirectFlows
8: end procedure
```

In line 3 of algorithm 6, auxiliary instructions are added in order to handle inter-procedural value flow. Lines 4 to 6 then subsequently compute the indirect value flows inside each function referencing the memory object $alloc_o$. The functions referencing $alloc_o$, required to be known by line 4, are the encompassing functions of all Store and Load instructions in the pointed-to-by set of $alloc_o$. Both procedures called by algorithm 6 are describe in detail in the following two sub-chapters.

### 4.5.1 Preprocessing the Global Flow

Inter-procedural indirect value flow is modeled as value flow between auxiliary variables inserted for every argument-to-parameter passage where the memory object under consideration is involved. This will subsequently allow the indirect flows to be computed through a reachability analysis in a sparse version of the CFGs of every function referencing the said memory object. The procedure for inserting the auxiliary instructions is depicted in algorithm 7.

A Store instruction S in the caller function can flow to a Load instruction L in the callee function if S can reach the call site and if L is reachable from the entry of the callee function. With the four instructions added in lines 6 to 9 of algorithm 7 this amounts to determining whether S can reach *ref.a = load a* and whether *store ref.a, p* can reach L. It should be noted that according to row 3 in table 3.1, the four instructions added introduce two additional direct edges into the VFG: *ref.a = load a* $\rightarrow$ *store ref.a, p* as well as *ref.p = load p* $\rightarrow$ *store ref.p, a*.

The resulting CFGs of preprocessing the running example, shown in listing 2.1, for the memory objects $Memory_a$ and $Memory_b$ can be found in

**Algorithm 7** Preprocessing Global Flow by Inserting Auxiliary Instructions

```
1:  procedure PREPROCESSGLOBALFLOW(alloc_o)
2:      PREPROCESSGLOBALFLOW(alloc_o)
3:      for every argument-to-parameter passage a → p of alloc_o do
4:          Let c be the call site and f be the callee function
5:          Add the following 4 instructions:
6:              ref.a = load a      before call site c
7:              store ref.a, p      at the entry of f
8:              ref.p = load p      at the exit of f
9:              store ref.p, a      after call site c
10:     end for
11: end procedure
```

listings 4.2 and 4.3. The additional direct flows to be added to the initial VFG, shown in listing 4.1, can be found in listing 4.4.

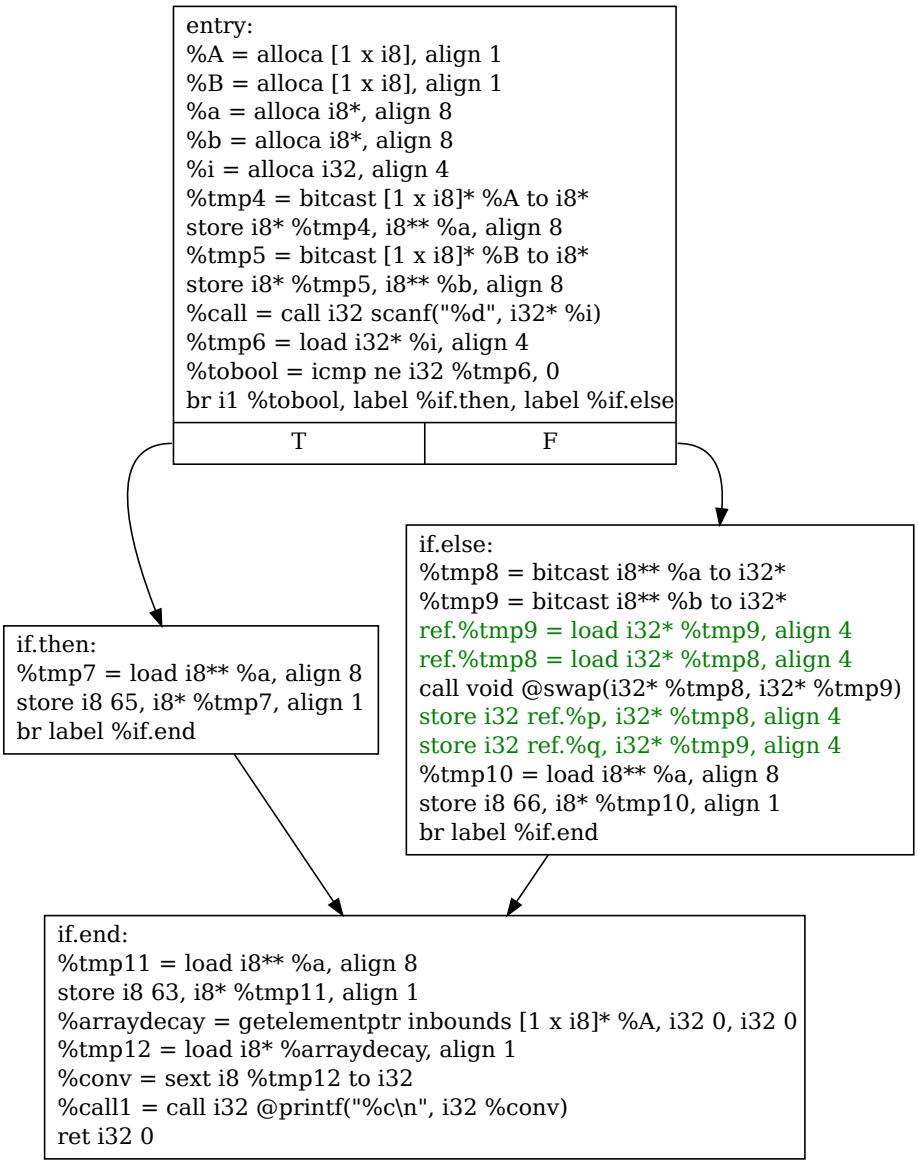
```
entry:
store i32 ref.%tmp9, i32* %q, align 4
store i32 ref.%tmp8, i32* %p, align 4
%tmp0 = load i32* %p, align 4
%tmp1 = load i32* %q, align 4
store i32 %tmp1, i32* %p, align 4
store i32 %tmp0, i32* %q, align 4
ref.%p = load i32* %p, align 4
ref.%q = load i32* %q, align 4
ret void
```

**Listing 4.2:** Preprocessed IR CFG of swap(int *p, int *q)

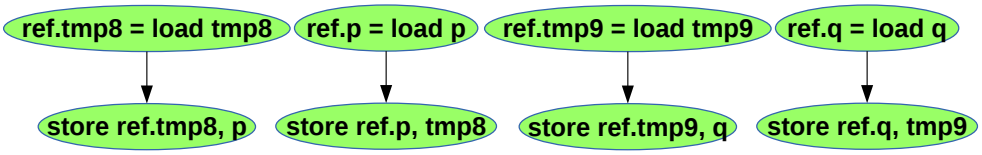
### 4.5.2 Local Flow Computation

Computing the indirect flows of a memory object inside a function referencing it is done, as described in algorithm 8, through a reachability analysis in a sparse version of the function's CFG.

As a first step, algorithm 8 computes in line 3 all Store instructions inside the given function f referencing the memory object under consideration. In line 5 the iterated dominance frontier of all basic blocks containing the Store instructions gathered before is computed. This is done because a Store instruction can reach a Load instruction in the CFG only if the Load is either dominated by the Store or the Load is dominated by an element of the iterated dominance frontier of the Store. Therefore, a subsequent reachability analysis does not need to work on the whole CFG of the function but just on the set of basic blocks containing the Stores, computed in line 4, unified




