Purdue University
Purdue e-Pubs

Open Access Theses | Theses and Dissertations

8-2018

Improving Precision for x86 Binary Analysis Techniques

Lovepreet Singh
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_theses

Recommended Citation
Singh, Lovepreet, "Improving Precision for x86 Binary Analysis Techniques" (2018). *Open Access Theses*. 1596.
https://docs.lib.purdue.edu/open_access_theses/1596

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

IMPROVING PRECISION FOR x86 BINARY ANALYSIS TECHNIQUES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Lovepreet Singh

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Mathias J. Payer, Chair
Department of Computer Science
Dr. Byoungyoung Lee
Department of Computer Science
Dr. Dongyan Xu
Department of Computer Science

Approved by:

Dr. Voicu Popescu by Dr. William J Gorman
Head of the Department Graduate Program

ACKNOWLEDGMENTS

First, I would like to thank Dr. Mathias Payer for his valuable support and guidance for my research. His feedback and suggestions helped me steer forward my research and compile it into this thesis.

I would like to thank Sushant Dinesh for informative discussions on my research and answering countless questions related to binary analysis. I would also like to thank him for his feedback on making major portions of this thesis coherent.

I would also like to thank Prashast Srivastava for helpful comments on initial sections of this thesis.

My sincere thanks to other members of my thesis committee: Dr. Byoungyoung Lee and Dr. Dongyan Xu, for providing helpful feedback on my research.

Last, I would like to thank Dr. William Gorman for assisting me in understanding and fixing various formatting issues on a short notice.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . vi

LIST OF FIGURES . . . vii

ABSTRACT . . . viii

1 INTRODUCTION . . . 1
- 1.1 Problem Statements . . . 2
  - 1.1.1 Binary Optimization Detection . . . 2
  - 1.1.2 Memory Layout Detection . . . 3

2 BACKGROUND AND RELATED WORK . . . 4
- 2.1 Binary Analysis . . . 4
- 2.2 Memory Layout . . . 5
- 2.3 Binary Optimization Detection . . . 6
- 2.4 Memory Layout Detection . . . 7

3 BINARY OPTIMIZATION DETECTION . . . 10
- 3.1 Design . . . 10
- 3.2 Implementation . . . 12
  - 3.2.1 Features Collected . . . 12
  - 3.2.2 Feature Extraction . . . 15
  - 3.2.3 Classification . . . 15
- 3.3 Evaluation . . . 16
- 3.4 Discussion . . . 17
- 3.5 Summary . . . 18

4 MEMORY LAYOUT DETECTION . . . 21
- 4.1 Design . . . 21
  - 4.1.1 Assumptions . . . 23
- 4.2 Implementation . . . 24
  - 4.2.1 MemTree Structure . . . 25
  - 4.2.2 Basic MemTree Construction . . . 27
  - 4.2.3 Address Pattern Analysis . . . 29
  - 4.2.4 Restructuring using Identified Patterns . . . 32
  - 4.2.5 Inter-function Propagation . . . 34

Page

4.3 Evaluation 38
4.3.1 Obtaining Ground Truth 39
4.3.2 Evaluation Parameters 42
4.4 Discussion 49
4.5 Summary 51
5 SUMMARY 55
REFERENCES 56
APPENDIX 59

LIST OF TABLES

Table — Page

3.1 Confusion matrix for compiler and optimization detection – Iteration 1. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification. 18

3.2 Confusion matrix for compiler and optimization detection – Iteration 2. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification. 19

3.3 Confusion matrix for compiler and optimization detection – Iteration 3. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification. 19

3.4 Confusion matrix for compiler and optimization detection – Iteration 4. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification. 20

4.1 Evaluation parameter abbreviations 44

4.2 MemTree results for coreutils-8.29. See Table 4.1 for abbreviations 44

4.3 MemTree results for binutils-2.29. See Table 4.1 for abbreviations 48

4.4 MemTree results for SPECCPU2006 C binaries. See Table 4.1 for abbreviations 48

4.5 MemTree evaluation summary 49

A.1 Pattern name abbreviations 59

A.2 Rules for combining address patterns across add, subtract and multiply operations. See Table A.1 for abbreviations 59

A.3 Summary of rules for combining address patterns across operations 66

LIST OF FIGURES

Figure Page

2.1 Example of MIR and internal representation for x86-64 assembly instructions 9

3.1 Design for compiler and optimization detection 11

3.2 x86-64 instructions to load a value 12

4.1 MemTree analysis design 23

4.2 Different nodes (with thick border) representing same address in SSA form 26

4.3 A Phi node (with thick border) could represent multiple addresses 27

4.4 MemNode representing a hierarchical object 27

4.5 MemNode representing an array 27

4.6 Basic MemTree 29

4.7 Result of abstract interpretation 52

4.8 Example of recursive Phi node (with thick border) representing a loop . . 53

4.9 Example of Phi node (with thick border) which gives same patterns as a recursive Phi node, but does not represent a loop 53

4.10 Array elements accessed individually 54

ABSTRACT

Singh, Lovepreet MS, Purdue University, August 2018. Improving Precision for x86 Binary Analysis Techniques. Major Professor: Mathias J. Payer.

Static binary analysis is being used extensively for detecting security flaws in binary programs. Multiple solutions have been proposed to tackle challenges presented by static binary analysis. We propose two methods to improve these solutions for better precision on x86-64 binaries. First, we propose a machine learning based approach to detect compiler and optimization level for a binary program with the aim of augmenting existing heuristic based solutions to fine tune those heuristics. We are able to detect the aforementioned information with 83% precision on coreutils, binutils and SPECCPU2006 binaries. Second, we propose an analysis to detect memory layout from a binary program's perspective. This analysis aims to enhance existing solutions by allowing them to track values across loads and stores in fine grained memory locations. We are able to detect layout of stack objects with 56.3% accuracy for coreutils, binutils and SPECCPU2006 C binaries.

Chapter 1. INTRODUCTION

Binary analysis is a fundamental technique for finding vulnerabilities in a program as no source code is required. Although, a binary has less information about the semantics of a program, the benefits binary analyses offer surpass those of the source code analysis techniques. Since binary code is what actually gets executed on a machine, it offers much more accurate representation of execution semantics, as compared to source code. A compiler may not necessarily compile code to do what a programmer intends to do in source code [1]. With modern compilers pushing towards more aggressive techniques for optimizations and handling *undefined behaviors* in different ways, the difference between what is intended in source code and what is executed on the machine could be significant.

However, analyzing binary code presents a set of challenges. Source code allows programmers to explicitly specify semantic meaning of constructs in a program. For example, a programmer could explicitly use a C struct data type to represent related set of information. Similarly, a programmer could use loops to execute similar code over multiple datasets. Programmers could also define functions, which represents a related block of code. During compilation, these semantic constructs are lost, since they present no meaningful information to the processor executing the binary code. Further, the binary code could be stripped of symbol information to prevent them from being reverse engineered, as is common in case of Commercial Off-The-Shelf (COTS) binaries. Binary code consists of a sequence of instructions, including logical and control flow instructions. Performing any meaningful modification to the program requires use of semantic meaning of constructs in the program, and retrieving this information from a binary is a challenging task. There have been multiple attempts to retrieve semantic constructs from binaries, like functions [2, 3, 3–5], data types [6], or control flow [7].

1.1 Problem Statements

This thesis focuses on retrieving compiler, optimization level, and memory layout from point of view of a binary program, to improve precision of existing binary analyses techniques as well as enable new binary analyses. We explicitly focus on binaries compiled for x86-64 architecture for both problems. The following subsections introduce the individual problems and benefits they offer to existing binary analysis techniques.

1.1.1 Binary Optimization Detection

Compilers such as GCC and LLVM offer multiple flags to affect the compilation process from source code to binary format. These flags are commonly grouped into four different optimization levels – O0, O1, O2 and O3 – starting from lowest to highest optimization. The particular modifications performed for each flag depends upon the compiler family. Each flag causes a deterministic change in the output binary, which could help in detecting the optimization level used to compile a binary executable.

Detecting the optimization levels helps in tuning heuristics for binary analysis. For example, GCC offers the flag `-finline-functions` [8], which performs inlining of functions even when they are not declared `inline` in source code. On recognizing that a binary has been compiled using GCC with O3 optimization, which enables `-finline-functions`, an existing binary analysis technique to detect functions in the source code could learn that functions not marked `inline` could have been inlined during the compilation. Thereafter, the technique could use existing heuristics, such as code similarity measures on basic blocks, to find potential functions in source code which have been inlined by the compiler. As another example, Xandra [9] applied peephole optimization and register allocation on DARPA's Cyber Grand Challenge (CGC) binaries for improving performance of defense protections, because the team noticed that few of the challenge binaries were not optimized.

Therefore, detecting the compiler family and optimization level used to compile a binary would lead to better results for binary analysis techniques which use heuristics. With our system, we are able to detect compiler and optimization level with 83% precision.

1.1.2 Memory Layout Detection

In high level languages like C, programmers could assign type semantics, such as primitive data type, composite data type, or union data type, to memory objects. Primitive data types include types such as `char`, `int`, and `float`. Composite data types include arrays and C structs. Union data type allow the same memory location to be used for more than one data types. Whereas, a binary program views memory as just a sequence of bytes without any explicit semantic meaning related to them. We aim to infer a higher level abstract layout of memory from a binary program's perspective, based on how different memory objects are used in a binary program.

Memory layout detection is an important problem in binary analysis to enable a family of modifications which include modifying memory layout, to a binary program. For example, if a value is spilled into a stack slot in an unoptimized binary, and later it is detected during an analysis that the value could be promoted to registers without the need of stack slot, the memory layout could be modified to remove the stack slot for that value, hence giving an optimized binary as output. Other modifications to stack layout, such as reshuffling of slots assigned to local variables, would also be possible if we infer the complete layout. As another example, if all arrays in a binary program are known with correct sizes, an analysis pass could add red zones at end of these arrays to help sanitize these issues during testing of binaries.

Currently, we focus our analysis on binaries compiled from C programming language only. We are able to detect 79.6% offsets for stack objects, and correct sizes (aggregate sizes in case of non-basic objects) for 56.3% stack objects.

Chapter 2. BACKGROUND AND RELATED WORK

2.1 Binary Analysis

Binary analysis involves analyzing binary encoded instructions and data, in order to recover information about an executable binary. ELF [10] and PE [11] are two popular binary file formats for executable binaries, used on Unix-based and Windows systems respectively. In this thesis, we focus only on ELF format binary executables. ELF format contains information about how to execute a piece of binary code – including code itself, global data used by the code, memory segments and permissions on them, or exception handling. Several tools, such as radare, IDA, or Binary Ninja, can read ELF files and present this information in a manner easier for binary analysis.

Binary analysis techniques are broadly categorized into– Static binary analysis and Dynamic binary analysis techniques. Static binary analysis involves processing a binary without actually executing the binary. Hence, due to absence of runtime information, static analysis techniques can only approximate the actual behavior a binary would exhibit upon execution. But static analysis has an advantage of being able to explore all paths of execution which do not depend upon dynamic runtime targets. Dynamic binary analysis involves executing the binary under supervised conditions, for example by instrumentation or as child process, to collect information from the binary program. Dynamic binary analysis has an advantage of being accurate due to runtime values being observed. But it has a disadvantage of not being able to explore all possible paths of execution, for example in case of complex or non-deterministic conditional control flows.

Architectures provide different basic capabilities for executing code and binary programs use these capabilities, therefore abstracting these capabilities is an important step during static binary analysis. Examples of such capabilities are – instruc-

tion set, registers, or side effects of instructions. For this purpose, binary code is first converted to an Intermediate Representation (IR) in a process known as *lifting*, which provides an architecture independent representation of a program. Similarly, to abstract away high-level language features, source code is first converted to IR, and then multiple passes are applied to the IR before converting it to binary code. VEX IR is one such IR used for representation of binary code, whereas LLVM IR is commonly used for representation of source code. Source code based IR tends to be more refined than binary code based IR, due to more semantical information being present in source code. Single Static Assignment (SSA) is a form of representation of a program, where each variable gets assigned a value exactly once. Owing to single assignment of SSA representation, it provides better opportunities for analysis over the program. Abstract interpretation [12] involves interpreting operations performed by a program over a set of abstract objects in order to approximate the results of an actual execution of the program. Abstract interpretation is commonly [13, 14] used to retrieve approximation of semantics not explicitly present in a binary program.

2.2 Memory Layout

Programs execute logical statements on a working set of data. During compilation, a compiler tries to assign as many data values as possible to registers present in a specific architecture. But if the registers do not suffice for holding all the relevant data in the working set, these data values are spilled to primary memory storage, such as RAM. From point of view of a binary, the primary memory storage could be viewed as a list of consecutive memory chunks. Binaries without debug information neither have explicit indication about the type of data stored in each chunk, nor do they have explicit indication about semantic meaning of consecutive memory chunks which could be constructs in source code such as structs, or arrays.

