The Pennsylvania State University
The Graduate School

PRACTICAL STATIC BINARY ANALYSIS WITH BLOCK MEMORY MODEL

A Dissertation in
Computer Science and Engineering
by
Sun Hyoung Kim

© 2023 Sun Hyoung Kim

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2023

The dissertation of Sun Hyoung Kim was reviewed and approved by the following:

Gang Tan
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Sencun Zhu
Associate Professor of Computer Science and Engineering

Danfeng Zhang
Associate Professor of Computer Science and Engineering

Dinghao Wu
Professor of Information Sciences and Technology

Chita R. Das
Professor of Computer Science and Engineering
Department Head of Computer Science and Engineering

Abstract

The demand for analyzing binary programs has been increasing. Binary-level pointer analysis is a promising solution for many applications such as constructing high-precision control flow graphs (CFGs). However, the key challenge of pointer analysis is caused by memory accesses; source-code like variables at the binary level are represented by memory accesses usually through registers (indirect memory accesses). Value set analysis (VSA) is the first general-purpose binary-level pointer analysis, yet it is too conservative and does not scale on large binaries.

This dissertation proposes solutions for practical binary-level pointer analysis and generating high-precision CFGs on large binaries. To develop such solutions, our two major insights are 1) a block memory model and 2) Datalog based implementations. In our block memory model, we partition a memory region into a set of disjoint *memory blocks* that are used to address the challenges of memory-level pointer analysis. Datalog is a logic programming language suitable for implementing static analysis rules and provides modularity and high-performance. We have designed three different binary analysis systems: 1) The first system, BPA, uses pointer analysis techniques with the main goal of generating high precision CFGs, 2) BinPointer, which is an extension of BPA and a more general-purpose pointer analysis system, and 3) BinType, an arity-type based approach for efficiently resolving indirect call targets. These three systems' core static analysis components are all implemented in Datalog and each system adopts the block memory model concept in a slightly different way for their own purposes.

First we propose a novel block-based interprocedural pointer analysis called BPA. It assumes a block memory model in which a memory region is divided into disjoint memory blocks and a pointer to a memory block cannot be made to point to other blocks via pointer arithmetic. Taking advantage of the block memory model, BPA only tracks what memory blocks a pointer can point to, but not the offsets of the pointer from the beginnings of blocks. Also, we implemented BPA's pointer analysis for refining indirect call targets of input binaries, which is the major challenge of constructing high-precision

CFG. Our experiments with BPA show that this block-based pointer analysis is effective and gives precise results for CFG construction on real-world binaries.

BPA's design of not tracking offsets within a block may increase efficiency but it could also lead to significant overapproximation. Also BPA was only evaluated by CFG construction precision metrics and it was not yet evaluated for a general-purpose pointer analysis. Hence, we propose BinPointer, a new binary-level interprocedural general-purpose pointer analysis that relies on an offset-sensitive block memory model to achieve high precision while maintaining soundness and reasonable scalability. In detail, while BinPointer still performs pointer analysis based on the block memory model, BinPointer tracks offsets within blocks during pointer analysis, which is the key to improve the precision. This is the major difference of BinPointer from BPA. We also propose a dynamic analysis based strategy for evaluating the soundness and precision of pointer analysis. The idea is to collect runtime memory reference traces of binaries with respect to a set of test inputs and use that as the "pseudo" ground truth to compute precision and recall rates of BinPointer. Such kind of evaluation is missing in prior systems.

Pointer analysis frameworks including BPA and BinPointer in general are still expensive compared to type-based approaches for constructing CFGs. Therefore we propose BinType, a practical type-based framework for constructing high-precision CFGs. BinType is a new signature-matching approach that relies on type inference to improve the signature granularity. Methodology-wise, BinType identifies storage locations of high-confidence types, generates type equivalence relations between storage locations, and propagates types to callsite arguments and function parameters by following the type equivalence relations. This type inference process also leverages the block memory model concept to infer some memory accesses' memory locations that are used for the equivalence relation generation. Our experimental results demonstrate that BinType is much more efficient than BPA while it produces comparable precision results as BPA.

Table of Contents

List of Figures viii

List of Tables x

Acknowledgments xii

Chapter 1
Introduction 1
1.1 Motivation 1
1.2 Thesis Statement 2
1.3 Our Work 3
1.4 Outline 6

Chapter 2
Background 7
2.1 Source-level Pointer Analysis 7
2.2 Challenges of Binary-Level Pointer Analysis 7
2.2.1 Heterogeneity of Binaries 8
2.2.2 Lack of Source Level Information 8
2.2.3 Complexity of Assembly Instructions 9
2.3 Datalog 9

Chapter 3
Related Work 11

Chapter 4
Input Processing and Memory Block Generation 14
4.1 Input Processing: Datalog Fact Generation 14

4.1.1 Program Representation with RTL through Datalog 15
4.1.2 DCFG Refinement 18
4.2 Block Memory Model 18
4.3 Identifying Memory Block Boundaries 19
4.3.1 Stack Layout Analysis and Partitioning 19
4.3.2 Identifying Arguments and Parameters on x86 Binaries 21
4.3.3 Global Memory Region Partitioning 22
4.3.4 Heap Partitioning 24

Chapter 5
BPA: Block-based Pointer Analysis for Identifying Indirect Call Targets 26
5.1 BPA's Block Memory Model Assumptions for Pointer Analysis 26
5.2 BPA's System Overview 27
5.3 Block-based Value Tracking Analysis 29
5.3.1 Memory Block and Chunk Generation 29
5.3.2 Memory-block Access Transformation 31
5.3.3 SSA Transformation 35
5.3.4 Value Tracking Analysis 36
5.3.5 Discovering Indirect Branch Targets 41
5.4 Evaluation 42
5.4.1 Recovering Arity Information 43
5.4.2 AICT Comparison 44
5.4.3 Profiling-based Precision and Recall 46
5.4.4 Performance Evaluation 48
5.4.5 Case Studies 50