**Listing 4.3:** Preprocessed IR CFG of main()




**Listing 4.4:** Direct Edges resulting from Preprocessing the running Example

**Algorithm 8** Computing Function Local Flows

1: **procedure** COMPUTELOCALFLOW(f, $alloc_o$)
2: localIndirectFlows $= \emptyset$
3: Let S be the set of Stores in f referring to memory object $alloc_o$
4: Let SB be the set of basic blocks encompassing all Stores in S
5: Let IDF be the iterated dominance frontier of SB in f
6: Build a Sparse CFG G = (N, E) of f as follows:
7: N = { SB ∪ IDF }
8: $(n_0, n_1) \in$ E iff $n_1 \in DF(n_0)$ or $IDOM(n_1) = n_0$
9: Solve the following data flow equations in G:
10:

$$IN_k = \bigcup_{x \in pred(k)} OUT_x$$

$$OUT_k = GEN_k \bigcup (IN_k - KILL_k)$$

11: with $k \in$ N and $IN_k$ and $OUT_k$ sets of Store instructions
12: Let L be the set of Loads in f referring to memory object $alloc_o$
13: **for** every Load l ∈ L **do**
14: Let B be the basic block encompassing l
15: Let $B^+$ be the most immediate dominator of B in G
16: Let S be those Stores of $OUT_{B^+}$ reaching l in B
17: **for** every Store s ∈ S **do**
18: localIndirectFlows = localIndirectFlows ∪ { (s,l) }
19: **end for**
20: **end for**
21: **return** localIndirectFlows
22: **end procedure**

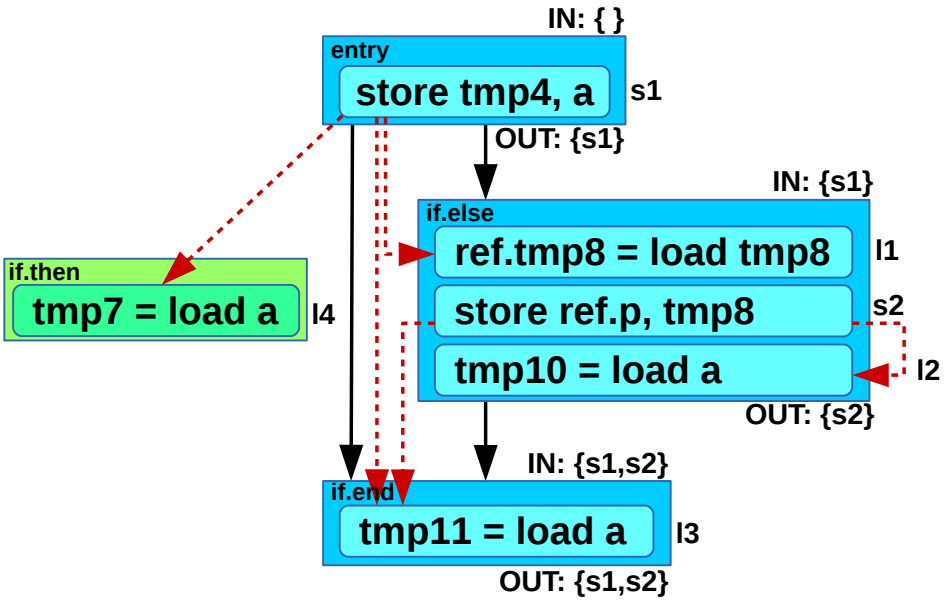
with the iterated dominance frontier thereof. According to the previous reasonings a sparse CFG is built in lines 6 to 8 and the data flow equations introduced in chapter 2.4.3 then have to be solved in it. The IN and OUT sets associated with each basic block in the data flow equations are sets of Store instructions. A Store instruction coming into a basic block is killed by it if the basic block contains a strong update Store on the memory location targeted by the incoming Store instruction. After the data flow equations have been computed until a fix-point was reached, all Load instructions referring to the memory object under consideration will be computed in line 12. Then, starting in line 13, algorithm 8 computes for all the Load instructions referring to the memory object under consideration all Store instructions that can reach it. In order to do this, every Load instruction first has

in lines 14 and 15 the most immediate dominator of its encompassing basic block computed. The set of outgoing Store instructions associated with the most immediate dominator basic block then represents all these Stores that can reach the encompassing basic block of the Load instruction. This set of incoming Store instructions then has to be filtered, done in line 16, such that only those Stores that can actually reach the Load instruction remain. Every remaining Store instruction then constitutes the source of a new local indirect flow to be added to the VFG.

In order to demonstrate how algorithm 8 works we apply it to the running example introduced in listing 2.1. First however, we have to consider the pointer information stored in the VFG after the initialization took place, which is found in listing 4.1. This information tells us that the memory objects $Memory_a$ and $Memory_b$ don't escape to any other objects since their addresses do not flow to any Store instructions. On the other hand, as there is a flow of addresses from $Memory_A$ to a Store instruction with a destination pointer pointing to $Memory_a$, $Memory_A$ escapes to $Memory_a$. Analogous, $Memory_B$ escapes to $Memory_b$. As indirect flows are computed in escape order, see chapter 3.3.2 for details, the indirect flows of memory objects $Memory_a$ and $Memory_b$ are computed first, the order between the two of them is irrelevant (we will start with $Memory_a$), and in a second step the indirect flows of objects $Memory_A$ and $Memory_B$ will be computed.

