SPADE: SEMANTICALLY PRESERVING ABSTRACT DECOMPILER EXPERIMENT

SPADE:

SEMANTICALLY PRESERVING ABSTRACT DECOMPILER EXPERIMENT

By

ANDREW JOSEPH DAVIDSON

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelor's degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

MAY 2008

Approved by:

_____

Saumya Debray
Department of Computer Science

STATEMENT BY AUTHOR

I hereby grant to the University of Arizona Library the nonexclusive worldwide right to reproduce and distribute my thesis and abstract (herein, the "licensed materials"), in whole or in part, in any and all media of distribution and in any format in existence now or developed in the future. I represent and warrant to the University of Arizona that the licensed materials are my original work, that I am the sole owner of all rights in and to the licensed materials, and that none of the licensed materials infringe or violate the rights of others. I further represent that I have obtained all necessary rights to permit the University of Arizona Library to reproduce and distribute and nonpublic third party software necessary to access, display, run, or print my thesis. I acknowledge that the University of Arizona Library may elect not to distribute my thesis in digital format if, in its reasonable judgment, it believes all such rights have not been secured.

SIGNED: _____

ABSTRACT

This thesis presents SPADE, an experimental decompiler for the x86-32 instruction set. The thesis describes some of the challenges with decompilation in general, and outlines techniques to impose high level control flow structures and data types on machine code. To the knowledge of the author, SPADE is the first decompilation system that does not presuppose that the machine code being analyzed was translated from source code. As a result, particular attention is given to relying only on program semantics, rather than potentially misleading sources of program information such as the symbol table, which may be stripped from the executable or filled with misleading information.

ACKNOWLEDGMENTS

I owe a great debt to many members of the Computer Science Department for providing me with the skills and encouragement to finish this thesis, but there are a select few to whom I am particularly grateful:

I would like to thank the members of the SOLAR research team, who allowed me to play out various ideas and techniques, provided endless encouragement, and helped to teach me many of the tools required for this work.

I would also like to thank Dr. Greg Andrews for his guidance and advice with respect to coursework and paper writing in general.

I would especially like to thank my main adviser, Dr. Saumya Debray, whose patience and insight were of great benefit both to this thesis and to me personally.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1

Introduction

A decompiler is a program that reads a program written in a machine language and translates it into an equivalent program in a high-level language [1]. This thesis will understand the concept of "equivalent" to mean that the semantics of the input program match the semantics of the output program. Since a completely isomorphic decompilation is reducible to the Halting problem, decompilers necessarily will not be able to produce precise decompilations in all cases [2].

1.1 Program Analysis

Many of the algorithms employed for compiler optimizations can be used in decompilers. This is not suprising, since the algorithms underlying optimizations are used to approximate interesting program properties. Since many of these algorithms aid SPADE, some compiler concepts are worth noting:

1.1.1 Control Flow Graphs

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [3]. Consider the following set of instructions:

```
0x40107c incl %edx
0x40107d cmpl %edx,$0x9
0x401080 jle 0x401070
0x401086 addl 0xfffffff0(%ebp) <- $0x3
0x40108b movl %exc <-$0x4
```

```
0x40107c incl %edx
0x40107d cmpl %edx,$0x9
0x401080 jle 0x401070
```

```
0x401086 addl 0xfffffff0(%ebp) <- $0x3
0x40108b movl %exc <-$0x4
```

Figure 1. Instructions grouped as basic blocks

A control flow graph is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths [4]. Here is a very simple control

flow graph:

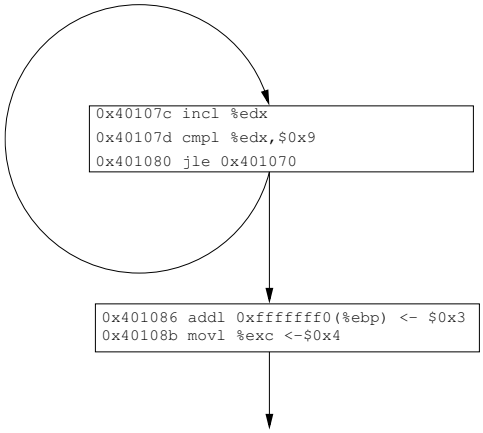

Figure 2. A simple control flow graph

1.2 Motivation

Motivation for using a decompiler comes in two steps. First, why it is important to understand what machine code is doing, and second, why it is important to understand why reading machine code directly is difficult.