Chapter 6
BinPointer: Offset Sensitive Pointer Analysis 51
6.1 BinPointer's System Overview 51
6.2 Pointer Analysis 52
6.2.1 Motivating Example 52
6.2.2 Offset-sensitive Value Tracking Analysis 54
6.3 Implementation 59
6.4 Evaluation Strategy 60
6.4.1 Dynamic Conversion of Memory Addresses 61
6.4.2 Runtime-data based Evaluation Metrics 62
6.4.3 Overapproximation Degree 64
6.5 Evaluation 64
6.5.1 Soundness and Precision 65
6.5.2 Performance Evaluation 66

6.5.3 Comparison with angr's VSA 67
6.5.4 Downstream Applications 68

Chapter 7
BinType: Type-based approach for Identifying Indirect Call Targets **70**
7.1 BinType's System Overview 70
7.2 Arity Background and Implementations 72
7.2.1 Background: Arity Matching 72
7.2.2 Counting Arity 72
7.2.3 Identifying Void/Non-void Return Types 73
7.3 Equivalence Class based Type Inference 74
7.3.1 Preprocessing 74
7.3.2 Coarse-grained Types 75
7.3.3 Type Initialization 76
7.3.4 Equivalence Relations Construction 79
7.3.5 Type Inference 81
7.3.6 Type-based Indirect Call Target Refinement 82
7.4 Caller-based Arity Inference 85
7.4.1 Indirect Call Target Detection 85
7.4.2 Indirect Call Target Refinement for BinType 88
7.5 Evaluations 88
7.5.1 AICT Results 89
7.5.2 Arity Result Details and Case Studies 91
7.5.3 Dynamic Analysis based Soundness Analysis 96
7.5.4 BinType's Efficiency and Comparison with BPA 97

Chapter 8
Discussions, Future Work and Conclusion **99**
8.1 Discussions 99
8.1.1 Memory Blocks for BPA and BinPointer 99
8.1.2 Calling Convention Assumptions for BinType 100
8.2 Framework Extensions 100
8.2.1 Binaries of Different Architectures 100
8.2.2 C++ Binaries 101
8.3 Generating More Fine-Grained and Sound Memory Block Boundaries . 101
8.4 More Precise Type Inference 102
8.5 Summary of Our Work 102

Bibliography **104**

List of Figures

4.1 The major syntax of RTL [1]. 15
4.2 An example for partitioning the stack. 20
4.3 Global region partitioning example. 23

5.1 BPA's System flow. 28
5.2 The syntax of the MBA-IR. 33
5.3 Representative rules of the basic value tracking analysis. 37
5.4 Toy example for demonstrating BPA's mechanisms. 39
5.5 New rule for $*ireg$ to support memory chunks. 40
5.6 Workflow of fixed point recursion for updating a CFG. The components are recursively performed until no more indirect branch targets are found. 41
5.7 Simplified code snippet from `456.hmmer`. 49

6.1 BinPointer's system flow. 52
6.2 Motivating example for the 0-base abstraction. 53
6.3 Simplified RTL syntax in the SSA form. 54
6.4 Value tracking rules defined over instructions. 56
6.5 Runtime-data driven evaluation strategy. 60
6.6 Instructions causing a segmentation fault in `array5` 69

7.1 System flow. 70
7.2 Simplified RTL syntax with the SSA form. 75
7.3 Type initialization. 77
7.4 Equivalence class construction rules. 80
7.5 Type inference from the equivalence relations 81
7.6 Resolving indirect call targets based on TypeArmor's arity matching technique. 82
7.7 Indirect call targets detection based on BinType's type signature matching. 83
7.8 Initializing indexed registers with functions. 88
7.9 Caller-based arity inference. 89

7.10 Modified rules for arity matching with the enforcement of caller-based arity inference 89
7.11 Simplified assembly code snippet from nginx. 95
7.12 Simplified source and assembly code snippet of 445.gobmk. 96

List of Tables

5.1 MBA-IR transformation rules. 34
5.2 Definition of $\mathrm{VSet}(-)$. 37
5.3 AICT evaluation results (for GCC 9.2). 45
5.4 Profiling based precision rates (for GCC 9.2). 47
5.5 Execution time by BPA and VSA. ∞ means timeout (exceeding 10 hours). 48
5.6 Memory consumption by BPA for O2. 48

6.1 A set of maps assumed by pointer analysis. 55
6.2 Definitions of $\mathrm{ValSet}(VExp)$ and $\mathrm{LocSet}(AExp)$. 57
6.3 Precision evaluation results of BPA and BinPointer (-O0 and -O2). . . . 65
6.4 Overapproximation degree reduction rate of BinPointer over BPA (-O2). 66
6.5 Performance evaluations of BPA and BinPointer on large binaries (-O2). 67
6.6 VSA results by ANGR and BinPointer on SVF micro-benchmarks (-O0). 68
6.7 AICT results for large benchmarks. 69

7.1 Definitions of $\mathrm{InferTypeFromExp}(VExp)$ and $\mathrm{GetIReg}(VExp)$, in OCaml-like syntax. 76
7.2 AICT evaluation results of four different systems. The lower AICT number means the higher precision. 90
7.3 Precision increase (%) comparisons. TyInfer and BinType rows show AICT precision increase from Arity to TyInfer and TyInfer to BinType from Table 7.2 respectively. 92
7.4 Arity argument count information statistics on large binaries (-O2). This table shows the number of address-taken functions (AT), indirect calls (ICall), and functions resolved by $\mathrm{ExactParamCount}(_,_)$ (Detected) that expect a specific count of arguments (N) where N = 1 to 5, or 6 or higher. 93