Computing the indirect flow of $Memory_a$ first leads to a preprocessing of the global flows involved, see line 3 of algorithm 6. The result of this are the two left most direct edges depicted in listing 4.4. As both the main function as well as the swap function of the running example have instructions referencing $Memory_a$, the algorithm then continues to compute the local flows for both of these functions through a call to algorithm 8. The final result of calling algorithm 8 on the main function and for the memory object $Memory_a$ is depicted in listing 4.5.

Listing 4.5 contains an assembly of blue blocks connected with solid black arrows representing the sparse CFG computed in lines 6 to 8 of algorithm 8. As the entry basic block and if-else basic block belong to the set of basic blocks computed in line 4 of algorithm 8 and basic block if-end is an element of the iterated dominance frontier computed in line 5 those are the three basic blocks making up the sparse CFG. The edge from the entry to the if-else block exists because the entry block is the immediate dominator of the if-else block. The edge from the entry to the if-end block exists because the if-end block is an element of the dominance frontier of the entry block and the edge from the if-else block to the if-end block exists because the if-end block is also an element of the dominance frontier of the if-else block. Furthermore, a green block representing a basic block from the original non-sparse CFG of the main function was inserted into the listing. Note that it was only inserted

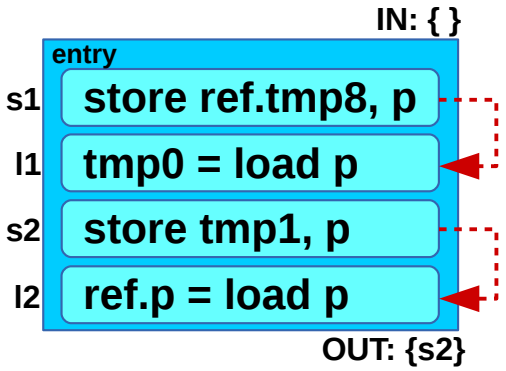



**Listing 4.5:** All indirect flows of $Memory_a$ inside function main

into the listing for demonstration purposes and is not present in the real sparse CFG. Additionally, the listing shows all Load and Store instructions referring to $Memory_a$ and the results of computing the data flow equations are found in the IN and OUT sets attached to each sparse CFG basic block. Now, determining the Store instructions that can reach the Load instruction l1 goes as follows: the encompassing basic block of the Load instruction l1 is the if-else block whose most immediate dominator is the entry block. The outset of the entry block now constitutes the set of Store instructions that can reach the if-else basic block. As the Load instruction l1 is the first instruction in the basic block if-else, all Store instructions from the OUT set of the entry block can reach l1. Therefore, an indirect flow from s1 to l1 exists. In order to determine the Store instructions that can reach the Load instruction l2 we again proceed by first finding the OUT set of the most immediate dominator of the basic block encompassing l2. This is again the OUT set of the entry block. Then, as this OUT set only constitutes the set of Store instructions that can reach the basic block if-else but not necessarily the set of Store instructions that can reach the Load, we have to analyze all Store instructions before l2 in the basic block if-else and check whether they are strong updates on any of the Stores coming into the basic block. And indeed, in this example Store s2 is a strong update on $Memory_a$ thereby preventing the Store s1 from being propagated any further. As a result, not the Store s1 reaches the Load l2, but s2. Due to this, it's also Store s2 that will flow to the end of the basic block if-else and thereby being the only Store in the OUT set of the if-else block. The indirect flows from s1 to l3

and from s2 to l3 are computed according to an analogous reasoning. In order to find the indirect flow from Store s1 to Load l4, algorithm 8 makes use of the most immediate dominator concept as the if-then basic block is part of the original CFG but is not part of the sparse CFG. After the entry block has been determined as the most immediate dominator of the if-then block, extracting the indirect flow from s1 to l4 is done as in the other cases described above.