For a binary program, the primary memory for data storage could be divided into 3 broad areas – Global, Stack, and Heap. Each thread of execution in a binary

program may have its own Stack and Global storage area [15]. We describe these broad areas below:

- **Global.** This includes global data that is accessible throughout the binary. For an ELF file, **.rodata**, **.bss**, and **.data** sections represent global data sections. Each thread of execution may also have its own instance of global data, present in **.tdata** and **.tbss** sections. Additionally, global variables used by a binary program may be present in Dynamic Shared Objects (DSOs). These are usually accessed using Global Offset Table (GOT), which is updated during dynamic linking when a binary program is executed [16].
- **Stack.** Upon invocation, each function is assigned memory space in Stack, to persist its working set of data. This space is intended to store data local to a function, since the memory space is reclaimed when the function returns back to its caller function. Similarly, each thread is assigned its own stack during execution to persist thread local data.
- **Heap.** Memory space from heap is allocated at runtime. Therefore, it is commonly used to store data whose size will be known at runtime. Operating systems offer system calls and higher level APIs to allocate and deallocate memory from heap at runtime. Data stored in this space could be passed from one function to another, until the memory space is explicitly freed.

2.3 Binary Optimization Detection

Machine learning is a field involving use of optimization methods to create and reason about models which allow machines to make complex decisions. Machine learning is currently used in various fields to perform tasks which previously seemed difficult for machines. These tasks involve detecting and classifying objects in images, solving CAPTCHAs and even driving cars. Machine learning has also found its way in binary analysis [2–4, 17–20], for tasks such as detecting function boundaries, malware

detection and toolchain provenance for binaries. A previous work at toolchain provenance [19] is limited to coarse grained optimization levels – High and Low. Based on a similar approach, we offer a fine grained optimization level detection, using features specifically targeted for this purpose.

2.4 Memory Layout Detection

Aggregate Structure Identification [21] aims at identifying memory layout by decomposing memory, initially assumed to be a large aggregate object, into small units called atoms, using access patterns in a program. However, ASI assumes sizes of data references and data references to arrays to be known, which is not always the case in assembly code. Value Set Analysis [14] aims to find the over-approximation of possible set of values, in form of Reduced Interval Congruence (RIC), for any given *a-loc* at a point in a program. An *a-loc* could be a register or a memory location. VSA starts with a set of easily identifiable memory locations, such as globals and offsets relative to stack pointer, and find more memory objects using possible set of values for addresses loaded from existing *a-locs*. DIVINE [22] combines VSA with ASI to find structures of *a-locs*. Since VSA represents set of possible values using RIC, it is able to augment ASI using strides for arrays. Reps & Balakrishnan [23] improve upon the existing VSA by using context sensitive VSA algorithm. Our aim is similar to DIVINE, but involves address pattern analysis instead of tracking possible set of values for *a-locs*. Mycroft [24] generates type constraints over operations in Register Transfer Level (RTL) representation, and perform type reconstruction using these constraints to obtain structural information. However, it aims primarily at decompilation of binary programs instead of detecting layout of memory.

For detecting memory layouts in binary programs, we build our analysis upon Fafnir. Fafnir is a framework built to apply static analyses over SSA form of IR for binary programs. The goal of this framework is to provide basic abstractions over the IR, which could be further refined by multiple analysis passes to give an IR as close

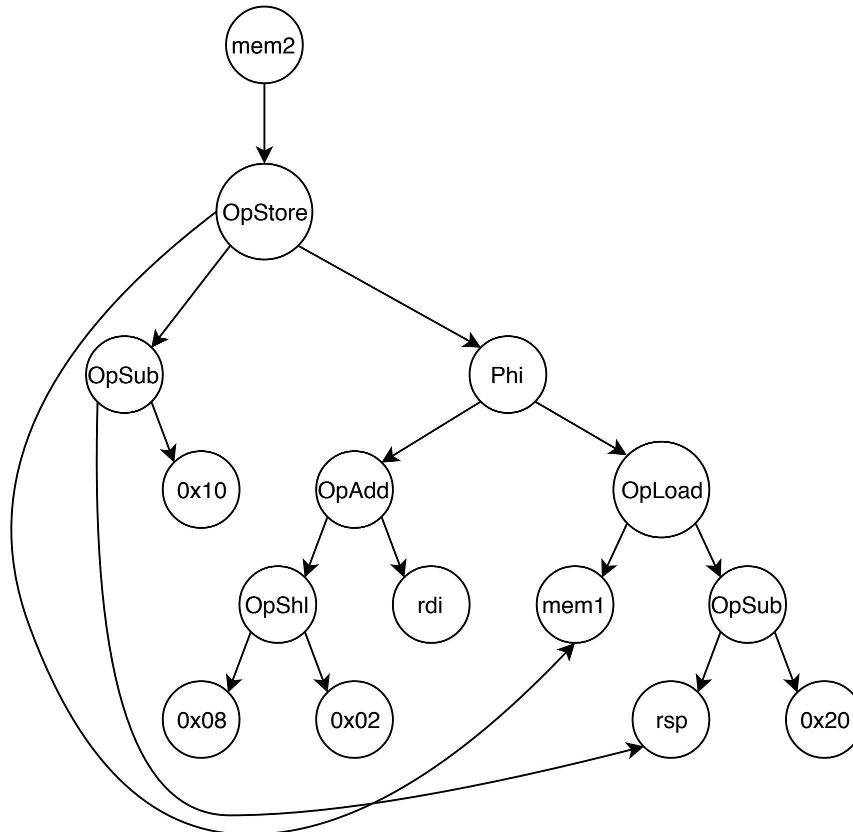
as possible to a source code based IR such as LLVM IR. Since a binary executable does not have explicit semantic information about the program, Fafnir allows quickly building a SSA form of IR for the binary executable, with the missing pieces of information termed as "uncertainties", and subsequent analysis passes reduce the amount of "uncertainties" involved in the IR. Some examples of "uncertainties" are – targets of indirect jumps and calls, distinction between code/data references and scalars, or types/structures of memory objects. Fafnir internally uses an intertwined Control Flow Graph (CFG) and Abstract Syntax Tree (AST) to represent individual functions. These functions are then related using a call-graph containing mappings for arguments passed and parameters received by caller and callee functions respectively. Figure 2.1 gives an example of a simplified AST generated by Fafnir, corresponding to a set of x86-64 assembly instructions.

```
|        ...
|        mov rax, [rsp-0x20]
|        jmp 0xfoobar
`->      mov rax, 0x08
         shl rax, 0x02
         add rax, rdi
0xfoobar: mov [rsp-0x10], rax
```

(a) Example x86-64 code snippet, arrow represents a jump from some other part of the code

```
%1:$Unknown64 = rsp - 0x20
%2:$Unknown64 = Load(mem1, %1)
...
%4:$Unknown64 = 0x08 << 0x02
%5:$Unknown64 = %4 + rdi
%6:$Unknown64 = Phi(%2, %5)
%7:$Unknown64 = rsp - 0x10
%8:$Unknown0 = Store(mem1, %7, %6)
```

(b) Representative MIR for given assembly code



(c) Internal representation for given MIR

Figure 2.1.: Example of MIR and internal representation for x86-64 assembly instructions

Chapter 3. BINARY OPTIMIZATION DETECTION

3.1 Design

This analysis aims to detect the compiler and optimization level used to convert high-level source code to a binary. In essence, our approach involves searching for features indicative of signatures for individual compiler and optimization level in a training set of binaries, and using machine learning to learn a model that classifies these features in unseen test binaries. The work flow is divided into 3 stages (Figure 3.1) described below:

- **Feature collection.** In the first stage, features are collected from the binary to create a feature vector representing the binary, in a high dimensional vector space. Binaries are obtained from source code by compiling them with multiple compiler families and different optimization levels for each compiler family. Then specific features are extracted by processing the binary, and assigned labels corresponding to the compiler and optimization level used for the binary. The collected feature vectors are split into 2 different sets – 70% for training (called training set) and 30% for testing (called testing set).

- **Training a classifier model.** In the second stage, a classifier model is trained on the feature vectors in the training set, to classify these feature vectors into correct classes. The classes here correspond to all possible combinations of compiler families and optimization levels. The model is trained using 10-fold cross validation on the training set to learn best hyper-parameters for the classifier.

- **Classifying new binaries.** In final stage, the model trained in second stage is used to classify feature vectors in testing set into classes. The model is then evaluated on the basis of how many feature vectors it correctly classifies.

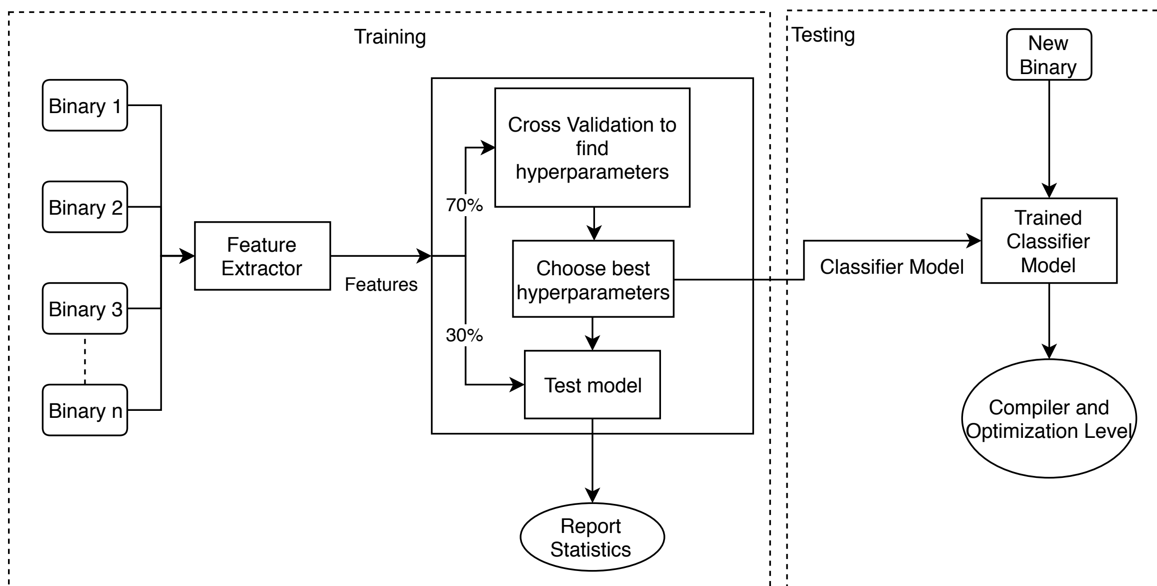


Figure 3.1.: Design for compiler and optimization detection

In this analysis, we assume that the binary contains a symbol table, to extract boundary information about functions present in the binary.

3.2 Implementation

3.2.1 Features Collected

The features extracted are specifically targeted at deterministic effects of using different optimization flags during compilation. These features are present in all non-adversarial binaries compiled with general purpose compilers and they implicitly capture the compiler family. Following are the details for specific features:

- **Instructions on which load/store depends.** Programmers store the data to be used in form of high-level abstraction called variables. Few of these variables get assigned to registers based on specific architecture the program is compiled for, and remaining variables are assigned to locations in memory. Given a real world application program, there are enough reads and writes from/to memory to extract meaningful features for classification of optimization levels. Our intuition is that the mnemonics of instructions used to read/write a value are a possible indicators of optimization level used. For example, to load an address and write a local variable at offset **rsp - 0x8**, a compiler may emit the different instructions at a low optimization level compared to instructions at a higher optimization level, as shown in Figure 3.2.

```
mov eax, rsp
sub eax, 0x08
mov [eax], 0x10
```

(a) Low optimization level

```
lea eax, [rsp-0x08]
mov [eax], 0x10
```

(b) High optimization level

Figure 3.2.: x86-64 instructions to load a value

- **Average load/store dependency length.** Similar to the argument above, the number of instructions on which a load/store instruction depends could differ across different optimization levels. For example, Figure 3.2 shows a load instruction depending on 2 instructions in case of low optimization level and 1 instruction in case of high optimization level. Therefore, the average number of such instructions in a binary could be a possible indicator of optimization level.

- **Presence of stack canary.** Stack canaries provide a protection against buffer overflow attacks. Compilers add a stack canary between variables local to a function and the return address on the stack. This prevents return address from being overwritten undetected. At higher optimization levels, a compiler may chose not to add a canary value in stack for a function, if the function is considered safe from buffer overflow attacks. Therefore, we use the percentage of functions which have stack canary added to their stack, as a feature.

- **Presence of *loop/loopx* instructions.** Compilers may choose to use **loop/loopx** instructions for efficient code, at higher optimization levels. Therefore, presence of such instructions is a possible indication of high optimization level.