1.3 The importance of understanding machine code

With the proliferation of malicious binaries, there is a very real need to understand the actions of a program before they happen. To claim that machine code is the product of a compiler is erroneous; not all machine code is written by compilers. As such, there are circumstances where there may be no high-level source code to read. Even in those situations where a high-level representation of a program already exists, it may not be available to the user of the machine code. This is often the case with malicious code for which the intent of the program is intentionally hidden.

1.4 The difficulty in reading machine code directly

The primary objectives of high-level languages and machine languages are fundamentally different. Consider first the goals of a high-level language:

- Abstract machine-specific details away from the program writer.
- Provide semantic constructs that make reading programs easy.
- Provide safety features to identify programmer errors.

Machine languages exist to provide a representation of a program that a particular architecture can actually execute. Control flow structures such as `if` statements and loops are absent, replaced instead with machine instructions that modify the program counter or repeat for a number of iterations. Safety features that exist in high-level languages such as variable types and data structures are gone completely, while the machine registers and memory are exposed.

Although not all low level code is written by compilers, a great deal of code is. Compilers are expected to perform "optimized" machine code, meaning that the compiler will automatically perform some transformations on the code to improve some aspect of the output. In some cases of optimization that improve the runtime performance of code, machine code is modified such that the flow of control in the program takes advantages of machine specific details such as number of registers, branch delay slots, and caching in ways that may obfuscate the program logic.

1.5 Related Work

A number of reverse engineering tools have been written. [5] implements a disassembler and the binary analysis framework upon which SPADE is built. The freely available decompiler REC by Giampiero Caprino provides good decompilation when a program does not make any attempt to avoid disassembly, i.e. when

the symbol table remains intact. However, REC does no type analysis when the symbol table is stripped. Similarly, [1] presents the dcc decompiler, but the scope of programs that it can disassemble is limited to small programs. The boomerang project, that carries on the work in [1] also does very little type analysis, and tests in this thesis show that it is incapable of providing any decompilation at all when fed a meaningful program. [6] presents a technique to recover some type information, but it is limited to analyzing registers only, and there is no known software implementation of the technique.

List of References

[1] C. Cifuentes, "Reverse compilation techniques," Ph.D. dissertation, Queensland University of Technology, 1978.

[2] P. T. Breuer and J. P. Bowen, "Decompilation: the enumeration of types and grammars," *ACM TOPLAS*, pp. 1613–1647, 1994.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[4] F. E. Allen, "Control flow analysis," in *ACM SIGPLAN Notices*. ACM, 1970.

[5] B. Schwarz, S. K. Debray, G. R. Andrews, and M. Legendre, "Plto: A link-time optimizer for the Intel IA-32 architecture," *Proc. 2001 Workshop on Binary Translation (WTB-2001)*, 2001.

[6] A. Mycroft, "Type-based decompilation," *European Symposium On Programming, Lecture Notes in Computer Science*, 1999.

CHAPTER 2

Overview

A decompiler can be grouped logically into three pipelined modules; a disassembler, an abstractor, and a code generator.

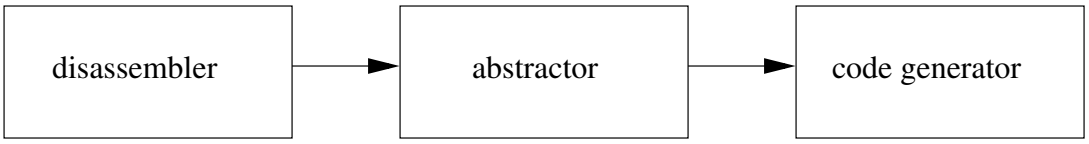

Figure 3. Three phases of decompilation

An overview of each of these modules follows, along with a brief discussion of some of the challenges faced in each module.