7.5 BinType's type inference statistics on large binaries (-O2). This table shows the number of address-taken functions (AT) whose parameters' types can be precisely inferred and the number of indirect calls (ICall) whose arguments' types can be precisely inferred. The results are in the form of (`l_n` / `m_n` / `r_n`), where `l_n` represents the number of instances for which the inferred type is the pointer type, and `m_n` presents the cases for which the inferred type is the non-pointer type, and `r_n` refers to the total number of arguments/parameters at a position. 94
7.6 AICT, runtime, and memory consumption evaluation results of BPA [2] and BinType on large binaries (-O2) with more than 100K instructions. 97

Acknowledgments

First of all, I would like to express my gratitude to my advisor, Prof. Gang Tan for his mentoring, supports, and advice for my Ph.D study and research. Through his guidance, I have been able to learn how to do research and make progress. His door is always open for discussions. I am truly grateful for his significant amount of time to support and guide me. I would never have been able to made this achievement without his advice and supports.

I would also like to thank my dissertation committee's members Prof. Sencun Zhu, Prof. Danfeng Zhang, and Prof. Dinghao Wu for their insightful comments and efforts. I truly appreciate their time and involvement that improved my research.

In addition, I would like to thank my manager Cody Addison and my colleagues on the Compiler OCG team at Nvidia. I had pleasant experience during my Summer internship and I am very fortunate to work with great engineers currently at Nvidia.

It was my pleasure to meet and work with my lab mates Shen Liu, Dongrui Zeng, Robert Brotzman, Michael Norris, Yongzhe Huang, Ashley Henley, Ashish Kumar, and Xiaodong Jia. Last but not least, I would like to thank my family for their supports for my whole life. Without my family's continuous encouragement and supports, I would not have made it this far.

This dissertation is based on the work supported by Office of Naval Research (ONR) research grant N00014-17-1-2539 and US NSF grant CNS-2243632. The findings and conclusions do not necessarily reflect the view of the funding agency.

Dedication

To my family.

Chapter 1

Introduction

In this chapter, we first discuss the motivation of our research and give a thesis statement. Then, we introduce our work and give an outline of this dissertation.

1.1 Motivation

Building precise and sound control flow graphs (CFG) is critical for security applications, such as control flow integrity (CFI), which constrains attackers' abilities of control-flow manipulations by enforcing a predetermined CFG. The key to strengthen CFI enforcement is to have a fine-grained CFG. In general, more fine grained CFGs provide less freedom to attackers [3–5]. The major challenge of constructing a precise and sound CFG is the existence of indirect calls in the input program. Indirect calls are function calls through register or memory operands. Moreover, high-precision indirect call target determination is also important for any interprocedural static analysis.

Due to the lack of input coverage in dynamic analysis, static analysis is the preferred way to resolve indirect call targets, especially for applications that require sound CFGs such as CFI. Motivated by the importance of high-precision CFGs, source-level static analysis has demonstrated high precision in resolving indirect call targets with the aid of type information [3, 5–12]. In the meantime, the demand for CFI is also critical in commercial off-the-shelf (COTS) binaries and legacy code, where source code is not available. However, precision is relatively low by pure binary analysis, due to the absence of type information and complexity of assembly instructions.

To resolve targets for an indirect call on binary programs, a promising direction is to

utilize a binary-level static pointer analysis, also known as points-to analysis, to decide what functions may be pointed to by the code pointer used in the call. Binary-level pointer analysis brings extra challenges compared to source-level analysis, due to the lack of source-level information as well as the complexity of assembly instructions. Source-level variables at the binary level are represented by memory accesses usually through registers (indirect memory accesses), which complicates pointer analysis. Value set analysis (VSA) [13] is a popular binary-level pointer analysis, but recent works [2,14–16] criticized VSA for being too conservative and interprocedural VSA were unable to scale to large binaries. Therefore, precise, scalable, and sound binary-level pointer analysis techniques are desired.

A precise binary pointer analysis is a desired technique not only for CFG constructions but also for other applications including binary debloating [17–20], vulnerabilities detection [21–23], malware analysis [24, 25], and binary diffing [26–28]. Evaluating binary pointer analysis systems is also important, yet there is no such work for evaluating pointer systems with large binaries. Therefore, a methodology for collecting the correct ground-truth for evaluating binary pointer analysis is needed.

Another popular direction for accurately resolving indirect call targets is a type-based signature matching technique [29–31]. In general, type-based approaches are more efficient than pointer analysis approaches at both source-level and binary-level. However, the type-based approach for analyzing binary programs is less precise than source-level analysis due to the absence of source-level type information. Therefore, it is desired to develop a more precise type-based approach for refining indirect call targets. In detail, precise yet sound type inference for arguments and parameters is needed. One way is to perform dataflow analysis to understand how the arguments and parameters are used in the program, which requires pointer alias information.

1.2 Thesis Statement

As discussed earlier, designing precise, sound, and scalable pointer analysis and CFG construction frameworks for commercial off-the-shelf (COTS) binaries is difficult. The key challenge is how to precisely and correctly infer the relations of pointer aliases through memory accesses, in which the previous research work lacks. Given the challenges, are there any promising pointer analysis and CFG construction methods for complicated and

large binaries?

Thesis Statement. *With a block memory model and Datalog based implementations, it is possible to perform efficient and accurate pointer analysis and type-based techniques on large binaries and generate high precision control flow graphs.*

1.3 Our Work

We have implemented two binary pointer analysis systems, BPA [2] and BinPointer [32], and BinType, a type-based CFG construction platform. The key insight of these systems is their adoption of a block memory model, which is inspired by the CompCert project [33]. In this model, memory is divided into a set of disjoint *memory blocks*. Each block is comprised of a logically cohesive set of memory slots. For example, the stack frame for a function that has an integer local variable and a local array can be divided into two blocks: one for the integer and the other for the entire array. Note that our three systems use similar but slightly different adoptions of the block memory model concept for their own purposes.