- **Average number of *rsp* adjustments.** To assign space for function local variables on a stack, compilers emit instructions to adjust pointer to the top of stack. Depending upon the optimization level, a compiler may emit multiple adjustments whenever space is needed, or combine them into as few adjustments as possible. Therefore, we use the average number of stack top pointer adjustments for each function in the binary as a measure to detect the optimization level.

- **Function prologue.** Frame pointer is generally used by compilers for easier addressing of variables and arguments on a function's stack frame. For achieving this, the frame pointer of the previous function is pushed to the stack, and the

current stack top pointer value is moved to the frame pointer. The instructions for this task are present at the beginning of function, usually called *function prologue*. But at higher optimization levels, a compiler may decide to use the frame pointer register for a different purpose, hence ignore addition of function prologue at beginning of a function. Therefore, the percentage of functions in which a prologue is present acts as an indicator of the optimization level.

- **Function calls to constant address.** Call to from a function to another functions may use a value stored in a register or a constant address for modifying control flow. Although not used in any optimizations, we detect percentage of calls made to constant address in a binary program to detect flags such as `-fno-function-cse` in GCC.

- **Number of pops following a function call.** Compilers usually follow conventions when a call is made to a function from within the code. These conventions specify the role of registers across function call, and the role of caller and callee functions. Usually some registers are specified as caller save registers, which must be saved by the caller to preserve their values, since they may be clobbered by the callee function. So, a compiler could choose to push these values to the stack before the call, and pop these values to restore the register values after the call returns. Depending upon the optimization level, a compiler may choose to pop these values as soon as the call returns, or defer popping them until necessary. Therefore, the number of pop instructions just following a function call is a possible indicator for optimization level.

- **Code duplication.** Similar code snippets could be present at multiple places in high-level source code, especially in large code bases. A compiler could choose to perform additional analyses at higher optimization levels to minimize code duplication. We use a code normalization technique, similar to [25], to roughly estimate the amount of duplicate code in the binary.

3.2.2 Feature Extraction

We implement the feature extraction in Python using angr [26]. angr provides a framework to lift a binary to VEX IR, and perform analysis on top of it. First, the binary is loaded without loading its dependent libraries. To iterate over all functions in the binary, we use a list of starting address of functions in the binary, collected from the symbol table present in the binary. For each function, an accurate intra-function Control Flow Graph (CFG) is computed. This CFG is used to compute an intra-function Control Dependence Graph (CDG) and Data Dependency Graph (DDG). All these analyses are provided out-of-the box by angr.

Upon computing these graphs, we iterate over instructions in all basic blocks in each function, and compute values for each feature depending upon the instruction. For example, if the instruction is **rsp - 0x01**, the feature corresponding to stack adjustments is incremented. Similarly, if the instruction is a load/store from/to a memory location, all the instructions on which the address operand and value operand to this instruction depends are taken into account for load/store dependency length and mnemonics for load/store. To find the instructions on which load/store depends, Backward Slicing analysis provided by angr is used, which uses the CFG, CDG, and DDG computed earlier. After computing the features for each individual function, they are summarized into a feature vector for the binary as a whole.

3.2.3 Classification

After computing a feature vector for each binary, the feature vectors dataset is split randomly into two separate sets – 70% for training and 30% for testing. We use a MultiLayer Perceptron to learn a model on the training set because of its advantage for learning complex non-linear models. We perform a 10-fold Grid Search cross validation over different activation functions (namely identity function, logistic function, hyperbolic tan function and rectified linear unit function), number of hidden layers (from 1 to 20) and number of nodes in each hidden layer, to identify the hyper-

parameters best suited for classification. We use a heuristic approach by randomly choosing the number of nodes in each hidden layer between number of input nodes and number of output nodes. This process of randomly choosing number of nodes in each hidden layer is performed 10 times for each value corresponding to number of hidden layers, to diminish the effect of randomness. The cross validation process assigns a mean accuracy to each combination of hyper-parameters, and hence allows to choose the hyper-parameters best suited for classification of data. The classifier trained with the best suited hyper-parameters is used to classify the feature vectors in the testing set. The evaluation script is implemented in Python, using machine learning related functionality provided by scikit-learn [27].

3.3 Evaluation

We evaluate the system on 1,267 binaries – from coreutils, binutils, and SPEC-CPU2006, compiled using GCC and Clang with optimization levels O0, O1, O2, O3, and Ofast. We skip few of them due to long time taken during feature extraction process . The dataset is split randomly into 70% training set and 30% testing set as mentioned above, and statistics are collected on the 30% test set of features. The statistics collection is done for 4 iterations with different training set and testing set splits and hyper-parameters are chosen independently for each iteration. Following are the best hyper-parameters over the grid search performed for each iteration:

- **Iteration 1**:
  - **Activation function**: Hyperbolic tan function
  - **Hidden layers**: 9
  - **Nodes in each hidden layer**: [254, 258, 189, 43, 161, 56, 202, 73, 326]

- **Iteration 2**:
  - **Activation function**: Identity function
  - **Hidden layers**: 9
  - **Nodes in each hidden layer**: [231, 233, 118, 329, 170, 311, 56, 186, 75]
- **Iteration 3**:
  - **Activation function**: Hyperbolic tan function
  - **Hidden layers**: 5
  - **Nodes in each hidden layer**: [132, 326, 342, 330, 227]
- **Iteration 4**:
  - **Activation function**: Hyperbolic tan function
  - **Hidden layers**: 14
  - **Nodes in each hidden layer**: [153, 272, 125, 88, 173, 95, 227, 103, 49, 326, 195, 75, 220, 180]

Tables 3.1 to 3.4 show the confusion matrix for these 4 iterations:

3.4 Discussion

According to results in Tables 3.1 to 3.4, most misclassifications occur between binaries compiled with Clang using optimization levels O2 and O3. This could be explained by difference between effects of these optimization levels on a binary executable. For Clang, optimization level O3 enables all passes enabled by O2, and an extra pass named `-argpromotion` [28]. This extra pass promotes an argument passed to a function from "by reference" to "by value", if it could be proven that the "by reference" argument is only loaded from memory. Since the features we choose fail to capture such modifications in a binary executable, the classification model fails to classify accurately between these classes.

Table 3.1.: Confusion matrix for compiler and optimization detection – Iteration 1. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification.

a	b	c	d	e	f	g	h	i	j	←classified as/actual ↓
41	0	0	0	0	0	0	0	0	0	a=clangO0
0	**37**	*1*	0	0	0	0	0	0	0	b=clangO1
0	0	**15**	*15*	0	0	0	0	0	0	c=clangO2
0	0	*24*	**12**	0	0	0	0	0	0	d=clangO3
0	0	0	0	**38**	0	0	0	0	0	e=clangOfast
2	0	0	0	0	**43**	0	0	0	0	f=gccO0
0	0	0	0	0	0	**37**	*1*	0	0	g=gccO1
0	0	0	0	0	0	*1*	**40**	*2*	0	h=gccO2
0	0	0	0	0	0	0	*1*	**36**	0	i=gccO3
0	0	0	0	0	0	0	0	0	**35**	j=gccOfast

3.5 Summary

To improve heuristics based binary analyses, we present a machine learning based approach to detect compiler and optimization level. Armed with this specific information, heuristics could be tuned to take into account the deterministic changes performed by the compiler during compilation process. We target features specifically for better classification among different optimization levels, which allows better precision on fine grained optimization levels. We evaluate the approach on generic Linux based utility binaries and are able to achieve 83% precision at classification of individual optimization levels along with compiler family.

Table 3.2.: Confusion matrix for compiler and optimization detection – Iteration 2. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification.

a	b	c	d	e	f	g	h	i	j	←classified as/actual ↓
39	0	0	0	0	0	0	0	0	0	a=clangO0
0	**39**	*1*	0	0	0	0	0	0	0	b=clangO1
0	0	**29**	0	0	0	0	0	0	0	c=clangO2
0	*2*	*30*	**1**	0	0	0	0	0	0	d=clangO3
0	0	0	0	**42**	0	0	0	0	0	e=clangOfast
1	0	0	0	0	**47**	0	*1*	0	0	f=gccO0
0	0	0	0	0	0	**33**	*4*	*1*	0	g=gccO1
1	0	0	0	0	0	0	**40**	*1*	0	h=gccO2
0	0	0	0	0	0	0	*1*	**30**	0	i=gccO3
0	0	0	0	0	0	0	0	0	**38**	j=gccOfast

Table 3.3.: Confusion matrix for compiler and optimization detection – Iteration 3. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification.

a	b	c	d	e	f	g	h	i	j	←classified as/actual ↓
40	0	0	0	0	0	0	0	0	0	a=clangO0
0	**38**	*1*	0	0	0	0	0	0	0	b=clangO1
0	*1*	**20**	*20*	*3*	0	0	0	0	0	c=clangO2
0	*1*	*23*	**14**	*2*	0	0	0	0	0	d=clangO3
0	0	0	*5*	**31**	0	0	0	0	0	e=clangOfast
0	0	0	0	0	**42**	0	0	0	0	f=gccO0
0	0	0	0	0	0	**36**	*2*	0	0	g=gccO1
0	0	*1*	0	0	0	0	**32**	0	0	h=gccO2
0	0	0	0	0	0	*1*	*3*	**33**	*2*	i=gccO3
0	0	0	0	*1*	0	0	*1*	0	**28**	j=gccOfast

Table 3.4.: Confusion matrix for compiler and optimization detection – Iteration 4. **Bold** text indicates correct classification. *Italicized* text indicates incorrect classification.

a	b	c	d	e	f	g	h	i	j	←classified as/actual ↓
43	0	0	0	0	0	0	0	0	*1*	a=clangO0
0	**36**	*1*	*2*	0	0	0	0	0	0	b=clangO1
0	0	**1**	*35*	0	0	0	0	0	0	c=clangO2
0	0	*2*	**35**	0	0	0	0	0	0	d=clangO3
0	0	0	0	**37**	0	0	0	0	0	e=clangOfast
0	0	0	0	0	**46**	0	0	0	0	f=gccO0
0	0	0	0	0	0	**37**	*1*	0	0	g=gccO1
0	0	0	0	0	0	0	**26**	*5*	0	h=gccO2
0	0	0	0	0	0	0	0	**40**	0	i=gccO3
0	0	0	0	0	0	0	0	0	**33**	j=gccOfast

Chapter 4. MEMORY LAYOUT DETECTION

4.1 Design

The goal of this analysis is to detect the layout of stack, global, and heap regions of memory in binary programs. In essence, our approach uses address patterns to identify what structure an object at a particular memory location will have. For example, a C struct could be represented as a hierarchical object. We represent the inferred memory layout as a tree structure, called MemTree, separately for each function in a binary program. Therefore, a MemTree represents the view of memory from the standpoint of each function in the binary.

Each MemTree consists of single root node with 4 child nodes, each child node representing a memory region – Global, Heap, Stack, and Unknown. Each of these memory region nodes further contain children nodes, called MemNodes, representing memory locations inside the respective region. At any point during the analysis, the Unknown region contains MemNodes which cannot be assigned to the other 3 memory regions. This helps in preserving uncertainty related to memory objects, until further analyses discover memory regions for them.

The work flow for this analysis consists of multiple stages, represented in Figure 4.1, and described below:

1. **Basic MemTree construction.** The first stage involves constructing a basic MemTree for individual functions, by identifying possible references using 3 sources – load/store operations, heap allocation functions (currently **malloc**, any other memory allocators need to be explicitly provided) and signatures of functions provided by libc. Each node which represents an address in the SSA tree is associated with a MemNode in the MemTree. The SSA tree could have

multiple nodes for the same address, representing aliases, which are not handled at this stage in MemTree.

2. **Address Pattern analysis.** The second stage involves running local abstract interpretation over subtrees corresponding to identified address nodes in SSA tree for each function, to identify address patterns that these nodes could possibly represent.

3. **Restructuring using identified patterns.** In third stage, the structure of MemTree is modified for each function, using address patterns obtained from second stage. This stage adds the following information to MemTree:

   - Whether a MemNode represent an array element;
   - Parent-child relationship between two different MemNodes;
   - Whether two different MemNodes correspond to same address;
   - Identifying memory regions to which MemNodes belong;
   - Merging MemNodes in stack using offsets.

4. **Inter-function propagation.** In the fourth stage, the address nodes and MemNode structures identified for individual functions are propagated across function boundaries by matching parameters passed and arguments received by functions. If new address nodes are identified during this stage and maximum iterations have not been performed [1], the analysis proceeds to the second stage again to process only those nodes. Otherwise, the analysis proceeds to the final stage.

5. **Finalizing stack layout.** In the final stage, we apply few heuristics using discovered offsets and sizes for memory objects, to enhance the layout of the stack obtained after fourth stage.

[1] Maximum iterations help the analysis proceed if a loop occurs. Currently, no such case occurs, so it has been set to a very large value.