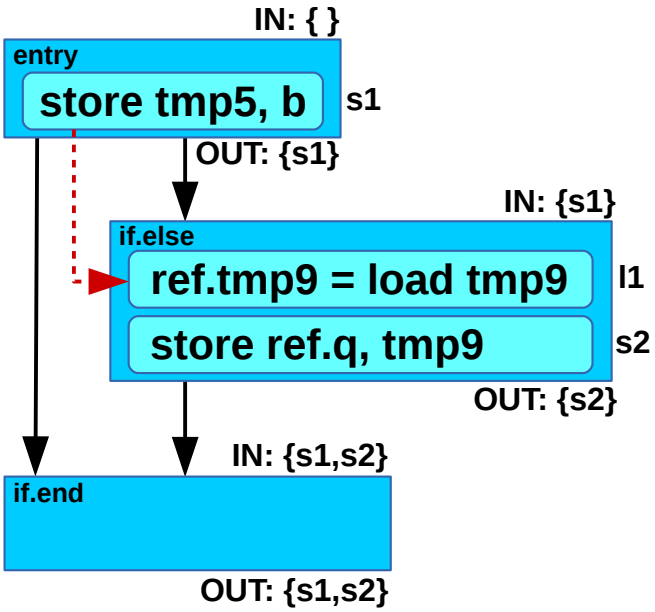
For the sake of completeness, the results of computing the local indirect flows of memory object $Memory_a$ in function swap as well as computing the local indirect flows of $Memory_b$ inside the main function and swap function are shown in listings 4.6, 4.7 and 4.8. These results are shown without any further comment as all the aspects and corner-cases of algorithm 8 were already met and discussed in the first example above.



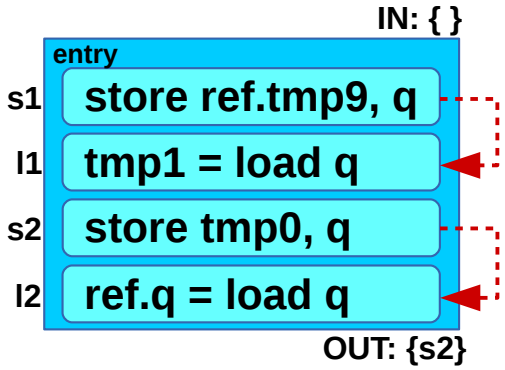

**Listing 4.6:** All indirect flows of $Memory_a$ inside function swap

The final VFG with all indirect flows of memory objects $Memory_a$ and $Memory_b$ inserted is depicted in listing 4.9.

This final VFG allows to deduce, for example, that variables tmp7 and tmp10 will *never* point to the same memory object, that tmp0 and tmp7 *always* point to the same memory object and that variable tmp7 and tmp11 *may* point to the same memory object. Applying these results to the C code version of the running example and knowing that the LLVM IR variable tmp11 contains the address of the memory location the store in line 19 of listing 2.1 targets, we can conclude that the character '?' can get written into either array A or array B and that the result of a subsequent read from array A, as occurring in line 20, therefore depends on the control flow path taken during the execution of the program.