BPA is a block-based pointer analysis system with the main purpose of finding indirect call targets of binaries for high-precision CFG construction. BPA adopts the block memory model that partitions memory regions for efficient pointer analysis. In detail, pointer arithmetic is allowed within one block, but the block model assumes that it is not possible to make a pointer to one block point to a different block via pointer arithmetic. For instance, if p points to block b_1, for any offset o, the result of pointer arithmetic $p+o$ must also point to block b_1, not any other block b_2. This approach enables sound pointer analysis, as long as block boundaries are correctly generated. Since adding an offset to a pointer does not change the block the resulting pointer points to, for a pointer BPA tracks only what blocks it might point to, not the offsets the pointer has from the beginnings of blocks. By not tracking offsets, BPA treats a block as a whole, meaning that memory reads and writes through pointers to the block at different offsets are not distinguished. This design enables BPA to be scalable and achieve higher precision CFGs on real-world binaries, compared to state-of-the-art techniques.

BinPointer is an extended pointer analysis system on top of BPA. Here, both BPA and BinPointer adopt the block memory model, but their major difference is that BinPointer

tracks offsets during pointer analysis. The main purpose of BinPointer is to increase the precision of pointer analysis by tracking offsets within memory blocks. BinPointer also aims to achieve reasonable scalability without losing soundness. To accomplish the goals, we propose new pointer analysis abstractions for tracking offsets within memory blocks. To evaluate BinPointer's applicability as a general-purpose pointer analysis, we also propose dynamic analysis based mechanisms to collect pseudo ground-truth. This dynamic analysis is suitable for other kinds of pointer analysis platforms for soundness evaluations. Based on the evaluations, BinPointer achieves higher precision than BPA as a general-purpose pointer analysis.

We also propose BinType, a new type inference approach tailored for generating precise and sound CFGs. It employs an equivalence-class based technique to infer the aliasing relations among variable-like entities so that types of parameters/arguments can be inferred for indirect call target refinement. In detail, our analysis focuses on inferring whether a parameter/argument is of the memory-address type or the non-memory-address type. Values of the memory-address type are used to reference/dereference memory locations, while values of the non-memory address type can never be used for memory accesses. BinType also presents a new technique for determining the exact arity of certain functions, which increases the precision of identifying indirect call targets. While the number of expected parameters in a function can be approximated by analyzing how parameters are used in the function, the binary program analyzer may infer a smaller arity than the actual one when some parameters are unused in a function. BinType's idea is to formulate a must analysis that determines definite callers of a function and, if there is such a definite caller, use the caller's supplied number of arguments to infer the arity of the function. BinType leverages the block memory model concept to develop these equivalence-class based techniques. For example, it utilizes the memory block generations for the stack region to retrieve stack-based alias analysis information on memory accesses to perform the equivalence-class based techniques.

BPA, BinPointer and BinType are all implemented in Datalog, a declarative logic programming language. For binary-level pointer analysis and type inference for CFG construction, to our best knowledge, this is the first time such kind of work is implemented by Datalog. Our frameworks with the block memory model concept are suitable with the nature of Datalog's logic programming paradigm, and we also achieve high efficiency.

Overall, our current and future work make the following contributions:

- Our prototype BPA is a novel binary-level block-based pointer analysis for stripped binaries. It takes as input a stripped binary, and starts with a memory access analysis to divide memory into memory blocks. BPA then performs the block-based value tracking analysis to perform pointer analysis on memory accesses and infer the targets of indirect branches. This analysis is scalable and precise in terms of CFG generation on a set of real-world benchmarks, including SPEC CPU 2k6. Also, our system supports binaries compiled by both GCC and Clang as well as different compiler optimization levels, including -O0 to -O3. Our experiments show that BPA achieves higher precision than the state-of-the-art technique that uses arity-type signature matching techniques for stripped binaries.

- We also propose BinPointer that tracks offsets within the block boundaries during pointer analysis. Offset-tracking mechanisms enable the precision increase of pointer analysis systems. Our experiments show that BinPointer sacrifices some performance compared to BPA but achieves higher precision than BPA in terms of general-purpose pointer analysis.

- We introduce an evaluation method for evaluating binary-level pointer analysis that uses a block memory model. Its key component is a dynamic analysis that maps concrete memory addresses accessed during runtime into block-offset pairs in the block memory model. Runtime memory access traces collected by this component are used to compute the precision and recall rate of binary pointer analysis that uses the block memory model. This component is general and also applicable for evaluating recall rates of other kinds of binary pointer analysis that does not use the block memory model.

- To refine indirect call targets more efficiently, we also propose BinType, a type based approach that is more efficient than BPA and BinPointer. It utilizes the block memory model to infer alias analysis information on memory accesses and uses them to perform equivalence-based arity type inference techniques. Then, the inferred arity types on function callees and indirect callsites are used to perform type based signature matching techniques to refine indirect call targets. Our experiments show that BinType produces similar precision for identifying indirect call targets as BPA whereas BinType shows much more efficiency over BPA.

- BPA, BinPointer and BinType are implemented in Datalog, a logic programming language suitable for static analysis. Our block memory model concept adopted by these systems are compatible with Datalog's implementations.