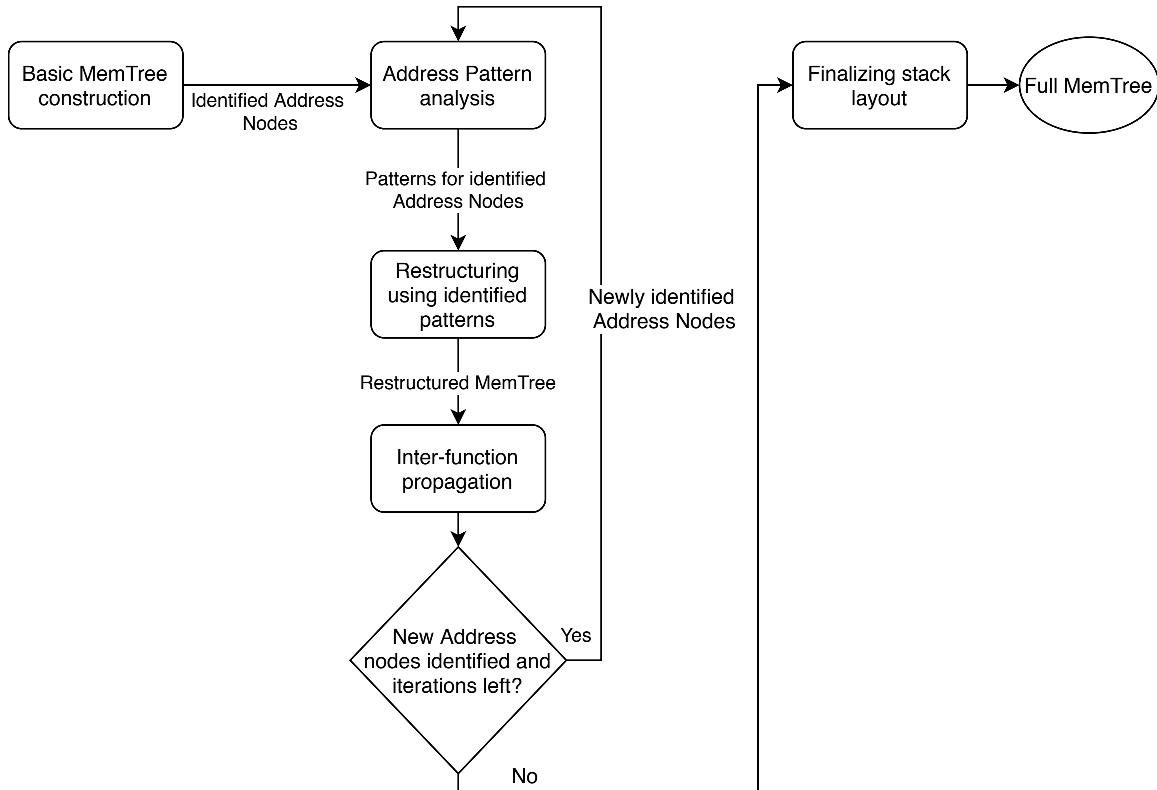


Figure 4.1.: MemTree analysis design

4.1.1 Assumptions

The current implementation of the analysis assumes availability of some auxiliary information. Note that these sources are not a fundamental requirement for the analysis, but rather a limitation of the current framework where those analyses are not yet implemented. With additional analysis passes this information can be recovered from binaries before proceeding to MemTree construction. The information assumed to be available is as follows:

- **Symbol Table.** The analysis uses symbol table to identify function starts and sizes. Recent advancements in function recovery have achieved a high precision of $> 95\%$, and the symbol table may be replaced by such techniques with additional engineering effort.

- **LibC function signatures.** Function calls to dynamically linked libraries, such as the libc, are particularly problematic as the function code is not present within the binary and hence cannot be analyzed. This results in the function being a black box and the analysis cannot reason about arguments or returns from these functions. To overcome this limitation, the analysis is preloaded with a large number of function signatures, automatically extracted from libraries built with debug information, for functions from libc. In theory, this requirement can be removed by analyzing the shared library and caching the results as a part of a preprocessing step. The recovered function signatures can then be reused across analysis of multiple binaries.

- **Argument Count for each function in binary.** We assume the argument count for each function in the binary to be already known. This information is used in inter-function propagation stage of analysis. Without this information, spurious information can propagate across functions in absence of a perfect SSA tree.

- **Calling convention.** We assume that all arguments passed and value returned during a function call use general purpose registers specified by System V AMD64 [29] calling conventions. We currently do not handle arguments passed on stack or floating point registers (**xmm0-xmm7**).

4.2 Implementation

We implement MemTree in Rust leveraging the SSA AST provided by Fafnir. Each stage is implemented as a different pass over the SSA AST, allowing individual passes to run independently whenever required. We describe in detail the basic structure of MemTree and each of the stage below.

4.2.1 MemTree Structure

A MemNode represents a memory object and its structure, at a given memory location. The following fields are stored in each MemNode:

- **Segment Type.** The memory segment to which the address belongs. The possible values are – Global, Heap, Stack, or Unknown.

- **Offset.** For MemNodes belonging to the stack segment, this field stores the offset from stack-top pointer at beginning of the function. For MemNodes belonging to the heap segment, this field stores the offset from the address where heap memory is allocated. For MemNodes belonging to the global segment, this field stores the absolute address of the memory location. For MemNodes belonging to the unknown segment, this field may store an absolute address, if it is calculable, otherwise it stays empty. For a child MemNode, described later, this field represents the offset of its address from the immediate parent MemNode's address.

- **Allocation address.** This field stores the instruction address where heap memory is allocated. This helps in keeping track of MemNodes belonging to heap segment, inside and across functions. For MemNodes in other segments, this field stays empty.

- **Size.** This field stores the size of basic memory object present represented by the MemNode. Initially, the size is assigned using size of value loaded/stored from/in memory, as provided by the underlying framework. Later, it could be refined using other analyses, such as type detection, or using heuristics.

- **Aggregate Size.** This field denotes the aggregate size of object allocated at a memory location in case it is a complex object, such as hierarchical object or array. This field is computed for MemNodes belonging to heap segment using argument to **malloc**, and by further analyses for MemNodes belonging to stack segment.

Each MemNode is connected to one or more nodes in SSA tree using MemRef edges as shown in Figure 4.6. The tree nodes correspond to root of address subtree for the particular MemNode. Since, multiple nodes in the SSA tree could possibly represent the same address, as shown in Figure 4.2, a MemNode can have multiple correspondences in SSA tree. Only Phi nodes in SSA tree are allowed to have multiple outgoing MemRef edges to different MemNodes, since they are capable of representing multiple addresses, as shown in Figure 4.3. Further, MemNodes are connected among themselves, or to Segment Nodes, using MemChild edges as shown in Figure 4.4. A MemChild edge from a Segment Node to a MemNode indicated that the MemNode and all its children belong to the particular memory segment. A MemChild edge from MemNode `P` to MemNode `C` indicate that `C` is a subfield of a hierarchical object located at address represented by `P`. The offset of a child MemNode is always relative to its immediate parent MemNode. The first field of a hierarchical object (at offset 0) and the parent of the hierarchical object are represented using the same MemNode, since they correspond to the same address. To represent an array, a MemNode can be connected to itself using a MemLoop edge, as shown in Figure 4.5.

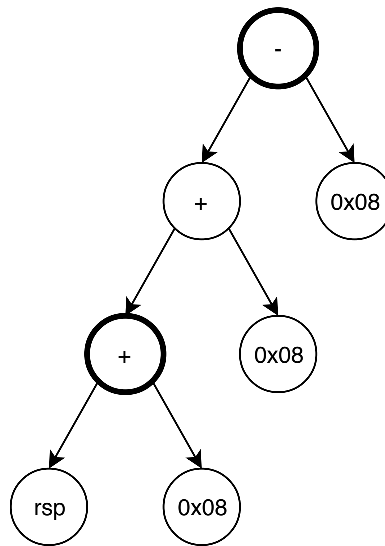


Figure 4.2.: Different nodes (with thick border) representing same address in SSA form

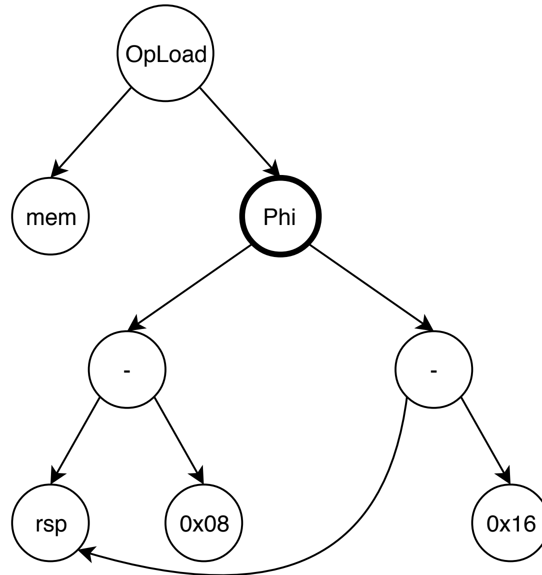


Figure 4.3.: A Phi node (with thick border) could represent multiple addresses

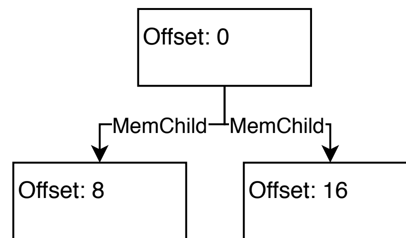


Figure 4.4.: MemNode representing a hierarchical object

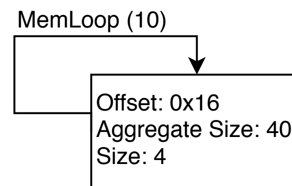


Figure 4.5.: MemNode representing an array

4.2.2 Basic MemTree Construction

Fafnir associates each function with a SSA AST and partially computed CFG. As first step, a basic MemTree intertwined with the SSA tree is constructed for each

function. First, heap allocation functions, like **malloc**, are identified in the SSA tree. For each such call, a MemNode belonging to heap segment is created, assigning it the address of the call-site. The first argument passed to **malloc** is evaluated, if it contains operations only on constants, to assign an aggregate size to the created MemNode. Then, use of stack-top pointer (**rsp** for x86-64 systems) in SSA tree is associated with a MemNode representing the stack-top pointer at beginning of a function. Next, function signatures collected for libc functions are used to identify address nodes in SSA tree, by matching argument passed and parameter received by the functions. This enables identifying address nodes that are passed outside the binary code, and also allows identifying structure of memory objects using the signature information. Lastly, address operand to all load/store operations in SSA tree are associated with MemNodes. Fafnir allows simple backward slicing, due to its tree structure, to find the subtree corresponding to the address operand.

While creating a MemNode, the analysis tries to assign a segment to each MemNode, based on the address subtree, as shown in Algorithm 4.1:

Algorithm 4.1 Procedure to classify MemNode for segments

if address is constant $\in$ data sections – **.data**, **.rodata** and **.bss** **then**
    Assign MemNode to Global segment
else if address is at constant offset from stack-top pointer at beginning of function **then**
    Assign MemNode to Stack segment
else if address is at constant offset from MemNode (say `H`) belonging to Heap segment **then**
    Assign MemNode to Heap Segment
    Add MemNode as child of `H`
else
    Assign MemNode to Unknown Segment
end if

The analysis also does a preliminary merge of MemNodes belonging to Global, Stack, and Heap Segments, based on absolute address, offset and allocation address respectively. Figure 4.6 shows a basic MemTree constructed for a function after this analysis.