**Listing 4.7:** All indirect flows of $Memory_b$ inside function main



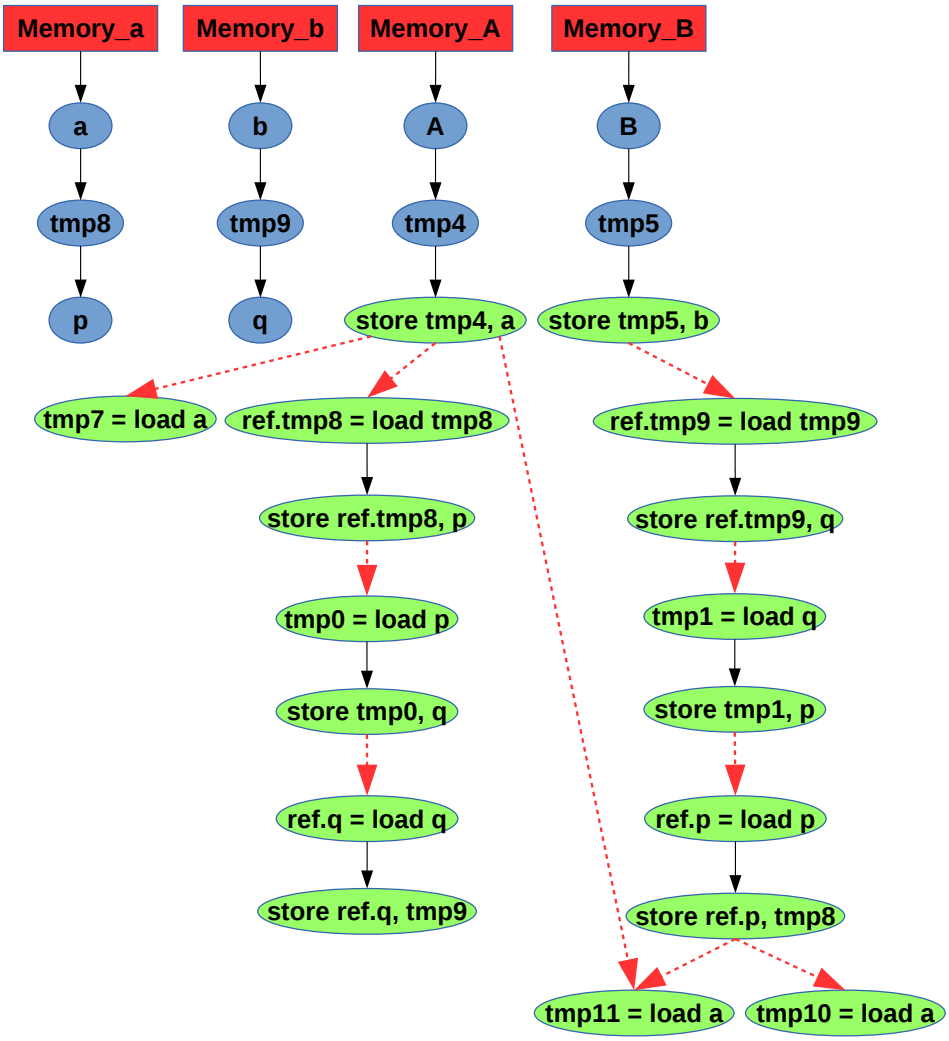

**Listing 4.8:** All indirect flows of $Memory_b$ inside function swap

## 4.6 Technical Details

### 4.6.1 Integration of Binary Decision Diagrams into the Analysis

This thesis made use of the publicly available BuDDy BDD library[1] [18] to represent the BDDs. As this thesis implemented the analysis [1] in a set representation agnostic manner with the only requirement an actual set implementation to provide an interface for standard operations on mathematical sets as well as the possibility to iterate over all the elements in the set, the BDDs from the BuDDy library were wrapped into a pointer data aggregate that provides this interface and then linked into the analysis.

[1]BuDDy: itu.dk/research/buddy/. Last accessed: September 29, 2014




**Listing 4.9:** Final VFG of running Example

### Representing the Sets

Generally, representing points-to sets can be done in various ways. Assume that variable v can point to variables x and y whereas variable w can point to variables x and z. One way to represent this information is to have one large set of ordered pairs, i.e. $\{(v,x),(v,y),(w,x),(w,z)\}$. Another way is to give each variable its own set of variables it can point to, i.e. $v \rightarrow \{x,y\}$ and $w \rightarrow \{x,z\}$. In terms of BDDs, the first case means to have one single BDD to represent all points-to information whereas in the second case each variable is given its own BDD. This thesis experimented with both versions and found a huge slowdown in the first case, resulting from the analysis algorithm often requiring the points-to set of one specific variable

to be erased. Doing this while having only one single large set for all points-to information requires every ordered pair in the set being examined and seeing whether the first element equals the variable for which the points-to set should be erased. If such a pair is found it has to be removed from the BDD while maintaining the canonicity property of the BDD. All this is a huge computational overhead compared to the second version where erasing the points-to information of a variable only amounts to resetting the BDD associated with it. Programming the characteristic functions of the sets themselves, as described in chapters 2.5.1 and 2.5.2, then could be done straightforwardly as BuDDy provides convenience wrappers for all propositional logic operations.

#### Listing the Elements of a Set

One difficulty in using BuDDy in a space critical environment is the retrieval of the elements stored in a BDD. BuDDy gives access to this information only via returning at once all satisfying assignments of the propositional logic sentence representing the characteristic function of the set under consideration. The returned information is an interleaving of necessary truth value assignments and don't care values resulting from the possibly high compression of logical sentences when brought into the canonical representation introduced in chapter 2.5. As an example, consider the set $S = \{ x \in \mathbb{N} \mid x \text{ odd} \wedge 0 < x < 100 \}$. BuDDy returns the satisfying assignments as the following sequence of strings: 1X000XX, 1X0010X, 1X00110, 1X01X0X, 1X01X10, 1X1XX0X, 1X1XX10. The encoding is simply the binary representation of each element in the sets with X being the don't care value meaning that it can be either 0 or 1. The first string for example represents the set $\{ 1, 3, 33, 35, 65, 67, 97, 99 \}$. A straightforward decoding of the strings returned leads to spoiling the space savings gained when using BDDs to represent sets. Repeatedly calling BuDDy to compute all satisfying assignments and picking just the next string not seen so far trades a lot of time efficiency for the space savings gained. The approach implemented in this thesis is to call BuDDy only once and store all the strings as returned, decoding them on the fly when iterating over the pointer data aggregate into which the BDD is wrapped.