1.4 Outline

The rest of the document is organized as follows. Chapter 2 introduces background information and challenges, and Chapter 3 summarizes related work. Chapter 4 discusses our input processing and memory block generation process, which our systems rely on. Chapter 5 presents BPA's block-based pointer analysis for binary programs for high-precision CFG construction. Chapter 6 elaborates on offset-tracking techniques for our pointer analysis framework and dynamic-analysis based evaluation strategies for general-purpose binary pointer analysis systems. Chapter 7 discusses our arity-type based approach for efficiently identifying indirect call targets on large binaries. Chapter 8 proposes future directions and concludes this dissertation.

Chapter 2

Background

This chapter provides background information for our research in this dissertation. In detail, we will discuss source level pointer analysis, binary level pointer analysis along with its challenges, and Datalog.

2.1 Source-level Pointer Analysis

Pointer analysis, also known as points-to analysis, is a fundamental building block of program analysis. Most pointer analysis is formulated at the source level or the intermediate-representation (IR) level, where there is a rich set of information such as types that can be exploited for better analysis precision and scalability. Classic pointer analysis for C code include Anderson's algorithm [34] and Steensgaard's algorithm [35]. Recent pointer analysis for LLVM IR code include systems such as DSA [36] and SVF [37]. Several source-level pointer analysis frameworks for Java and C/C++ code [38–42] have been implemented in Datalog, where they have enjoyed benefits of modularity, high-performance, and precise static analysis offered by Datalog.

2.2 Challenges of Binary-Level Pointer Analysis

Taking as input an executable binary file, binary pointer analysis is typically performed on a set of instructions after disassembling the input binary. The goal of pointer analysis for binaries is to infer what memory locations are pointed to by the memory operands

of instructions in the input program. It is also called (memory) alias analysis that aims to capture which memory accesses of different instructions point to the same memory locations. Performing precise, sound, and scalable binary pointer analysis on real-world binaries is challenging, especially compared to source-level analysis, due to multiple reasons.

2.2.1 Heterogeneity of Binaries

For executable binaries, there are multiple types of binary formats such as executable and linkable (ELF) and Windows binaries. Also, different architectures such as x86, x64, and ARM have their own sets of thousands of assembly instructions. Therefore, unifying binaries of different formats and architectures brings additional obstacles, whereas this problem does not exist in source level analysis that unifies the target languages by intermediate representations. In addition, different architectures could exhibit different features, which makes the unification of analysis harder. For example, x86 and x64 have different calling conventions in a way that parameters of functions are typically only passed by memory-locations in x86 whereas they are also passed by registers in x64. This difference prevents heuristic implementations for binary pointer analysis when targeting different architectures of binaries.

2.2.2 Lack of Source Level Information

Source level information such as types and symbols is missing at the binary level, which brings extra challenges to binary pointer analysis compared to source level pointer analysis. For example, a source level analyzer can refer to the local variables that are explicitly declared, whereas a binary analyzer needs to recover them through pointer analysis on memory accesses. Also, a source level analyzer often uses type information to classify the pointers' associated types so that pointer analysis is performed through the type classifications to avoid drastic over-approximation; a previous work [41] argues that type information is necessary for precise analysis. Besides, identifying function boundaries of input programs, which is necessary for interprocedural analysis, is straightforward at source level. Similar design choices are not applicable at the binary level due to the absence of such source level information.

2.2.3 Complexity of Assembly Instructions

As discussed earlier, the key challenge of binary pointer analysis is to infer what memory locations an instruction might access. It becomes more challenging when there is a pointer arithmetic on a register that holds a memory address. For example, suppose there is pointer arithmetic $p + o$ in a loop where o is difficult to compute by static analysis. To pursue soundness, pointer analysis needs to over-approximate the result of the pointer arithmetic, where precision could drastically decrease. Note that the existence of loops make binary analysis more challenging compared to source-level analysis, where loops are well-defined constructs by explicit loop control statements such as **for** or **while**, which offer clear loop boundaries. However, during the compilation process, these structured loops undergo transformations that result in unstructured loop representations in binary code. At the binary level, loops are often implemented using conditional and unconditional jump instructions, altering the flow of execution. Additionally, conducting path sensitivity analysis, which involves tracking possible execution paths through a loop while accounting for conditional branching and CPU flag computations, becomes challenging due to the complexity of low-level control flow instructions and requirement of alias analysis information. Furthermore, compilers' optimizations may involve loop fusion, where multiple loops are merged. These transformations can result in non-linear and unstructured binary code.

The challenge of interleaved data and code complicates binary analysis. In detail, the convergence of data and executable instructions within memory blurs the distinction between them. This issue arises due to multiple reasons including compiler optimizations, memory layout considerations, and the absence of high-level abstractions in binary representations. Even in cases of accurate disassembly, differentiating between data and code remains challenging due to the lack of contextual information that separate code and data. Such challenges are absent in source level analysis, where high-level programming languages enforce clear separation between code and data.

2.3 Datalog

Datalog [43] is a declarative logic programming language and has lately found applications in static analysis [38–40, 44, 45]. It is a subset of Prolog that guarantees program

termination. A Datalog program consists of a set of logical rules in the form of "c :- $a_1, ..., a_n$", where c is the conclusion and a_1 to a_n are assumptions; that is, if assumptions a_1 to a_n all hold, then conclusion c holds. Both the conclusion and assumptions are expressed with *predicates* that describe relations. Consider the following Datalog example about graph reachability.

```
path(x, y) :- edge(x, y).
path(x, z) :- edge(x, y), path(y, z).
```

By the first rule, if there is an edge from x to y, then from x we can reach y. By the second recursive rule, if there exists a node y so that there is a edge from x to y and y can reach z, then x can reach z. A datalog rule without assumptions is called a *fact*. A Datalog execution engine then uses the rules and the facts to derive tuples that are true. For the above example, if the facts are edge(1, 2) and edge(2, 3), then the engine generates path(1, 2), path(2, 3), and path(1, 3).