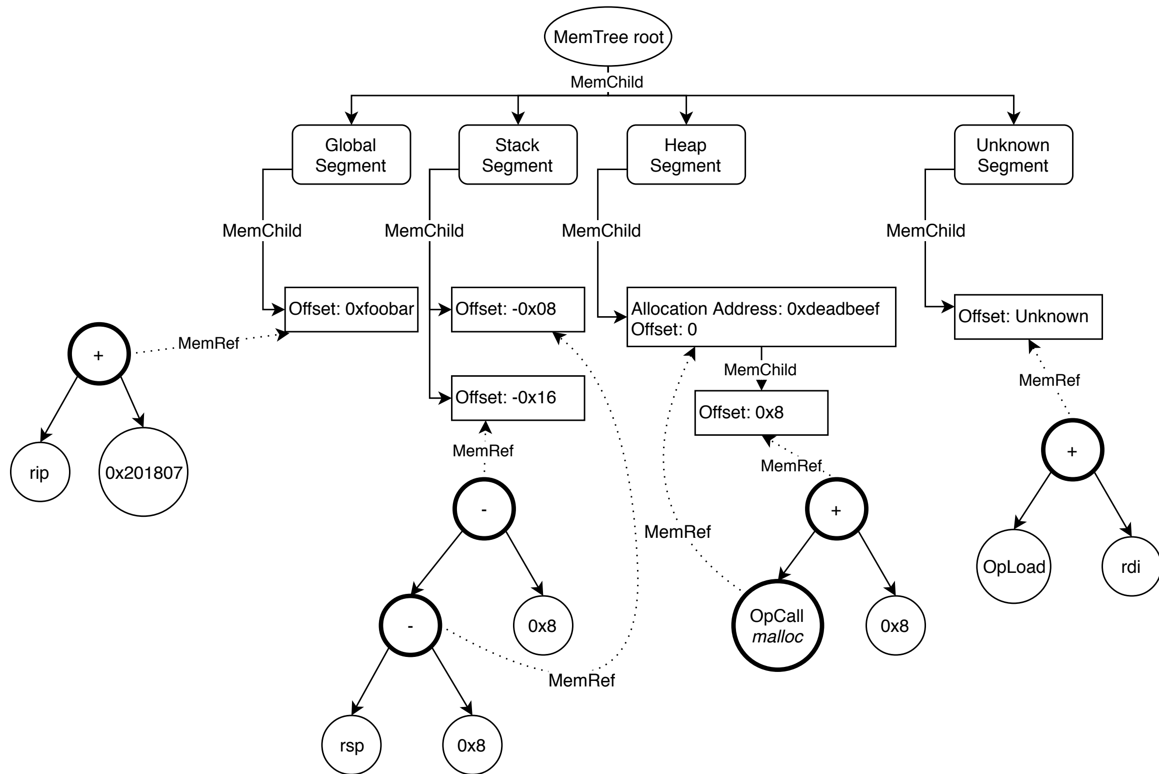


Figure 4.6.: Basic MemTree

4.2.3 Address Pattern Analysis

In the second step, address patterns for MemNodes created during basic MemTree construction or during inter-function propagation, are analyzed to allow more refinements during subsequent steps.

Based on underlying x86-64 ISA, we use four basic components of any address:

1. **Base.** This represents a constant or variable value which forms the base address. It could represent registers like **rsp**, constants like global addresses, or values

loaded from memory. An address node classified as Base represents the starting address of an object in memory. Currently, we use Base pattern to also represent AlignedBase pattern, where alignment is performed using `And` operation on a node representing Base pattern.

2. **Offset.** This represents a constant value relative to the Base, where the memory object is present. For example, in case of address **rsp - 0x10**, the value **-0x10** is identified as Offset.

3. **Index.** This represents a variable value which is used to index into an object such as arrays. This also represents ModIndex pattern, where a Mod operation is applied on node representing Index pattern.

4. **Scale.** This represents a constant value which is multiplied with an Index to scale it. This is useful in identifying stride of successive accesses in case of arrays.

The four basic components are further combined to give 8 more relevant intermediate patterns as follows:

1. **OffsettedBase.** This represents a Base address combined with a constant Offset using addition operation. Offsets can be negative to handle subtraction from Base address. An example of such an address is **rsp - 0x10**, where **rsp** is Base address and **-0x10** is constant Offset.

2. **IndexedBase.** This represents a Base address combined with a variable Index using addition operation. An index could be negative to represent decrement from Base address. An example of such address is **rsp - rax**, where **rsp** is Base address and **-rax** is variable used to index from base.

3. **ScaledIndex.** This represents an Index variable scaled with a constant value using multiplication operation. Examples of this patterns are **rax*4** and **rax**<<**2**, where **rax** is Index and **4** is Scale.

4. **IndexedOffsettedBase.** This represent a Base address combined with a constant Offset and a variable Index using addition operations. An example of such address is **rsp - 0x10 + rax**, where **rsp** is Base address, **-0x10** is constant Offset and **rax** is variable Index.

5. **ScaledIndexedBase.** This represents a Base address combined with a ScaledIndex using addition operation. An example of such address is **rsp - rax*4**, where **rsp** is Base address and **-rax*4** is ScaledIndex.

6. **ScaledIndexedOffset.** This represents a constant Offset combined with ScaledIndex using addition operation. An example of such pattern is **-0x10 + rax*4**, where **-0x10** represents constant Offset and **rax*4** represents ScaledIndex.

7. **ScaledIndexedOffsettedBase.** This pattern combines all basic components and is the most general representation of a memory access. An example of such address is **rsp - 0x10 + rax*4**, where **rsp** is Base address, **-0x10** is constant Offset, **rax** is variable Index and **4** is constant Scale.

Using these 12 patterns, an abstract interpretation is performed over the nodes in SSA tree. This interpretation marks each node in SSA tree with all possible patterns the node could represent. For example, a constant node could represent either an Offset or a Scale. Similarly, value loaded from a location in memory could represent a Base or an Index. The patterns for operands of an operation are combined based upon the operation. For example, a multiplication operation combines two Scale patterns to give a Scale pattern, or an Index pattern with a Scale pattern to give a ScaledIndex pattern. These pattern combining rules have been determined heuristically based on common address patterns observed in binary programs. Appendix A lists down how various operators combine these patterns for performing abstract interpretation. During interpretation, the existing nodes in SSA tree are combined with new nodes representing addition and multiplication operations, so that each pattern captures

a subtree relevant to it. Figure 4.7 shows an example where two different subtrees represent the Base address and Offset from the Base address respectively.

At the end of the interpretation, each node in the SSA tree has a list of associated address patterns it could represent. We discard address patterns associated to nodes in SSA tree that are currently not identified as address nodes. The list of address patterns might contain intermediate patterns that are irrelevant. Assuming that all address patterns have a Base address, we prune out all other patterns except the following – ScaledIndexedOffsettedBase, IndexedOffsettedBase, ScaledIndexedBase, OffsettedBase, IndexedBase and Base. A node could have multiple of these chosen patterns, so they are prioritized according to the complexity of the address pattern, in the order listed earlier.

4.2.4 Restructuring using Identified Patterns

The third step involves restructuring the MemTree based on address patterns from the second step. This step is further divided into sub-steps as follows:

1. **Calculating offsets.** First, we try to resolve offsets values for MemNodes associated with patterns having Offset component in them. This step calculates the offset only if the subtree consists of values that are statically resolvable, such as constants and program counter. Calculating offsets helps in de-aliasing different MemNodes which have the same Base address and same Offset value, but are represented by different nodes in SSA tree. If for a node, Offset turns out to be 0 in its associated patterns, the patterns are removed from the list and corresponding patterns without Offset component are inserted into the list.

2. **Detecting loops.** Based on our observations of address patterns for GCC corresponding to array accesses, we detected two common patterns:

   - **Indexed/ScaledIndexed address.** This consists of addresses where an Index or ScaledIndex is present along with a Base or OffsettedBase. For

example, **rsp - 0x10 + rax*4** represents such an address. Currently, 4 address patterns represent such addresses – ScaledIndexedOffsettedBase, ScaledIndexedBase, IndexedOffsettedBase, and IndexedBase.

- **Recursive Phi.** This consists of a Phi node which resolves to multiple element accesses during runtime on successive interpretation. Figure 4.8 shows an example of such a Phi node. Currently, a combination of Base and OffsettedBase or 2 OffsettedBase patterns, with the same Base address and different Offset values, represent such a recursive Phi node. However, these combinations of patterns could also result from other Phi nodes, such as shown in Figure 4.9. So, we specifically keep track if an OffsettedBase pattern arises due to a recursive Phi node during abstract interpretation.

The patterns listed above detect the presence of an array at a given memory location. These patterns may assist in calculating array strides as well as array boundaries, using limits on indices using further analysis.

3. **Moving MemNodes to appropriate segments.** Based on which segment the Base of an address lies inside, the MemNodes are moved to appropriate segments from the Unknown segment. For example, a MemNode having **rsp** as identified Base would be moved to Stack segment, and a MemNode having a Heap allocated MemNode as identified Base will be moved to Heap segment.

4. **Merge MemNodes on basis of identified Base.** Due to currently present uncertainties in SSA tree across Phi nodes, there could be different MemNodes present for a single Base address. Propagating the Base address across Phi nodes help resolve this uncertainty and merge these MemNodes together.

5. **De-duplicating Stack MemNodes.** More than one MemNode in Stack segment might have the same offset from stack-top pointer at the beginning of the function. These MemNodes represent the same memory object and are merged to a single MemNode.

6. **MemNode adoption.** If a MemNode is associated with one of the following patterns – ScaledIndexedOffsettedBase, IndexedOffsettedBase and OffsettedBase, it is moved to be a child of the MemNode corresponding to the identified Base of the respective pattern. Hence, these patterns help identify aggregate memory objects such as C structs. An exception to this procedure is when the identified Base is **rsp**, in which case the MemNodes are kept as it is, since these patterns would just represent normal stack access.

To merge two MemNodes corresponding to the same memory object, we use a method similar to lazy decomposition in Aggregate Structure Identification [21], keeping in mind the uncertainty present in MemTree and the resulting differences in tree structures.

4.2.5 Inter-function Propagation

The fourth step consists of propagating per function MemTree structure, obtained until this step, across functions by matching arguments passed by a caller function's code and parameters received by the callee function's code. This task is performed for all caller/callee present in the call-graph generated by Fafnir, until either a fix-point is reached, or a maximum number of threshold iterations occur. This task also uses the process of merging two MemNodes, but in two different SSA trees corresponding to different functions. We currently limit this process to propagate information from caller function to callee function only. In C programs, a callee might accept a `(void *)`, which allows caller to pass pointer to any object to the callee. The callee uses other parameters to detect the type of memory object passed and processes it. Since the callee could process the same parameter in multiple ways, current analysis would assign a memory object structure, which is union of structures of all the inferred objects, to the parameter. Propagating this information to the caller in such a case would result in incorrect MemNode structure being inferred for the caller. This effect would affect multiple functions, in case the function accepting pointers to generic

object type is present deep inside the call-graph. Further, inter-function propagation might identify new address nodes in SSA trees. This occurs in case where an address, represented by a pointer in C, is not used inside the function defining it, but is passed to another function which uses it like a reference. These newly identified address nodes are stored, and steps starting from Address Pattern analysis are re-run on these nodes, until either inter-function propagation does not identify any new address nodes, or a maximum number of threshold iterations occur.

The final step involves applying heuristics on Stack segment of MemTree to increase the precision of inferred Stack layout. These heuristics assume that all offsets inside the stack region have been recognized correctly. This step is divided into following sub-steps in order:

- **Find aggregate size by last child's offset.** In case of hierarchical memory object, such as C struct, MemTree analysis may not have inferred correct aggregate size for the corresponding MemNode. This heuristic allows correcting (or inferring) aggregate sizes of such memory objects. If the child at maximum offset for current MemNode is not an aggregate object, the aggregate size of current MemNode is set to `max(current MemNode's aggregate size, child MemNode's offset + size of child MemNode)`. In case, the child at maximum offset is an aggregate memory object, the procedure is first applied recursively to the child MemNode, and the aggregate size of current MemNode is set to `max(current MemNode's aggregate size, child MemNode's offset + aggregate size of child MemNode)`. This heuristic does not infer correct size in case the last sub-field of a hierarchical memory object is not used at all, but allows to infer a minimum aggregate size of such an object. Algorithm 4.2 represents this in a concise manner.

- **Subdue siblings by aggregate size.** After fixing aggregate size of hierarchical memory objects using the heuristic above, this steps finds siblings of current MemNode whose offset lie within offsets covered by current MemNode's offset

Algorithm 4.2 Heuristic procedure for finding aggregate size of hierarchical objects

function FINDAGGREGATESIZE(MemNode M)
    last_child ← child of M at maximum offset
    if last_child is not an aggregate object **then**
        M.aggregate_size ← MAX(M.aggregate_size, last_child.offset+last_child.size)
    else if last_child is an aggregate object **then**
        FINDAGGREGATESIZE(last_child)
        M.aggregate_size ← MAX(M.aggregate_size, last_child.offset+ last_child.aggregate_size)
    end if
end function

and aggregate size. This is recursively performed for each child node of current MemNode. Algorithm 4.3 represents this in a concise manner.

Algorithm 4.3 Heuristic procedure for refining structure of hierarchical objects

```
function SubdueSiblings(MemNode M)
    children ← all children of M in decreasing order of offsets
    for child P ∈ children do
        for child S ∈ children, where S.offset > P.offset do
            if S.offset ∈ [P.offset + P.aggregate_size] then
                Make S as child of P
            end if
        end for
    end for
    for child T ∈ children do
        if T has children then
            SubdueSiblings(T)
        end if
    end for
end function
```

- **Recalculate sizes using offsets.** The initial sizes assigned to MemNodes are based on the size of value loaded/stored from/to memory location represented by the MemNode. Fafnir currently depends on instruction opcodes to obtain this information. But these size values may not always be correct. For example, when using an ASCII character, with 1 byte size, the program may load 4 bytes from an aligned address and perform operations on it to get the 1 byte character value. Performing multiple such operations may be faster to than load from an unaligned address. To fix incorrect sizes due to similar issues, the size of a MemNode is truncated depending on the next higher offset. In case of a hierarchical MemNode, the next higher offset is taken as the smallest

offset among its children. In all other cases, the next higher offset is taken from among the siblings of current MemNode. This procedure is applied recursively to all children of a hierarchical MemNode. Algorithm 4.4 represents this in a concise manner.