2.1 Disassembler

A disassembler does the job of translating machine code into assembly code. Conceptually, one can consider disassembly to be an isomorphic translation from a first generation language to a second generation language.

SPADE is built on top of an existing binary analysis system called PLTO [1], which already contains a disassembly. However, it should be noted that the von Neumann architecture does not draw a distinction between data and executable code, thus making disassembly an imprecise science. Although this paper makes no contribution to the problem of disassembly, the experimental decompiler does attempt to clean up the disassembly when possible.

2.2 Abstractor

This thesis will refer to the process of translating second generation code into a sequence of machine independent constructs as abstracting, and the module responsible for this translation as an abstractor. In practice, this process entails

taking the input from the disassembler and providing an syntax tree tree to the code generator.

It is in the abstractor that constructs such as named variables and algebraic expressions are imposed upon a program. The main contribution of this work is in the abstractor, which attempts to move the input binary to as abstract a representation as possible while still retaining semantic equivalence with the machine code.

2.3 Code Generator

A code generator produces third generation code from a syntax tree and whatever auxilary information is produced in the abstractor. SPADE uses a custom built code generator to produce output that is similar to C. In cases where no appropriate C statement matches can be enlisted to represent some aspect of a given syntax tree, the code generator will instead capture that aspect using pseudocode or the corresponding assembly code.

List of References

[1] B. Schwarz, S. K. Debray, G. R. Andrews, and M. Legendre, "Plto: A link-time optimizer for the Intel IA-32 architecture," *Proc. 2001 Workshop on Binary Translation (WTB-2001)*, 2001.

CHAPTER 3

Control Flow Analysis

3.1 Criteria for Evaluation

To the knowledge of the author of this thesis, no reverse engineering tool has attempted to quantify the effectiveness of imposing high-level control flow structures onto a function. It has, however, been suggested that an aim of a programmer should be to make programs in which the code length is proportional to ease of understanding[1]. Therefore, this thesis proposes that the goal of control flow analysis in a decompilation context is to minimize the number of non-commenting source statements in the program while retaining semantic equivalence.

3.2 Jump conditions

Conditional branches in the x86 architecture is usually specified via a pair of instructions: a conditional jump instruction, i.e. `je`, that depends on some flags of the program status word (PSW). This jump is preceded by some instruction that sets those flags of the PSW. To identify conditional statements, an algorithm is in place that will begin at every jump instruction and traverse the control flow graph backwards until a instruction is found that defines the correct PSW flags.

3.3 Loops

SPADE identifies loops using a pre-existing algorithm from the PLTO system. Essentially, this algorithm finds natural loops by traversing the PLTO generated control flow graph and identifying any back edge as a natural loop. The exit condition of the loop is already known from the previous algorithm. If the conditional is checked at the end of the natural loop, the loop will be structured as a `do/while`. If the condition is found at the beginning, the loop is structured as a `while`

3.4 Acyclic Control Flow

SPADE uses an algorithm to identify conditional branch statements inspired by the natural loop algorithm presented in the previous section. For simplicity , discussion of this algorithm will only consider boolean conditions, though the algorithm can handle n-way conditions.

Intuitively, a branch statement consists of two mutually exclusive branches. A branch begins with a condition that specifies which of these branches to enter, and ends with a vertex that all of the subgraphs reach.