Predicates are classified into either Extensional Database (EDB) predicates or Intensional Database (IDB) predicates. EDB predicates are defined by facts in external databases and are input to a Datalog program. For the above example, link is an EDB predicate. IDB predicates are defined by logical rules such as the above. A Datalog execution engine then uses the rules and the EDB facts to derive tuples that hold in the IDB predicates. For the above example, if the facts are link(1, 2) and link(2, 3), then the engine generates reachable(1, 2), reachable(2, 3), and reachable(1, 3).

Chapter 3

Related Work

This chapter introduces existing work related to our binary analysis that will be discussed in this dissertation. We present CFG generation work at both source and binary level, as well as binary-level pointer analysis techniques. Also, we describe Datalog based static analysis techniques and type recovery approaches from binary code.

Source level CFG generation Most systems that generate CFGs require source code or source-level information, such as relocation information and debugging information [3, 5–12, 46]. For example, the original CFI [6] allows an indirect call to target all address-taken functions, which are identified by relocation information. Forward-CFI [7] matches indirect calls and functions by the arity information with the help of the GCC compiler. MCFI [8] and TypeDive [10] generate high-precision CFGs by utilizing source-level type information for the matching. Another track of high-precision CFG generation performs static analysis to infer the targets of indirect branches with the help of source-level information. For example, Kernel CFI [9] applies source-level taint tracking to infer what code addresses can flow to what indirect branches. Control Jujutsu [3] employs a source-level alias analysis (DSA, Data Structure Analysis) to construct the CFG. Another CFG construction system [46] starts with compiler-generated meta information including debugging information, performs a binary-level type inference to infer types of function pointers used in indirect calls, and uses the inferred types to build a CFG. Although this work performs binary-level type inference, it relies on source information such as debugging information.

Binary level CFG generation The absence of source code or source-level information prevents systems from generating high-precision CFGs. CCFIR [47] identifies address taken functions at the binary-level and assume any indirect calls can target those functions, which is a coarse-grained way of generating CFGs. TypeArmor [29] proposes a binary level solution by generating an arity-based CFG that allows an indirect call to target any function whose number of parameters is compatible with the number of arguments supplied at the call. While the analysis is efficient and achieves higher precision than CCFIR, the precision is generally lower than source-level approaches. In detail, due to the absence of source-level type information, TypeArmor performs a few over-approximation policies to be conservative when resolving indirect call targets, leading to lower precision; more details regarding these policies are described in Sec. 7.2.1. τCFI [30] and a recent paper [31] incorporate register-width based types upon arity to increase higher precision than TypeArmor but they do not show guarantees of soundness [48]. CALLEE [48] leverages deep neural networks to identify indirect call targets and achieves higher precision than other TypeArmor in general, but the produced CFGs are unsuitable for CFI defense. To our best knowledge, before our work, there has not been practical pointer analysis based techniques or more fine-grained type-based solutions for generating CFGs suitable for CFI applications.

Binary-level pointer analysis Value set analysis (VSA [13]) is the first general-purpose binary-level pointer analysis, widely adopted binary-level pointer analysis in reverse engineering platforms such as BAP [49], ANGR [14] and CodeSurfer [50]. It partitions memory into memory regions (e.g., a stack frame for a function) and tracks the set of numeric values in abstract locations (alocs) via the *strided intervals* abstract domain. The strided interval is in a form of $s[l, u]$, where s is the stride, l is the lower bound and u is the upper bound. For a pointer value, its value set tracks both which memory regions the pointer points to and the *offsets* that the pointer has from the beginnings of the regions. Since those offsets are numbers, VSA has to track the numeric values in storage locations such as registers; e.g., its numeric analysis may decide that `rax` has values 1, 3, 5, etc. It discovers abstract locations during pointer analysis. The computation design of VSA's strided interval is conservative and over-approximated, which often significantly decreases precision and scalability [14, 15]. For example, when there is a pointer arithmetic between two alocs' strided intervals, the result stride is

computed by taking the greatest common divisor of two strides, which can eventually over-approximates value-set results for both intervals. Also, the arithmetic result of lower and upper bounds of the two intervals need to be updated conservatively, which in the worst case (e.g. pointer arithmetic in an unstructured loop) could become $[-2^{31}, 2^{31}-1]$ (for x86-32bit), which is almost the entire memory space. Thus, VSA needs to be customized for individual applications, where the design of abstract locations and the representation of values are the critical components for customization. For example, ANGR adopts the signedness-agnostic-domain to avoid heavily over-approximated results.

BDA [15] aims at a scalable binary-level dependency analysis on a set of sampled paths. While BDA achieves higher precision on dependency analysis than VSA and IDA [51], its path sampling does not cover all paths and cannot guarantee the soundness of the underlying pointer analysis.

Datalog static analysis Pointer analysis frameworks at source level have been implemented in Datalog for Java and C/C++ programs [38–42], where they have enjoyed benefits of modularity, high-performance, and precise static analysis offered by Datalog. A recent work [52] implemented a binary disassembly and reassembly framework in Datalog, which can be used for providing inputs to our tool. To the best of our knowledge, Datalog has not been used to implement practical binary-level pointer analysis or CFG construction frameworks before our work.

Type Recovery from Binary Code Traditional approaches leverage static analysis [14, 49, 51, 53–60] to recover source-level types for assembly instruction operands by inferring how known types can be propagated across different instructions. Another popular research direction employs machine learning techniques [16, 61–65], which can deal with more complicated data structures. While all of them can recover more fine-grained types than BinPointer, to our best knowledge, none of them aimed to recover arity types with the pursue of no false negatives. Thus, existing type inference approaches may not be suitable to CFG applications required by CFI defense.