Algorithm 4.4 Heuristic procedure for truncating size of basic memory objects

```
function TruncateSize(MemNode M)
    children ← all children of M in increasing order of offsets
    for child P ∈ children - {child at maximum offset} do
        S ← MemNode next to P in children
        P.size ← min(P.size, S.offset-P.offset)
    end for
    lowest_offset_child ← child on M with smallest offset
    M.size ← min(M.size, lowest_offset_child.offset)
    for child C ∈ children do
        TruncateSize(C)
    end for
end function
```

4.3 Evaluation

We evaluate MemTree implementation on coreutils-8.29, binutils-2.29, and SPEC CPU2006 C binaries. These binaries are compiled using GCC version 5.4.0 on an Intel Core i7-6700 CPU x86-64 machine, with O2 optimization level and with debug information. Currently, we fail to analyze 403.gcc from SPEC CPU2006 C binaries, due to an issue while constructing AST. We compare the stack layout inferred by MemTree with the ground truth obtained from the debug information. Before running inference using MemTree, the binaries are first stripped of debug information, using `strip --strip-debug` on a Linux system.

4.3.1 Obtaining Ground Truth

We obtain ground truth for stack layout of functions in a binary program from debug information embedded inside it in DWARF [30] format. DWARF is a standardized format for embedding source level debug information in binary executables. For ELF binaries compiled using GCC with DWARF information, multiple debug sections are embedded – for abbreviations, source line information, information related to unwinding stack, or information related to variables used in each function. We are interested in using information related to local variables in a function to obtain the ground truth. This information is arranged hierarchically with Compilation Units (CUs) at the top level. Each CU corresponds to a compiled source file, and contains multiple Debugging Information Entries (DIEs). Each DIE contains information about a basic entity, such as a function, local variable, parameter to a function, lexical blocks, or data types. DIEs have an associated tag, with prefix `DW_TAG`, which helps in identifying the entity it corresponds to. Each DIE also has a set of attributes (prefixed with `DW_AT`) related to it, which depending upon DIE tag contains information related to the entity such as its name, starting address in case of a function entity, type in case of a parameter or local variable, etc. DIEs corresponding to local variables in a function *may* [2] contain attributes for location description, named `DW_AT_location`, and for type description, named `DW_AT_type`. The attribute `DW_AT_type` points to another DIE which contains information about the size of the type corresponding to the local variable. The attribute `DW_AT_location` is either a DWARF expression which on evaluation would result in location of the local variable, or a pointer to location list which contains multiple DWARF expressions each corresponding to a different range of program counter values inside the function. We use PyELFTools [31] to parse raw debug information in the binary executable and present it in form of objects in Python. PyELFTools also provides a generic framework for operating on DWARF expressions, which we augment by adding new operations introduced in DWARF 5

[2]These attributes are not present at least in cases where the local variable was found to have constant value, and hence was inlined

standard. We use this framework to simulate calculation of location of local variable using DWARF expressions, and obtain the location value if it does not depend on runtime values. Currently, we assume all x86-64 registers except **rip** and **rsp** to be dynamic, but it might be possible to improve precision using static analyses to obtain the values of other registers at a given address in the function. We also assume dereferencing operations on memory addresses to be dynamic. The procedure for extracting location of variables is summarized below:

```
1: function processDWARF
2:     all_DIEs ← getAllDIEs()
3:     for function_DIE ∈ all_DIEs do processBlock(function_DIE)
4:     end for
5: end function
6:
7: function processBlock(block_DIE)
8:     for all child_DIE ∈ children of block_DIE do
9:         if child_DIE corresponds to lexical block then processBlock(child_DIE)
10:        end if
11:        if child_DIE corresponds to variable then
12:            location_containing_DIE ← child_DIE
13:            if child_DIE is abstract instance then
14:                location_containing_DIE ← DIE at concrete instance
15:            end if
16:            if location_containing_DIE does not contain location attribute, or represent a constant value then
17:                skip this DIE
18:            end if
19:            possible_locations ← empty list
20:            if form of location attribute is an expression then
```

```
21:             possible_locations.insert(tryEvaluateExpression(location
    expression))
22:         else if form of location points to a location list then
23:             for location expression ∈ location list do
24:                 possible_locations.insert(tryEvaluateExpression(location
    expression))
25:             end for
26:         end if
27:         type_DIE ← DIE at offset pointed by value of type attribute
28:         type_info ← getTypeInfo(type_DIE)
29:         for location ∈ possible_locations do
30:             assign variable of type_info.size at location in stack
31:         end for
32:     end if
33:   end for
34: end function
35:
36: function GetTypeInfo(type_DIE)
37:   if type is pointer then
38:     type_info.size ← ARCH_PTR_SIZE
39:     type_info.is_Reference ← true
40:   else if type is a base type, such as int or char then
41:     type_info.size ← value of size attribute in type_DIE
42:     type_info.is_Reference ← false
43:   else if ... then
44:     ...
45:   end if
46:   return type_info
47: end function
```

```
48:
49: function TryEvaluateExpression(DWARF expression)
50:     Evaluate expression using augmented framework.
51:     if expression contained static components then
52:         return Offset from frame pointer
53:     else
54:         return None
55:     end if
56: end function
```

The current method for extracting ground truth has a possibility of missing local variables, because it only takes into consideration expression that can be evaluated without any runtime values. Also, DWARF information contains size corresponding to type of value stored at location, not the actual size assigned by the compiler on the stack.

4.3.2 Evaluation Parameters

The layout of stack inferred by MemTree is compared against the ground truth obtained from DWARF information on basis of following parameters:

1. **Correctly Recognized Offsets.** Stack offsets recognized by MemTree present in ground truth

2. **Faulty Recognized Offsets.** Stack offsets recognized by MemTree, but not present in ground truth

3. **Correctly Recognized Sizes.** Size of memory objects recognized correctly, compared to ground truth, at each Correctly Recognized Offset

4. **Correctly Recognized Arrays.** Number of offsets at which array structure is recognized by both MemTree and ground truth

5. **Faulty Recognized Arrays.** Number of offsets at which MemTree detects array structure, but ground truth does not

6. **Correctly Recognized Array Sizes.** Number of offsets at which MemTree recognizes array size correctly (out of Correctly Recognized Arrays)

7. **Correctly Recognized Structs.** Number of offsets at which C struct is recognized by both MemTree and ground truth

8. **Faulty Recognized Structs.** Number of offsets at which MemTree detects C struct, but ground truth does not

9. **Correctly Recognized Struct Sizes.** Number of offsets at which MemTree recognizes C struct size correctly (out of Correctly Recognized Structs)

The current methodology used to compare C structs is incomplete, since it only compares the size of the C struct recognized vs the size of C struct according to ground truth. In the future, we intend to implement a better way of comparing C structs – using children offsets and sizes, in addition to struct sizes. Tables 4.2 to 4.4 show detailed results of binaries on basis of these evaluation parameters. Table 4.5 shows summary of these results for each binary suite.

Table 4.1.: Evaluation parameter abbreviations

TO	Total Offsets present in ground truth
FO	Faulty Recognized Offsets
CRO	Correctly Recognized Offsets
CRS	Correctly Recognized Sizes
TA	Total Arrays – Number of arrays according to ground truth (out of Correctly Recognized Offsets)
FA	Faulty Recognized Arrays
CRA	Correctly Recognized Arrays
CRAS	Correctly Recognized Array Sizes
TSt	Total Structs – Number of C structs according to ground truth (out of Correctly Recognized Offsets)
FSt	Faulty Recognized Structs
CRSt	Correctly Recognized Structs
CRStS	Correctly Recognized Struct Sizes

Table 4.2.: MemTree results for coreutils-8.29. See Table 4.1 for abbreviations

Binary	Offset/Size				Arrays				C Structs			
	TO	FO	CRO	CRS	TA	FA	CRA	CRAS	TSt	FSt	CRSt	CRStS
b2sum	54	465	48	32	10	0	4	0	6	4	6	6
base32	41	307	37	25	5	0	4	0	7	5	7	6
base64	40	307	37	26	5	0	4	0	7	4	6	6
basename	27	250	26	19	1	0	1	0	6	2	6	6
cat	42	277	40	32	1	0	1	0	6	3	6	6
chcon	75	528	66	46	6	0	1	0	16	2	13	12
chgrp	70	508	58	43	1	0	1	0	14	2	12	11
chmod	62	477	55	41	1	0	1	0	12	2	10	9
chown	78	528	65	49	1	0	1	0	14	2	12	11

Continued...

chroot	42	289	38	30	1	0	1	0	6	2	6	6
cksum	30	254	28	19	2	0	1	0	6	3	6	6
comm	42	327	36	21	8	0	6	0	6	2	6	6
cp	165	1077	142	99	11	0	2	0	37	9	26	25
csplit	45	334	33	24	1	0	1	0	9	2	9	9
cut	39	294	36	28	2	0	1	0	6	2	6	6
date	124	590	94	59	9	0	2	0	23	4	20	16
dd	86	521	67	45	3	0	2	0	15	3	13	10
df	123	675	102	80	2	0	2	0	23	2	19	15
dircolors	31	288	30	23	1	0	1	0	6	2	6	6
dirname	27	246	26	19	1	0	1	0	6	2	6	6
dir	115	951	88	65	6	0	4	0	19	3	15	14
du	121	806	100	71	2	0	1	0	14	3	9	8
echo	28	239	26	19	1	0	1	0	6	2	6	6
env	29	238	27	19	1	0	1	0	6	2	6	6
expand	33	265	30	21	2	0	1	0	6	2	6	6
expr	33	386	31	21	1	0	1	0	10	3	8	8
factor	91	610	80	59	3	0	2	0	20	2	8	8
false	27	231	26	19	1	0	1	0	6	2	6	6
fmt	32	297	29	21	1	0	1	0	6	2	6	6
fold	36	268	31	23	2	0	1	0	6	2	6	6
getlimits	37	300	35	21	6	0	1	0	6	3	6	6
ginstall	195	1248	163	115	11	0	3	0	43	9	30	27
groups	29	261	28	21	1	0	1	0	6	2	6	6
head	43	332	38	26	5	0	2	0	7	2	7	7
hostid	28	242	27	19	1	0	1	0	6	3	6	6
id	37	292	33	25	1	0	1	0	6	2	6	6
join	40	368	35	25	2	0	2	0	6	3	6	6
kill	31	261	30	22	2	0	1	0	6	3	6	6
libstd-buf.so	1	8	1	1	0	0	0	0	0	0	0	0
link	28	243	27	19	1	0	1	0	6	3	6	6
ln	66	619	64	47	4	0	2	0	19	2	16	15

Continued...

logname	28	242	27	19	1	0	1	0	6	3	6	6
ls	115	953	88	65	6	0	4	0	19	3	15	14
make-prime-list	1	18	1	1	0	0	0	0	0	0	0	0
md5sum	44	315	42	30	2	0	2	0	8	2	8	6
mkdir	109	686	89	66	3	0	2	0	22	3	17	14
mkfifo	72	552	66	50	2	0	1	0	16	3	13	12
mknod	78	562	70	54	2	0	1	0	16	3	13	12
mktemp	39	336	37	26	2	0	2	0	7	2	7	7
mv	155	1087	134	96	8	0	2	0	37	8	26	25
nice	31	244	28	20	1	0	1	0	6	2	6	6
nl	31	284	28	20	1	0	1	0	6	2	6	6
nohup	34	274	32	21	1	0	1	0	6	3	6	6
nproc	33	261	30	22	1	0	1	0	7	2	7	7
numfmt	49	386	43	34	1	0	1	0	6	2	6	6
od	92	590	71	44	11	2	2	0	9	2	9	7
paste	32	262	31	22	1	0	1	0	6	2	6	6
pathchk	30	246	29	22	1	0	1	0	6	2	6	6
pinky	35	287	33	23	4	0	1	0	7	2	7	7
printenv	27	239	26	19	1	0	1	0	6	2	6	6
printf	65	369	48	34	3	0	2	0	8	3	8	6
pr	84	483	57	39	3	0	1	0	12	2	10	10
ptx	67	539	63	41	1	0	1	0	20	3	8	8
pwd	35	267	34	27	1	0	1	0	11	2	11	11
readlink	41	382	40	31	1	0	1	0	11	2	10	9
realpath	41	403	40	31	1	0	1	0	11	2	10	9
[	60	329	46	34	2	0	2	0	10	2	10	8
rmdir	54	292	42	31	2	0	2	0	8	2	8	6
rm	70	535	58	43	1	0	1	0	13	2	11	9
runcon	33	245	32	25	1	0	1	0	6	2	6	6
seq	72	319	58	42	2	0	2	0	13	2	9	7
sha1sum	46	335	44	31	3	0	3	0	8	2	8	6
sha224sum	68	345	65	49	3	0	3	0	10	2	10	6