Consider each block h with multiple successors. Without loss of generality, successor b may begin a branch if b will only be executed iff edge (h, b) is taken. All blocks that satisfy this notion of being part of a branch can be identified using the following algorithm:

```
Let h be a basic block in a control flow graph.
Let b be a basic block in a control flow graph.
Let l be a list of basic blocks, initially empty.
Let v be a list of basic blocks, initially empty.
procedure add_to_branch(h, b, l, v)
if b not in v and h preDom b and not b postDom h:
  l add b
  for each successor s of b:
    add_to_branch(h, s, l)
v add b
```

add_to_branch(h, b, {}, {}) should be called in turn for each successor b of h. This should be done for every h

```
Let r be the entry to a function.
procedure find_headers(r)
stack := r
headers := empty
visited := empty
while stack is not empty:
  pop h off of stack
  if h is a branch statement and h is not a loop header:
    headers add h
  visited add h
  push each successor of h not in visited onto stack
```

A small example of the above algorithm might operate over the following snippet of code:

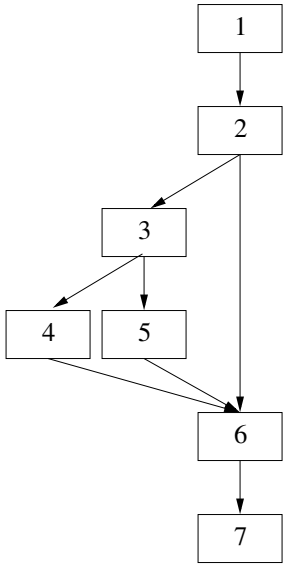

Figure 4. Control flow graph with acyclic control

The corresponding dominator trees are as follows:

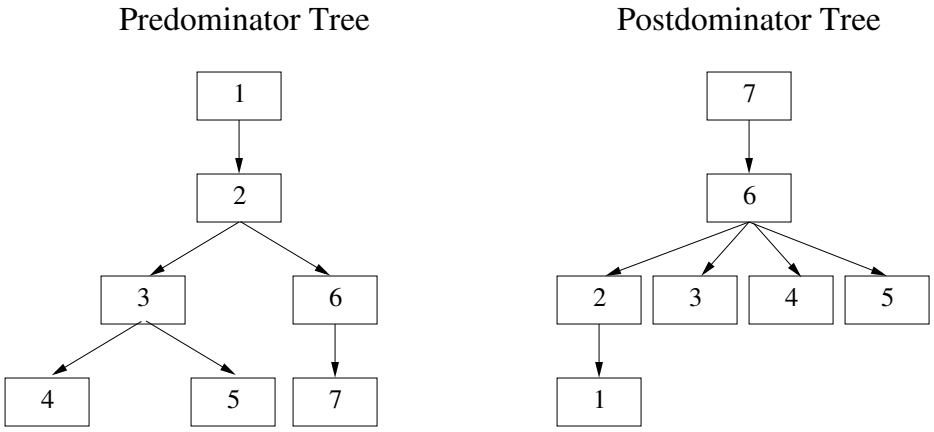

Figure 5. Dominator Trees

After calling *find_headers(1)*, the *headers* list contains {2,3}. These are the only branch instructions in the graph. Next, *add_to_branch* is called for each edge incident from an element of *headers* a branch. These calls are detailed

below:

add_to_branch(2,3,{},{})

h	e	l	v	addition?
2	3	{}	{}	yes, 3
2	4	{3}	{3}	yes, 4
2	6	{3,4}	{3,4}	no, 6 postDom 2
2	5	{3,4}	{3,4,6}	yes, 5
2	6	{3,4,5}	{3,4,6,5}	no, 6 is in v
2	6	{3,4,5}	{3,4,6,5}	no, 6 is in v

add_to_branch(2,6,{},{})

h	e	l	v	addition?
2	6	{}	{}	no, 6 postDom 2

add_to_branch(3,4,{},{})

h	e	l	v	addition?
3	4	{}	{}	yes, 4
3	6	{4}	{4}	no, 6 postDom 3

add_to_branch(3,5,{},{})

h	e	l	v	addition?
3	5	{}	{}	yes, 5
3	6	{5}	{5}	no, 6 postDom 3

```
1
2:
  3:
    4
  not 3:
    5
6
7
```

In practice, it is also convenient to give each basic block a pointer to the branch in which it it belongs. If headers are visited in a depth first order, then each pointer with identify the most deeply nested branch. Note also that edges that do not participate in loops or branches will be structured as `goto` statements.