Chapter 4

Input Processing and Memory Block Generation

Our infrastructure's core static analysis components are implemented in Datalog and utilize memory block boundary information to perform pointer analysis and CFG construction. As an input processing (initialization) step, our systems take an input binary and disassemble it to generate and represent the binary's program instructions in the Datalog compatible format. Then, we use the Datalog formatted instructions to generate boundaries of memory blocks.

4.1 Input Processing: Datalog Fact Generation

To perform our analysis, we disassemble the input binary and process the disassembly results to generate Datalog facts that represent the program and necessary information for the later components in our infrastructure. We require the disassembly being represented by a DCFG, a CFG whose nodes are basic blocks of assembly instructions and whose edges are for the targets of direct branches. During input processing, our infrastructure first converts the disassembled code into an intermediate representation, which is then encoded into Datalog facts. Afterwards, we identify function boundaries and performs DCFG refinements.

$$
\begin{array}{rcl}
v & \in & \text{Integers} \\
sz & \in & \text{Integers} \\
reg & := & \text{EAX} \mid \text{EBX} \mid \ldots \\
flag & := & \text{CF} \mid \text{ZF} \mid \ldots \\
loc & := & \text{PC} \mid reg \mid flag \mid \ldots \\
bvop & := & \text{add} \mid \text{and} \mid \text{shl} \mid \ldots \\
cmp & := & \text{lt} \mid \text{eq} \mid \text{gt} \mid \ldots \\
exp & := & \text{bitvec}(sz,\ v) \mid \text{arith}(bvop,\ exp,\ exp) \\
 & \mid & \text{test}(cmp,\ exp,\ exp) \\
 & \mid & \text{ite}(exp,\ exp,\ exp) \\
 & \mid & \text{load_loc}(loc)\ \mid\ \text{load_mem}(exp) \\
instr & := & loc = exp\ \mid\ \text{Mem}[exp] = exp \\
 & \mid & \text{IF}\ exp\ \text{DO}\ instr
\end{array}
$$

Figure 4.1: The major syntax of RTL [1].

4.1.1 Program Representation with RTL through Datalog

Our infrastructure converts assembly instructions into an intermediate language called RTL [1]. The RTL language is a RISC-like language, with a small set of instructions. An assembly instruction is translated to a sequence of RTL instructions. There are several benefits of translating the input x86 program into an RTL program. First, it simplifies the value tracking rules, which can be designed for a small instruction set, instead of a large set of x86 assembly instructions. Second, it makes our infrastructure easier to support a new architecture in the future, as long as there is a translation from the instruction set of the new architecture to RTL. Therefore this RTL representation brings a promising solution to overcome the challenges of heterogeneity of binaries discussed in Sec. 2.2.1.

Fig. 4.1 presents the major syntax of the RTL language. Locations *loc* represent storage in the CPU, including the program counter, registers (*reg*), CPU flags (*flag*), and others. Expressions are pure and produce values when evaluated. They include bit-vectors of certain sizes (bitvec), common bit-vector operations (arith), comparison between two expressions (test), conditional expressions (ite, if-then-else), and loads from locations and memory. Instructions have side effects. Instruction "$loc = exp$" updates location loc with the value of expression exp; instruction "$\text{Mem}[exp_1] = exp_2$" updates memory at address exp_1 with the value of exp_2. There is also a conditional instruction. However, we do not perform any path sensitive analysis; so we ignore the conditions in

conditional instructions. The input DCFG, after its instructions are translated into RTL, is encoded into Datalog facts.

RTL to Datalog The encoding is compact in that an expression is always assigned the same ID even if the expression appears multiple times (e.g., in different instructions).

We describe the major predicates for encoding RTL instructions and expressions in Datalog. In total, we have 6 predicates to represent 2 instruction types and 4 expression types:

- **set_loc_rtl**($addr$, $order$, loc_eid, src_eid) is designed for the location-update instructions, which modifies location loc_eid with the value of expression src_eid; the instruction is at address $addr$ and with order $order$.

- **set_mem_rtl**($addr$, $order$, mem_eid, src_eid) is designed for the memory-update instructions, which modifies memory at address mem_eid with the value of expression src_eid; the instruction is at address $addr$ and with order $order$.

- **arith_rtl_exp**(eid, $bvec$, exp_l, exp_r) is designed for arithmetic expressions; the expression ID is eid, of which the value is computed by a bit-vector operation $bvec$ between expressions exp_l and exp_r.

- **get_mem_rtl_exp**(eid, mem_exp) is designed for memory-load expressions; the ID is eid, and its value is computed by getting the memory content at address mem_exp.

- **get_loc_rtl_exp**(eid, loc) is designed for location-load expressions; the ID is eid, and the value is computed by getting the value from location loc.

- **imm_rtl_exp**(eid, int) is designed for bit-vector expressions; the ID is eid, and the value is int.

Essentially, RTL instructions and expressions that are not represented by predicates are unnecessary for our systems including BPA due to our design choice of avoiding path sensitivity for better scalability, which is detailed in Sec. 5.3.2. The major complexity lies in the encoding of RTL expressions, which can be nested. For example, in "$e_1 + e_2$" (abbreviation for $\mathrm{arith}(+,\ e_1,\ e_2)$), both e_1 and e_2 are expressions and have their own structures. Such kinds of recursive structures cannot be directly encoded in Datalog,