### 4.6.2 Integration of the Analysis into LLVM

This thesis integrated the implementation of the analysis [1] into LLVM's Alias Analysis infrastructure[2]. In LLVM, every alias analysis must provide an interface for answering alias queries with answers of types *must* alias, *may* alias or *no* alias. Multiple alias analysis implementations can be chained

[2]LLVM Alias Analysis Infrastructure Documentation: llvm.org/docs/AliasAnalysis.html. Last accessed: September 29, 2014

together such that a first alias analysis can fall back on to another alias analysis in case the first analysis couldn't figure out an answer of type *must* or *no*. As all of the required types of answers can be given upon completion of analysis [1], integrating the implementation of [1] into LLVM was, thanks to the exhaustive documentation, straightforward.

Chapter 5

# Evaluation

This thesis set out to investigate the effect on the space and time requirements of an implementation of the points-to analysis algorithm [1] when Binary Decision Diagrams are used to represent the points-to information computed during the analysis. The implementation developed during this thesis is set representation agnostic and only expects an actual set implementation to deliver an interface for standard mathematical set operations like insertion and lookup. This allows to compare different representations with ease. For the purpose of this evaluation, the base set representation used to compare the performance of BDDs against was chosen to be the set implementation of the C++ Standard Template Library STL[1,2].

## 5.1 Environment and Setup

The implementation of analysis [1] was integrated into LLVM [2] version 3.5.0 and the BuDDy BDD library used had version number 2.4. The evaluation was conducted on a host with an Intel Core i7 2820QM 2.3GHz CPU, 8GB RAM and running Debian with kernel version 3.11.0. The general workload on the system was reduced to the minimum and the system process running the analysis was run with the highest CPU priority. As the space requirements of the analysis are assumed to be deterministic, getting the space consumption figures was done by running the analysis once on each benchmark. In order to also track the space consumption of analyses on benchmarks taking very long to complete, possibly even exceeding an arbitrarily set execution time limit, the space usage measurements were conducted at the end of the main loop of algorithm 1. This allowed to track

[1] C++ Standard Template Library STL: cplusplus.com/reference/clibrary/. Last accessed: September 29, 2014

[2] C++ STL Set Implementation: cplusplus.com/reference/set/set/. Last accessed: September 29, 2014

the last common point of both the BDD using and the set using version of an exceeding analysis where the information computed was equal and then to still compare the space consumption between both versions. Measuring the time requirements was done by executing the analysis on each benchmark 120 times and then averaging the execution times after the slowest and fastest 10 measurements were removed.

## 5.2 Benchmarks Used

The benchmarks used for the evaluation is a set of open-source C code applications with source code sizes ranging over several orders of magnitude. A description of the benchmarks can be found in table 5.1.

| Benchmark | Lines of Code | Description |
|---|---|---|
| miniz v1.15 | 4.9K | Data compression library |
| Mongoose v5.4 | 5.1K | Web server |
| bzip2 v1.0.2 | 7.0K | Data compression library |
| gzip v1.3.5 | 8.6K | Data compression library |
| oggenc v1.0.1 | 58.4K | Audio Codec |
| sqlite3 v3.8.5 | 147.8K | SQL database engine |
| gcc v4.0.0 | 753.8K | GNU Compiler Collection |

Table 5.1: Benchmarks used in the evaluation

## 5.3 Performance Results

Tables 5.2 and 5.3 list the spaces and times used by the implementation of the points-to analysis when C++ STL sets or BDDs were used, respectively. Table 5.4 lists some statistics of the VFGs involved in each benchmark analysis.

| Benchmarks | C++ STL Set | | | |
|---|---|---|---|---|
| | Space | | | Time |
| | Start | End | Delta | |
| miniz v1.15 | 73′428 | 75′008 | 1′580 | 0:00:73 |
| Mongoose v5.4 | 73′424 | 75′008 | 1′584 | 0:00:64 |
| bzip2 v1.0.2 | 74′924 | 75′980 | 1′056 | 0:00:53 |
| gzip v1.3.5 | 72′768 | 73′812 | 1′044 | 0:00:48 |
| oggenc v1.0.1 | 85′852 | 90′504 | 4′652 | 0:05:69 |
| sqlite3 v3.8.5 | 105′648 | 121′412 | 15′764 | 0:11:07 |
| gcc v4.0.0 | 475′764 | 2′764′488 | 2′288′724 | OOT |

Table 5.2: Performance Results for Sets. Unit of **Space**: kB. Format of **Time**: *min:sec:hsec*.