List of References

[1] E. W. Dijkistra, *Notes on Structured Programming.* New York: Academic Press, 1972, pp. 1–82.

CHAPTER 4

Data Type Analysis

In the context of decompilation, data type analysis can be considered to be a two-step process. The first step is to partition the uses/definitions of memory locations and registers into variables; high-level language constructs that have a specific data type, where type is either a primitive data type such as int, float, char, or an abstract data type such as an array or a struct. The second step is to select specific data types for those variables that are as abstract as possible without violating the semantics of the machine language.

4.1 Criteria for Evaluation

As is the case with control flow analysis, there is no known precedent for evaluating a decompiler's data type analysis. Maximizing the number of primitive variables would wrongly reward fragmented decompilations, such as one that assigns a variable to every use or definition of a register or memory location. On the other hand, minimizing the number of primitive variables would reward decompilations that make no effort to split variables into live ranges. However, since the objective of a decompiler is to develop a more structured representation of a program, it follows logically that a highly structured set of variables is more desirable than an unstructured set of variables. This thesis proposes the use of levels of nesting within the fields of abstract data types as a quantifiable measurement of progress for a data type analysis. This criterion rewards analyses that detect live ranges for variables, since a precise type can be assigned to those types, which is a necessary prerequisite in determining if one variable is in fact a field of a structure or array. At the same time, the criterion does not reward a decompiler that splurges on variables, since none of these variables can properly be represented as

abstract data types.

4.2 Challenges of Data Type Analysis

Determining the set of variables for a function, and the type of those variables, is an extremely valuable part of any decompilation. However, several factors in the x86 instruction set complicate type analysis

4.2.1 Aliasing

x86 is a load/store architecture. This gives rise to the possibility that a value may be referenced in more than one register. Since SPADE does not implement an alias analysis, it is very difficult to infer behavior of a type based on the uses of that variable, because that same variable may be used in a different way through a different alias.

Consider the following example:

```
1.  mov %eax, 0x401050
2.  mov %ebx, %ecx
3.  mov (%ebx), 5
4.  mov %edx, (%eax)
```

`%eax` is loaded with a constant value on line 1. That value is dereferenced on line 4, meaning that `%eax` is a pointer to some value. However, `%ebx` is loaded with an unknown value from `%ecx` on line 2.If `%ecx` contains `0x401050`, then `%ebx`, `%ecx`, and `%eax` all alias the same value at the end of statement 2. When the value pointed to be `%ebx` is modified on line 3, the effect of line 4 may be effected if `%ebx` aliases `%eax`

In situations where a truly unknown value is dereferenced, as in the code snippet above, the special value `UNKNOWN` will be assigned from the unknown value.

SPADE will be fall back on an extremely conservative analysis where all possible values that the pointer could alias will lose their type information. If an advanced alias analysis algorithm were dropped into the system, the `UNKNOWN` value could be replaced with a set of possible locations pointed to. In that case, less type information would be lost.

4.2.2 Bookkeeping Code

Assembly/machine code introduce a bookkeeping code to reserve space for arguments to functions and local variables. Since much of this bookkeeping is not related to an abstraction of the function, this stack code should not appear in a high-level representation. SPADE recognizes the depth of a stack by measuring the contribution each instruction makes to the stack using the existing PLTO use-depth and kill-depth analyses. If a function is well-behaved, it has a known number of stack slots that can be evaluated as memory locations just as though they were on the heap. However, the lack of an alias analysis in SPADE prevents bookkeeping information from being removed from the high-level output.

4.3 Primitive Type Analysis

The SPADE system uses an iterative analysis to abstract a set of local variables on a per function basis and a set of global variables that can be accessed by each function. The initial set of variables consists of 1 variable for each register used in a function and 1 variable for each distinct memory location in that function. This is an overly conservative estimate of types that does not take into account the concept of live ranges, namely that a register may be involved in some computation and may then have its value redefined for use in a completely different computation. While keeping a one to one correspondence between a memory location/register and a variable is a tolerable approximation for small examples, it does not prove

sufficient for larger cases.