Continued...

sha256sum	68	345	65	49	3	0	3	0	10	2	10	6
sha384sum	49	370	46	31	3	0	3	0	10	2	10	6
sha512sum	49	370	46	31	3	0	3	0	10	2	10	6
shred	68	482	62	43	4	0	2	0	10	2	9	9
shuf	54	488	48	34	2	0	2	0	8	3	8	7
sleep	34	265	33	25	1	0	1	0	8	3	8	8
sort	130	947	111	76	4	0	3	0	22	5	15	11
split	53	362	42	29	3	0	1	0	7	4	7	7
stat	107	541	78	61	2	0	2	0	16	2	15	13
stdbuf	60	327	46	34	3	0	3	0	8	2	8	6
stty	74	376	54	38	2	0	2	0	9	2	8	6
sum	40	329	35	24	2	0	1	0	6	2	6	6
sync	30	249	28	20	1	0	1	0	6	2	6	6
tac	37	293	32	23	1	0	1	0	6	2	6	6
tail	61	536	55	39	5	1	1	0	13	2	13	12
tee	33	295	31	21	2	0	1	0	6	2	6	6
test	58	304	44	34	1	0	1	0	10	0	10	8
timeout	39	288	38	29	3	0	1	0	8	4	8	8
touch	115	588	89	54	9	0	2	0	23	4	19	15
tr	40	328	34	23	2	1	1	0	9	2	9	7
true	27	231	26	19	1	0	1	0	6	2	6	6
truncate	34	263	31	22	1	0	1	0	7	2	7	7
tsort	31	308	30	21	2	0	2	0	7	3	7	7
tty	27	233	26	19	1	0	1	0	6	2	6	6
uname	30	239	26	19	1	0	1	0	6	2	6	6
unexpand	32	270	29	21	1	0	1	0	6	2	6	6
uniq	39	374	35	25	2	0	2	0	6	2	6	6
unlink	28	242	27	19	1	0	1	0	6	3	6	6
uptime	50	348	45	29	3	0	1	0	7	3	7	7
users	28	268	27	19	1	0	1	0	6	3	6	6
vdir	115	952	88	65	6	0	4	0	19	3	15	14
wc	49	329	42	32	2	0	1	0	9	2	9	8
whoami	28	242	27	19	1	0	1	0	6	3	6	6

Continued...

who	67	362	52	36	8	0	2	0	9	2	9	7
yes	29	257	28	20	1	0	1	0	6	3	6	6

Table 4.3.: MemTree results for binutils-2.29. See Table 4.1 for abbreviations

Binary	Offset/Size				Arrays				C Structs			
	TO	FO	CRO	CRS	TA	FA	CRA	CRAS	TSt	FSt	CRSt	CRStS
addr2line	1049	7620	825	594	55	1	9	0	119	17	49	31
ar	1066	7868	836	595	58	1	12	0	122	18	45	27
bfdtest1	1028	7443	808	584	53	1	9	0	116	13	47	30
bfdtest2	1028	7459	807	583	52	1	7	0	116	13	47	29
cxxfilt	1050	7611	824	594	54	1	9	0	119	17	49	31
elfedit	15	183	14	9	1	0	0	0	3	2	3	1
nm-new	1069	7711	838	599	54	2	7	0	120	18	42	24
objcopy	1187	9034	925	662	67	1	10	0	134	19	56	37
objdump	1391	8103	507	343	37	1	3	0	61	8	11	11
ranlib	1066	7854	835	603	57	1	11	0	122	18	52	35
readelf	336	1286	273	206	21	0	10	0	17	5	6	4
size	1052	7634	827	595	56	1	9	0	119	18	49	31
strings	1050	7622	824	595	54	1	10	0	118	18	48	31
strip-new	1187	9044	922	655	64	1	9	0	134	22	49	30
sysinfo	12	71	6	4	2	0	2	0	0	0	0	0

Table 4.4.: MemTree results for SPECCPU2006 C binaries. See Table 4.1 for abbreviations

Binary	Offset/Size				Arrays				C Structs			
	TO	FO	CRO	CRS	TA	FA	CRA	CRAS	TSt	FSt	CRSt	CRStS
bzip2	65	309	50	30	14	0	8	0	3	0	0	0
gobmk	1160	9218	598	360	82	4	19	1	24	14	13	13
h264ref	960	3719	878	580	116	0	75	4	49	13	26	0
hmmer	513	2473	316	235	25	0	3	0	4	2	1	0
lbm	26	81	23	18	2	0	0	0	2	0	2	2
libquantum	54	630	49	27	2	0	0	0	10	0	4	4

Continued...

mcf	20	90	19	18	1	0	0	0	0	0	0	0
milc	304	1171	291	210	68	0	13	0	22	6	17	8
sjeng	211	603	176	94	58	0	34	0	19	15	3	3
sphinx	265	1665	243	171	28	0	7	0	5	6	4	4

Table 4.5.: MemTree evaluation summary

Suite	TO	FO	CRO	CRS	TA	FA	CRA	CRAS	TSt	FSt	CRSt	CRStS
coreutils-8.29	5853	42213	5049	3616	285	4	168	0	1083	272	948	860
binutils-2.29	13586	96543	10071	7221	685	13	117	0	1420	206	553	352
SPECCPU2006 C	3578	19959	2643	1743	396	4	159	5	138	56	70	34

From Table 4.5, we observe that MemTree is able to detect that a C struct is present at an offset with 58.3% accuracy, and accuracy for detection of array at a given offset is 41.6%. MemTree recognizes the correct size for 42.9% C structs, but fails to recognize correct size for almost all arrays.

4.4 Discussion

MemTree represents a memory view from a binary program's perspective, which need not necessarily be the same as present in source code. For example, individual elements of an array object declared in source code could be accessed separately, instead of iterating over them, on basis of some other conditional variables. In this case, the binary would see those elements as fields of a hierarchical object instead of an array. For example, Figure 4.10 shows a code snippet from `binutils-2.29` [32], where individual elements of part of an array are accessed. Another limitation of current implementation of MemTree is differentiating between array of hierarchical objects and a hierarchical object with array object as the first field. This limitation occurs because in both cases the Base would be the same. One way to overcome this limitation is to find the stride of iteration. If the stride is equal to the aggregate size of hierarchical object, the object represents an array of hierarchical objects. Contrarily,

if the stride is equal to the size of first element of hierarchical object, the object is a hierarchical object with first element as an array object. This solution requires accurate aggregate size, size and stride information.

Two major sources of information about hierarchical objects are function signatures and inter-function propagation. The reason is impossibility to distinguish between access to a basic object at a stack offset versus access to a field of an hierarchical object in same stack, where both objects are defined in the same function, since both accesses will be relative to the stack pointer. On the other hand, if a hierarchical object is passed as an argument to a function, the access to fields of the object would be relative to the respective argument register. Assuming compiler generated assembly code, it is uncommon to see local variables of caller function being accessed using offsets from passed arguments. Therefore, it would be safe to assume that the object passed as argument is a hierarchical object in such case. The same issue occurs when nested hierarchical objects are present. If a nested hierarchical object is passed a argument to a function, it would be impossible to differentiate between access to fields of the parent hierarchical object and fields of the child hierarchical object in callee function, since both accesses would be relative to the same argument register in the callee. However, if the child hierarchical object is further passed to another function, its structure could be retrieved. Structure detection could be further improved using intrinsics based on additional information, such as function calls. For example, if we know the structure of one argument to **memcpy**, we could propagate the same structure to the another argument.

Current MemTree analysis does not detect sizes for arrays correctly. But since we know the indices used to iterate over arrays, applying other analyses such as VSA could help in determining bounds of these indices, and hence the sizes for arrays.

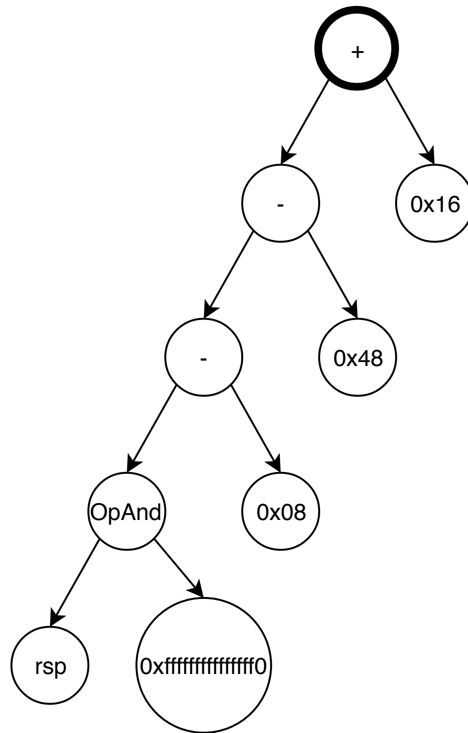
As mentioned earlier, the current ground truth extraction process has a limitation of being able to only extract local variables on stack with statically computable location expressions. In future, we would like to explore better ways of extracting ground truth, one which allows extracting information about variables in heap and global

segment too. A possible way would be to instrument the compilation process to emit this information when IR gets translated to assembly code for a given architecture.

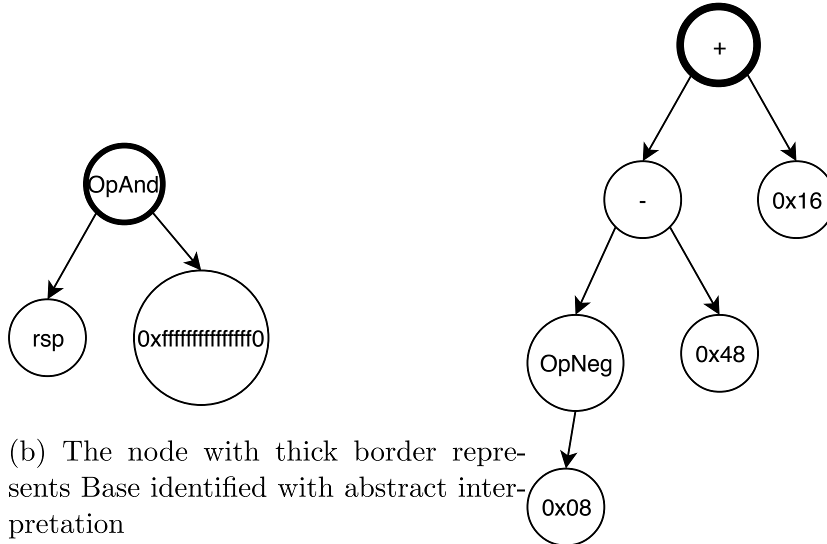
Tables 4.2 to 4.4 show that MemTree recognizes too many false offsets on the stack. One of the reasons for this observation is use of stack to store transient values. Programs may push values in registers on stack and restore them later, in case a value needs to be preserved when a register gets overwritten. Such a case occurs when caller-saved register is used in callee-function. The compiler would choose to push the value in caller-saved register on stack, and restore it after the call returns. These operations are represented as loads and stores on stack in the underlying IR, to provide transparency across architectures. Therefore, MemTree infers them as memory objects. Since DWARF information contains locations for user-defined local variables in source code, these pushes and pops will cause few identified memory locations to be marked false positives.

4.5 Summary

Aiming at tracking of values across loads and stores from memory regions in a binary program, we present an abstract interpretation based approach which infers layout of memory in form of a hierarchical structure called MemTree. Our approach involves finding memory addresses and analyzing possible patterns they could represent. Based on these possible patterns and our observations of existing address patterns in binary programs, we infer structure of object present at a given memory address. Currently, we limit evaluation of this approach only to inferred stack layout, using partial information available from DWARF debug information present in binary executables as ground truth. We also discuss limitations of our approach in detecting structure of memory objects.