| Benchmarks | Binary Decision Diagrams | | | |
|---|---|---|---|---|
| | Space | | | Time |
| | Start | End | Delta | |
| miniz v1.15 | 73′432 | 75′724 | 2′292 | 0:00:83 |
| Mongoose v5.4 | 73′416 | 76′040 | 2′624 | 0:00:82 |
| bzip2 v1.0.2 | 74′924 | 77′144 | 2′220 | 0:00:59 |
| gzip v1.3.5 | 72′768 | 74′616 | 1′848 | 0:00:59 |
| oggenc v1.0.1 | 85′852 | 93′432 | 7′580 | 0:09:02 |
| sqlite3 v3.8.5 | 105′648 | 126′036 | 20′388 | 0:14:13 |
| gcc v4.0.0 | 475′764 | 1′264′548 | 788′784 | OOT |

Table 5.3: Performance Results for BDDs. Unit of **Space**: kB. Format of **Time**: *min:sec:hsec*.

| Benchmarks | Memory Objects | Initial VFG | | Final VFG | | |
|---|---|---|---|---|---|---|
| | | Nodes | Edges | Nodes | Edges | |
| | | | | | Direct | Indirect |
| miniz v1.15 | 133 | 3′193 | 2′145 | 7′467 | 2′337 | 3′712 |
| Mongoose v5.4 | 579 | 4′095 | 2′920 | 7′158 | 3′342 | 2′091 |
| bzip2 v1.0.2 | 243 | 3′972 | 2′650 | 6′190 | 2′790 | 1′919 |
| gzip v1.3.5 | 392 | 3′311 | 1′970 | 5′053 | 2′064 | 1′514 |
| oggenc v1.0.1 | 3′417 | 13′770 | 7′983 | 22′082 | 8′643 | 6′612 |
| sqlite3 v3.8.5 | 1′731 | 34′784 | 26′168 | 61′373 | 28′538 | 20′916 |
| gcc v4.0.0 | 17′592 | 510′821 | 361′098 | 5′456′911 | 414′016 | 4′822′455 |

Table 5.4: Statistics of the VFGs involved in the analyses

For each benchmark, the final analysis space consumption figure can be found in the End sub-column whereas the Start sub-column indicates the space consumption LLVM had just before it reached our analysis. Knowing both of these figures will allow us to subsequently discuss the space requirements for storing just the very information we produce inside our analysis. The upper execution limit for the analysis was set to be 8h. Benchmarks that exceeded this limit have an Out-of-Time OOT mark in their respective time column and the figures in the respective space columns represent the space consumption the analysis had at the last common execution point of both set representation versions.

## 5.4 Discussion

### 5.4.1 Space Usage

As seen in the performance summary tables 5.2 and 5.3, the space requirement of the analysis when using BDDs to represent points-to sets is for all but one benchmark higher than the space requirement of an analysis using

C++ STL sets. On these benchmarks, an average space consumption overhead of 65.0% occurred. The only benchmark where a BDD using analysis had reduced space requirements was *gcc*. For this benchmark a space consumption reduction of 65.6% was observed. As explained in chapter 5.1, this is still a representative figure even though the analysis time limit was exceeded and due to the runtime differences, as discussed below, both the BDD using and C++ STL set using versions of the analysis were very likely at different points in the progress of the analysis.

The space consumption figures found can be explained as follows. When BDDs are used, the analysis first has to initialize the BuDDy package with a number representing the maximum number of BDDs that could possibly be used subsequently. However, at the stage where this initialization has to take place, the analysis has not yet computed any pointer information and therefore has to assume that all eligible variables and abstract memory locations occurring in the source can be either part of a points-to set or have a points-to set attached to themselves and the total number of these objects is the number with which BuDDy is initialized. BuDDy then internally allocates a pool of BDDs to be used later on. The BDDs allocated in this initial pool are small BDDs representing *False* but as for small benchmarks the initial analysis memory consumption is relatively low, even a moderately sized initial pool of BDDs has a noticeable effect on the total space consumption. On the other hand, the larger a benchmark gets, like in the case of *gcc*, the size of this initial pool of BDDs only marginally contributes to the final space consumption figure of the analysis and the space gaining effects of BDDs can start to show off.

### 5.4.2 Time Usage

Tables 5.2 and 5.3 show that an analysis using BDDs to represent points-to sets experiences a runtime slowdown on all benchmarks when compared with an analysis using C++ STL sets. The average slowdown over all the benchmarks tested was 27.3%.

Profiling a BDD using analysis with *Valgrind*[3] exposes one major source of the runtime slowdown to be the retrieval of information stored in BDDs. In order to lookup the information stored in a BDD, the BuDDy BDD library needs to traverse the BDD underlying directed acyclic graph and has to find all paths leading from the root to a terminal-node labeled with 1. Those paths leading to terminal-nodes labeled with 1 are then returned by BuDDy as encoded satisfying assignments of the original propositional logic sentence, see chapter 4.6, and in order to get the real information stored in the BDD, the encoded satisfying assignments subsequently have to be decoded.