4.3.1 Integer Types

Each integer register and memory location begins with the `void *` type. If it can be determined that an integer register or the contents of a memory location are never dereferenced, then the type of that variable is narrowed to `int`.

4.3.2 Floating Point Types

If a memory location is loaded into a floating point register, the contents of that memory location are typed `float`. Similarly, if a floating point register is stored at a memory location, that memory location is given the type float.

Use-depth analysis to determine stack slots. Currently a register is considered a distinct location, but live range is our goal. Abstract data type analysis. Amalgamate primitive types based on control flow.

4.4 Abstract Type Analysis

This paper introduces a novel technique for building abstract type information from primitive types. The general idea is that abstract types gain their meaning from the context in which they are accessed. For example, arrays are data structures whose elements are accessed iteratively. Structs are data structures whose elements are accessed off of a common base address. is usually accessed by running over each element sequentially in a loop. If a user were to program an array with exactly 2 indices, each of which was accessed completely statically and without respect to the other index, it would be more accurate to abstract the executable to represent two distinct values, since they are not being used in any way together.

SPADE will mark a variable as an array when there is a memory access that is iteratively updated off of a base address. It is theoretically possible to recursively perform type analysis on the elements of an array, but in practice this technique

has never been able to recover a structure more deeply nested than an array of a primitive type.

Conceptually, structs can be decompiled using a similar heuristic to arrays; if a base address is computed and then a second address is used to index off of that base address, it is possible to represent the base address as the start of a struct and the indices as elements thereof. Structs are rarely decompiled, because direct access to the fields of a struct (instead of using a base and index) appear to be uses and definitions of unrelated types. This is actually not considered by the author to be a weakness of the analysis. Rather, it fits with the design goal of this thesis to abstract machine code, rather than reverse engineer it.

4.5 Global Memory Analysis

In a CISC architecture like IA32 that supports x86-32, values that have scope outside of a function will rarely, if ever, be kept in register. Therefore, absolute memory addresses that are dereferenced and used as data are heuristically defined to be global memory locations. In situations where a global variable does live in a register, it should be considered to be passed in as an argument to any function that uses that variable.

4.6 Argument Analysis

If use/def information is present for a function, there will be a set of variables for which the underlying registers and memory locations are used in a function before being defined. Assuming that global memory is correctly identified and ignored, the remaining values will have been passed in as arguments to the function.

CHAPTER 5

Results

5.1 Control Flow Improvement

As mentioned in Chapter 3, minimizing non-commenting source statements can be used as a quantitative measurement for how much progress has been made from low-level code to high-level code. For SPADE, this was done by taking the number of statements from the disassemnbly without any abstraction or improvement whatsoever and comparing it to the number of statements with as much abstraction and improvement as possible. These numbers are shown here:

Decompilation Comparison

Program	Before Improvement	After Improvement	Percent Reduction
AcroRd32.exe	3111	2889	7.1
bzip2.exe	18472	18347	0.7
gimp-2.2.exe	25613	15032	41.3
iPodService.exe	46625	40996	12.1
moviemk.exe	628388	19634	96.9
PictureViewer.exe	27619	24474	11.4
RAWImage.exe	60927	24841	59.2
SDViewerDSC.exe	13754	9825	28.6
SlideShow	79065	54458	31.1
winamp	63123	54783	13.2
YahooMessenger.exe	694377	50305	92.8
ZoomBrowser.exe	4058	3370	17.0

It is also interesting to consider SPADE's result against that of other decompilers. Three decompilers were tested over the programs listed above: SPADE, REC, and boomerang. Of the test cases listed in the previous table, boomerang fails to provide output an all but three test programs, so it has not been shown for comparison.