(a) The node with thick border represents the complete address



(b) The node with thick border represents Base identified with abstract interpretation

(c) The node with thick border represents Offset identified with abstract interpretation

Figure 4.7.: Result of abstract interpretation

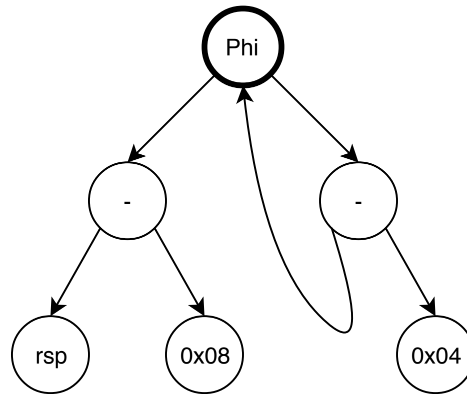


Figure 4.8.: Example of recursive Phi node (with thick border) representing a loop

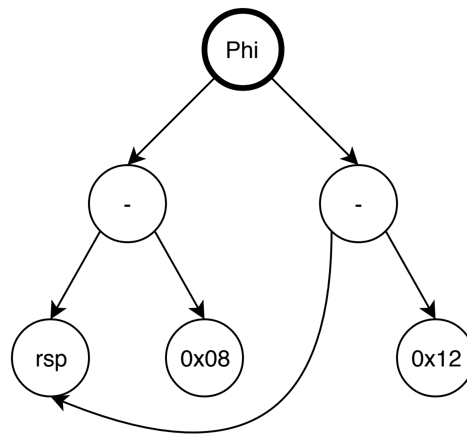


Figure 4.9.: Example of Phi node (with thick border) which gives same patterns as a recursive Phi node, but does not represent a loop

```
static int
shmedia_parse_reg (char *src, shmedia_arg_type *mode, int *reg,
                   shmedia_arg_type argtype)
{
  int l0 = TOLOWER (src[0]);
  int l1 = l0 ? TOLOWER (src[1]) : 0;
  ...
  if (l0 == 'f' && l1 == 'v')
    {
      if (src[2] >= '1' && src[2] <= '5')
        {
          if (src[3] >= '0' && src[3] <= '9'
              && ((10 * (src[2] - '0') + src[3] - '0') % 4) == 0
              && ! IDENT_CHAR ((unsigned char) src[4]))
            {
              *mode = A_FVREG_G;
              *reg = 10 * (src[2] - '0') + src[3] - '0';
              return 4;
            }
        }
        ...
    }
    ...
}
```

Figure 4.10.: Array elements accessed individually

Chapter 5. SUMMARY

We presents two methods to augment current binary analysis techniques on x86-64 binaries, to use targeted heuristics and perform fine grained analysis. We use machine learning based approach to extract accurate information regarding compiler and optimization level used to compile a binary, to help fine tune heuristics for binary analyses. We use an abstract interpretation based approach to extract layout of memory from a binary program's point of view for C binaries, to help existing binary analyses track values across loads and stores from memory.

REFERENCES

REFERENCES

[1] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems*, 32(6):23:1–23:84, August 2010.

[2] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to Analyze Binary Computer Code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 798–804. AAAI Press, 2008.

[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 845–860, Berkeley, CA, USA, 2014. USENIX Association.

[4] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 611–626, Berkeley, CA, USA, 2015. USENIX Association.

[5] Laune C Harris and Barton P Miller. Practical Analysis of Stripped Binary Code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[6] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*. Internet Society, 2011.

[7] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 131–141, New York, NY, USA, 2017. ACM.

[8] Optimize Options – Using the GNU Compiler Collection (GCC). Available at `https://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Optimize-Options.html` (2017/09/17).

[9] A. Nguyen-Tuong, D. Melski, J. W. Davidson, M. Co, W. Hawkins, J. D. Hiser, D. Morris, D. Nguyen, and E. Rizzi. Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge. *IEEE Security Privacy*, 16(2):42–51, March 2018.

[10] Executable and Linking Format (ELF) Specification. Available at `http://refspecs.linuxbase.org/elf/elf.pdf` (2018/07/13).

[11] PE Format. Available at `https://docs.microsoft.com/en-gb/windows/desktop/Debug/pe-format` (2018/07/13).

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[13] Adel Djoudi, Sébastien Bardin, and Éric Goubault. Recovering High-Level Conditions from Binary Programs. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 235–253, Cham, 2016. Springer International Publishing.

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

[15] Ulrich Drepper. ELF Handling For Thread-Local Storage. Available at `https://www.akkadia.org/drepper/tls.pdf` (2018/07/21).

[16] Ulrich Drepper. How To Write Shared Libraries. Available at `https://www.akkadia.org/drepper/dsohowto.pdf` (2018/07/21).

[17] Igor Santos, Jaime Devesa, Félix Brezo, Javier Nieves, and Pablo Garcia Bringas. OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware Detection. In Álvaro Herrero, Václav Snášel, Ajith Abraham, Ivan Zelinka, Bruno Baruque, Héctor Quintián, José Luis Calvo, Javier Sedano, and Emilio Corchado, editors, *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*, pages 271–280, Berlin, Heidelberg, 2013. Springer-Verlag.

[18] I. Firdausi, C. Lim, A. Erwin, and A. S. Nugroho. Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection. In *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, pages 201–203, Dec 2010.

[19] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. Recovering the Toolchain Provenance of Binary Code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2011.

[20] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. Extracting Compiler Provenance from Program Binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 21–28, New York, NY, USA, 2010. ACM.

[21] G. Ramalingam, John Field, and Frank Tip. Aggregate Structure Identification and Its Application to Program Analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 119–132, New York, NY, USA, 1999. ACM.

[22] Gogul Balakrishnan and Thomas Reps. DIVINE: Discovering Variables in Executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[23] Thomas Reps and Gogul Balakrishnan. Improved Memory-Access Analysis for x86 Executables. In Laurie Hendren, editor, *Compiler Construction*, pages 16–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Alan Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, pages 208–223, Berlin, Heidelberg, 1999. Springer-Verlag.

[25] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting Code Clones in Binary Executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[26] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] LLVMs Analysis and Transform Passes – LLVM 6 documentation. Available at `https://llvm.org/docs/Passes.html#argpromotion-promote-by-reference-arguments-to-scalars` (2018/07/16).

[29] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[30] DWARF Debugging Information Format Version 5. Available at `http://www.dwarfstd.org/doc/DWARF5.pdf` (2018/07/05).

[31] PyELFTools. Available at `https://github.com/eliben/pyelftools` (2018/07/05).

[32] Binutils – GNU Project – Free Software Foundation. Available at `https://www.gnu.org/software/binutils/` (2018/07/22).

APPENDIX

APPENDIX

Table A.1.: Pattern name abbreviations

Abbreviation	Pattern name
B	Base
O	Offset
I	Index
S	Scale
OB	OffsettedBase
IB	IndexedBase
IO	IndexedOffset
SI	ScaledIndex
IOB	IndexedOffsettedBase
SIB	ScaledIndexedBase
SIO	ScaledIndexedOffset
SIOB	ScaledIndexedOffsettedBase
-	No Pattern

Table A.2.: Rules for combining address patterns across add, subtract and multiply operations. See Table A.1 for abbreviations

Left Operand Pattern	Right Operand Pattern	OpAdd	OpSub	OpMul
B	B	-	-	-
B	O	OB	OB	-
B	I	IB	IB	-

Continued...

B	S	-	-	-
B	OB	-	-	-
B	IB	-	-	-
B	IO	IOB	IOB	-
B	SI	SIB	SIB	-
B	IOB	-	-	-
B	SIB	-	-	-
B	SIO	SIOB	SIOB	-
B	SIOB	-	-	-
O	B	OB	-	-
O	O	O	O	-
O	I	IO	IO	-
O	S	-	-	-
O	OB	OB	-	-
O	IB	IOB	-	-
O	IO	IO	IO	-
O	SI	SIO	SIO	-
O	IOB	IOB	-	-
O	SIB	SIOB	-	-
O	SIO	SIO	SIO	-
O	SIOB	SIOB	-	-
I	B	IB	-	-
I	O	IO	IO	-
I	I	I	I	I
I	S	-	-	SI
I	OB	IOB	-	-
I	IB	IB	-	-

Continued...

I	IO	IO	IO	-
I	SI	SI	SI	SI
I	IOB	IOB	-	-
I	SIB	SIB	-	-
I	SIO	SIO	SIO	-
I	SIOB	SIOB	-	-
S	B	-	-	-
S	O	-	-	-
S	I	-	-	SI
S	S	S	S	S
S	OB	-	-	-
S	IB	-	-	-
S	IO	-	-	SI
S	SI	-	-	SI
S	IOB	-	-	-
S	SIB	-	-	-
S	SIO	-	-	-
S	SIOB	-	-	-
OB	B	-	-	-
OB	O	OB	OB	-
OB	I	IOB	IOB	-
OB	S	-	-	-
OB	OB	-	-	-
OB	IB	-	-	-
OB	IO	IOB	IOB	-
OB	SI	SIOB	SIOB	-
OB	IOB	-	-	-

Continued...

OB	SIB	-	-	-
OB	SIO	SIOB	SIOB	-
OB	SIOB	-	-	-
IB	B	-	-	-
IB	O	IOB	IOB	-
IB	I	IB	IB	-
IB	S	-	-	-
IB	OB	-	-	-
IB	IB	-	-	-
IB	IO	IOB	IOB	-
IB	SI	SIB	SIB	-
IB	IOB	-	-	-
IB	SIB	-	-	-
IB	SIO	SIOB	SIOB	-
IB	SIOB	-	-	-
IO	B	IOB	-	-
IO	O	IO	IO	-
IO	I	IO	IO	-
IO	S	-	-	SI
IO	OB	IOB	-	-
IO	IB	IOB	-	-
IO	IO	IO	IO	-
IO	SI	SIO	SIO	-
IO	IOB	IOB	-	-
IO	SIB	SIOB	-	-
IO	SIO	SIO	SIO	-
IO	SIOB	SIOB	-	-

Continued...

SI	B	SIB	-	-
SI	O	SIO	SIO	-
SI	I	SI	SI	SI
SI	S	-	-	SI
SI	OB	SIOB	-	-
SI	IB	SIB	-	-
SI	IO	SIO	SIO	-
SI	SI	SI	SI	SI
SI	IOB	SIOB	-	-
SI	SIB	SIB	-	-
SI	SIO	SIO	SIO	-
SI	SIOB	SIOB	-	-
IOB	B	-	-	-
IOB	O	IOB	IOB	-
IOB	I	IOB	IOB	-
IOB	S	-	-	-
IOB	OB	-	-	-
IOB	IB	-	-	-
IOB	IO	IOB	IOB	-
IOB	SI	SIOB	SIOB	-
IOB	IOB	-	-	-
IOB	SIB	-	-	-
IOB	SIO	SIOB	SIOB	-
IOB	SIOB	-	-	-
SIB	B	-	-	-
SIB	O	SIOB	SIOB	-
SIB	I	SIB	SIB	-

Continued...

SIB	S	-	-	-
SIB	OB	-	-	-
SIB	IB	-	-	-
SIB	IO	SIOB	SIOB	-
SIB	SI	SIB	SIB	-
SIB	IOB	-	-	-
SIB	SIB	-	-	-
SIB	SIO	SIOB	SIOB	-
SIB	SIOB	-	-	-
SIO	B	SIOB	-	-
SIO	O	SIO	SIO	-
SIO	I	SIO	SIO	-
SIO	S	-	-	-
SIO	OB	SIOB	-	-
SIO	IB	SIOB	-	-
SIO	IO	SIO	SIO	-
SIO	SI	SIO	SIO	-
SIO	IOB	SIOB	-	-
SIO	SIB	SIOB	-	-
SIO	SIO	SIO	SIO	-
SIO	SIOB	SIOB	-	-
SIOB	B	-	-	-
SIOB	O	SIOB	SIOB	-
SIOB	I	SIOB	SIOB	-
SIOB	S	-	-	-
SIOB	OB	-	-	-
SIOB	IB	-	-	-

Continued...

SIOB	IO	SIOB	SIOB	-
SIOB	SI	SIOB	SIOB	-
SIOB	IOB	-	-	-
SIOB	SIB	-	-	-
SIOB	SIO	SIOB	SIOB	-
SIOB	SIOB	-	-	-

Table A.3.: Summary of rules for combining address patterns across operations

Operations	Interpretation rules summary
OpAdd	All cases are valid, except pattern with Base component cannot be added to another pattern with Base component and Scale can only be added to Scale
OpSub	Same as OpAdd, excluding cases where Base component is present in right operand pattern
OpMul	Only Scale and Index can be multiplied. Special case is IndexedOffset and Scale to handle cases such as `a[5*(i+1)]` in C, since `i+1` will be identified as IndexedOffset initially
OpDiv, OpLsl, OpLsr, OpRol, OpRor	Same as OpMul
OpAnd	Resulting node is Base or Index
OpLoad	Resulting node is Base or Index
OpConst	Resulting node is Scale or Offset. Also possibly Base, if node is known to be a reference
OpZeroExt, OpSignExt, OpNarrow	Remove all patterns from operand patterns which include Base component
OpMod	Resulting node is Index
OpStore	No pattern
OpOr	Resulting node is Index
OpXor	Resulting node is Index
OpNot	Remove all patterns from operand patterns which include Base component
OpCall	Resulting node is Base or Index
*	Resulting node can be any basic component, except for **rip** and **rsp** which are Base