[3]Valgrind - Linux debugging and profiling tool: valgrind.org/. Last accessed: September 29, 2014

Altogether, retrieving the information stored in a BDD constitutes a computationally expensive task. This is especially true when compared with the retrieval of elements stored in a C++ STL set which (mostly) has a balanced binary search tree at its core and allows for an easy information lookup with a time linear in the number of elements stored in the set and a small constant factor. Places where lookups of elements stored in a points-to set happen are numerous in the analysis algorithms and, according to the results of the profiler, the computational expensiveness of this task indeed seems to sup up and constitute a major source of runtime performance loss. Another part where a BDD using analysis spends a large percentage of its runtime in is, as exposed by Valgrind, the BuDDy function responsible for performing logical operations on the BDDs involved which means to perform complex transformations on the graphs representing the BDDs in order to keep them in a canonical state. See chapter 2.5 for more about this. As the logical operations performed on the BDDs as well as the information lookup in the points-to sets are dictated by the logic and methodology of the analysis, there is not much room left for improving the runtime of a BDD using analysis with the current versions of the algorithms and the slowdown would need to be counter-fought with restructuring the analysis itself. One such possibility would be to turn the analysis into a *demand-driven* version and will shortly be discussed in chapter 6.

With regards to the analysis runtimes in general, table 5.4 allows to conclude that there is a positive correlation between the runtime needed to analyze a benchmark and the number of edges in the final VFG. On the other hand, the number of memory objects produced by a benchmark is not the primary factor determining the runtime of an analysis. This is well demonstrated by the benchmarks *oggenc* and *sqlite3* where half the amount of memory objects for *oggenc* still results in a doubled runtime when compared with *sqlite3*. Table 5.4 also shows the reason for the runtime to blow up when analyzing the largest benchmark *gcc* when compared with the reasonably fast analysis runtime for the second largest benchmark *sqlite3*: the VFG at the point the analysis was terminated already had roughly 90x more nodes stored and more than a hundredfold of value flows were discovered. Again those numbers are directly linked to the analysis methodology and the already mentioned *demand-driven* version, to be discussed in the next chapter, could possibly help to circumvent such a big blowup even when analyzing sources with sizes ranging over multiple magnitudes.

Chapter 6

# Conclusion and Future Work

In this thesis we implemented the points-to analysis presented in [1] into LLVM [2] and investigated whether representing points-to information through BDDs from the BuDDy library is a useful mean for tackling the space requirement deficiencies found for the original version of the analysis. The evaluation showed that if the source to be analyzed is of a large enough size then BDDs can indeed significantly reduce the space consumption of the analysis. However, it was also found that an analysis of small sources suffers from an increase in space usage as BuDDy then requires more space for initializing its internal data structures than is gained by using BDDs to represent the points-to sets. Furthermore, the usage of BDDs instead of C++ STL sets led to a runtime slowdown for all benchmarks tested with the primary reason for this found in the handling of operations to be performed on the complex data structure a BDD represents.

Future work for even further improving the space requirements of the analysis could include the representation of a multitude of data structures other than points-to sets through BDDs. Generally, any kind of data structure storing information in a set wise manner would be eligible, the VFG underlying the analysis [1] being only one example.

As the severe runtime slowdown found in the analysis is either introduced by the usage of BDDs and directly linked to the structure and logic found in the algorithms of analysis [1] or is BDD independent but a direct result from the analysis inherent methodology, the current implementation is not assumed to have much potential for further time optimizations without fundamentally restructuring the analysis and its algorithms and methodologies. One possibility for such a restructuring results from the following observation: many compiler optimizations like a loop-invariant-code-motion [3] don't actually require points-to information to have been computed for all of the code under optimization but just for very specific sub-sections like loop bodies in the case of the loop-invariant-code-motion optimization. The

analysis [1] however is a whole-program analysis computing all of the information at once. In order to make the analysis [1] also attractive from a time consumption point of view, and maybe even for LLVM to consider it as an official alias analysis pass, the analysis could be turned into a *demand-driven* version where first all source code instructions possibly being relevant and possibly contributing to the final alias analysis query answer would be extracted and then the analysis as described in [1] would be performed on this extraction. As the authors of [1] reduce their analysis to a reachability analysis in a sparse CFG, the modification describe above and proposed for future work would reduce the analysis to a reachability analysis in a sparse CFG of a sparse code. This sounds promising and might be well worth investigating.

# Bibliography


[1] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 343–353, ACM, 2011.

[2] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, IEEE, 2004.

[3] S. S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[4] L. Li, C. Cifuentes, and N. Keynes, "Practical and effective symbolic analysis for buffer overflow detection," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 317–326, ACM, 2010.

[5] M. Naik, A. Aiken, and J. Whaley, *Effective static race detection for Java*, vol. 41. ACM, 2006.

[6] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[7] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 32–41, ACM, 1996.

[8] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 1, pp. 1–50, 1998.

[9] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *ACM SIGPLAN Notices*, vol. 44, pp. 226–238, ACM, 2009.

[10] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *ACM SIGPLAN Notices*, vol. 42, pp. 290–299, ACM, 2007.

[11] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, pp. 1–19, ACM, 1970.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[13] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[14] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 4, pp. 848–894, 1999.

[15] C.-Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959.

[16] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[17] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[18] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. IT University of Copenhagen. http://www.itu.dk/people/jln/. Last accessed: September 29, 2014.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

**Authored by** (in block letters):
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| | |
| | |
| | |
| | |

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*