One problem faced by a researcher attempting to compare decompilations is that each decompiler relies on its own disassembly techniques. Attempting to count the number of non-commenting source statements relies heavily on the quality of these techniques, which varies widely. A figure that is more robust to varying disassemblies is to determine the number of control flow structures (NCFS) that are recognized by each decompiler over the total number of source statements that it disassembles (NCSS). The following table summarizes the results of this comparison, over all those test cases for which both REC and SPADE provide reasonable disassemblies:

Decompilation Comparison

Program	SPADE NCSS	SPADE NCFS	REC NCSS	REC NCFS	SPADE NCFS/NCSS	REC NCFS/NCSS
AcroRd32.exe	4315	269	3126	252	0.062	0.081
bzip2.exe	25060	2804	96918	1999	0.112	0.021
gimp-2.2.exe	18195	1513	1271	50	0.083	0.039
iPodService.exe	42393	328	362610	6449	0.008	0.018
PictureViewer.exe	25235	195	92946	1394	0.008	0.015
RAWImage.exe	30910	2863	4275	186	0.093	0.044
SDViewerDSC.exe	32754	885	3399	256	0.064	0.075
YahooMessenger.exe	68568	3780	364446	5182	0.053	0.041
ZoomBrowser.exe	4763	368	65016	1227	0.077	0.019

5.2 Data Improvement

To the knowledge of the author, no other decompiler exists that is capable of inferring type information reliably. Although SPADE proposes a technique for imposing primitive and complex types on a binary, memory references cannot be perfectly analyzed. Furthermore, neither SPADE nor REC has an alias analysis system capable of reducing the possible locations to a tractable set of values. However, in limited cases, as described in the previous sections on type analysis, some heuristic methods can be applied to show a memory location as an abstract data structure.

SPADE cannot guarantee that some function will not be reached into by another function and have data modified in a way that violates its type constraints. As such, it is not safe to impose a particular abstract data type on a given memory location in the test cases presented. That being said, there are some assumptions that SPADE makes to reasonably structure types. If each function is considered independently (i.e. every function is considered to have disjoint uses and definitions of registers and stack), then at least one abstract data structure can be found in each of the test programs of depth 1.

CHAPTER 6

Conclusion

Although the SPADE system shows theoretical potential beyond its current implementation, more analyses need to be implemented in order to correct for large generalizations that the system currently makes. As it currently stands, SPADE is comparable to the best known freeware decompiler and superior to the best known open-source decompiler. The remainder of this section will explore the major areas for improvement on SPADE:

6.1 Liveness Analysis

The largest problem for the type analysis in SPADE stems from the conglomeration of all uses and definitions of a register into a single variable. Because registers are used so frequently in so many different capacities, SPADE is overly conservative with the types that can be assigned to variables.

6.2 Function Argument Analysis

SPADE's inability to recognize function arguments compounds the problem with variable recognition. Since there is no guarantee that variables are killed across function bodies, the type of a register or memory location is polluted by being used in different ways in different functions.

6.3 Expression Amalgamation

As minimizing non-commenting source statements is the criterion for the control flow analysis, a useful analysis to add to SPADE would be to determine when one expression is a subexpression of another and combine them into a single high-level expression. This would eliminate the appearance of intermediate computations as independent statements and therefore increase both readability and non-

commenting source statements score.

BIBLIOGRAPHY

Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

Allen, F. E., "Control flow analysis," in *ACM SIGPLAN Notices*. ACM, 1970.

Breuer, P. T. and Bowen, J. P., "Decompilation: the enumeration of types and grammars," *ACM TOPLAS*, pp. 1613–1647, 1994.

Cifuentes, C., "Reverse compilation techniques," Ph.D. dissertation, Queensland University of Technology, 1978.

Dijkistra, E. W., *Notes on Structured Programming*. New York: Academic Press, 1972, pp. 1–82.

Mycroft, A., "Type-based decompilation," *European Symposium On Programming, Lecture Notes in Computer Science*, 1999.

Schwarz, B., Debray, S. K., Andrews, G. R., and Legendre, M., "Plto: A link-time optimizer for the Intel IA-32 architecture," *Proc. 2001 Workshop on Binary Translation (WTB-2001)*, 2001.