which lacks support for inductive datatypes as other languages do. The main idea of our encoding is to give unique IDs for all subexpressions and use those IDs to encode expressions one level at a time. For $e_1 + e_2$, the encoding gives some eid to the entire expression, some eid_1 to e_1, and some eid_2 to e_2; then a fact "$arith_rtl_exp(eid, +, eid_1, eid_2)$" is generated to encode equation "$eid = eid_1 + eid_2$"; e_1 and e_2 are encoded similarly, in a recursive way. Another way of interpreting this process is that it treats an expression as a tree structure, gives an ID to every node in the tree, and adds facts that relate nodes to their child nodes. During the encoding, we also detect duplicate subexpressions and reuses ID for them. For $e_1 + e_2$, if e_1 and e_2 are structurally equivalent, the same ID is used for both e_1 and e_2. This duplicate detection happens for all expressions in a program and as a result turns the encoding of trees into DAGs. RTL instructions are straightforward to encode as there is no nesting. Since an assembly instruction is translated into a sequence of RTL instructions, the encoding for each RTL instruction also includes information about the address of the assembly instruction and the order of the RTL instruction in the sequence. Consider the translation example below:

```
Assembly:
  100: mov edx, [ebp-8]
RTL:
  100: edx = *(ebp-8)
Datalog facts:
  set_loc_rtl(100, 1, e1, e2)
  get_loc_rtl_exp(e1, "edx")
  get_mem_rtl_exp(e2, e3)
  arith_rtl_exp(e3, "-", e4, e5)
  get_loc_rtl_exp(e4, "ebp")
  imm_rtl_exp(e5, 8)
```

It shows an example of how an assembly instruction is translated into RTL instructions, and how corresponding Datalog facts are generated. In this example, the assembly instruction is translated to a single RTL instruction. This instruction loads from memory through an indirect address of `[ebp-8]` and stores the loaded value to `edx`. In the Datalog facts, predicate `set_loc_rtl` represents an RTL instruction that loads a value from expression with ID `e2` and stores it to the register with expression ID `e1`, which

represents edx. Note how "*(ebp-8)" is represented through multiple datalog facts: one dereference at the top level, one subtraction operation at the next level, and more for ebp and 8.

4.1.2 DCFG Refinement

To identify function boundaries, we perform heuristic based static analysis, similar to [47] and [29]. With function boundaries identified, the binary is scanned to identify a set of functions whose start addresses appear as constants in the binary's code/data sections. Those functions are determined as Address-Taken (AT) functions and they are potential targets of indirect calls.

After function boundaries are identified, our infrastructure performs a couple of refinements on the CFG, to make the following steps easier to proceed. The first refinement is to reverse the effect of tail-call optimization, which is often performed by an optimizing compiler. A tail call in a function is a call that is the last action of the function. In a tail-call optimization, the compiler turns a tail call into a jump instruction. We classify the following jumps as tail-call jumps: (1) direct jumps targeting function start addresses, and (2) indirect jumps that are not classified as table-jumps. Our infrastructure then replaces each detected tail-call jump to a call instruction followed immediately by a return instruction. Reversing the effect of tail-call optimization is important for interprocedural analysis, which needs to know what edges are interprocedural edges, and is also important for detecting functions that do not return, discussed next.

The second refinement is to add an intraprocedural edge from a call instruction to its follow-up basic block. In static analysis, such intraprocedural edges are added for the convenience of analysis (e.g., summary information for the call can be associated with these edges). However, we do not add such edges for calls to non-returning functions. We treat exit, abort, and assert_fail as non-returning functions.

4.2 Block Memory Model

Designing a sound and precise pointer analysis is known to be difficult even with source-level information. Performing pointer analysis on stripped binaries introduces extra challenges. The first challenge is how to model memory to have a good balance between

scalability and precision. Modeling the memory as an array of one-byte slots is unscalable. The other extreme is to model memory as a whole, meaning that reads and writes at different memory addresses are not distinguished. This, however, is highly imprecise.

We adopt a block memory model, which is inspired by the CompCert project [33]. It used the block memory model to specify the semantics of C-like languages and verify correctness of program transformations in compilers. We observe that the block memory model can be used for performing scalable binary-level pointer analysis, as well as for inferring arity types. In this model, memory is divided into a set of disjoint *memory blocks*. Each block is comprised of a logically cohesive set of memory slots. For example, the stack frame for a function that has an integer local variable and a local array can be divided into two blocks: one for the integer and the other for the entire array.

The block memory model is adopted by all our systems including BPA, BinPointer and BinType, but each system uses the model in slightly different ways for their own purposes. For example, once we generate memory blocks by identifying memory block boundaries, both BPA and BinPointer use them for pointer analysis but follow different pointer arithmetic assumptions of the block model, and BinType leverages them to perform equivalence-based arity type matching techniques.

4.3 Identifying Memory Block Boundaries

At a high level, the data memory has three kinds of disjoint memory regions: stack frames, the heap (for dynamically allocated data), and the global data regions. For each memory region, we analyze the memory accesses in the input program and partitions the memory region into disjoint memory blocks. We next discuss this process in detail.

4.3.1 Stack Layout Analysis and Partitioning

We first define the calling convention assumed for our systems, since calling convention can impact the stack layout. The core of the assumed calling convention is the classic **cdecl** x86 calling convention that most compilers conform to.

For a function, we analyze its code to partition its stack frame into a set of memory blocks. This is similar to how other systems (e.g., [13, 51, 53]) recover variable-like entities on the stack, except that BPA and BinPointer's partitioning must be coarse grained

```
 1.  ; esp=top
 2. push ebp
 3.  ; esp=top-4
 4. mov  ebp, esp
 5.  ; ebp=top-4, esp=top-4
 6. mov  [ebp-4], $1000
 7. mov  eax, [ebp+8]
 8. mov  [ebp-8], eax
 9. lea  eax, [ebp-8]
10. push eax
11. call <store_by_pieces_2>
...
```

Figure 4.2: An example for partitioning the stack.

to uphold the pointer-arithmetic assumption of the block memory model. Therefore, it proceeds in two steps. The first step is about gathering a set of boundary candidates through a stack layout analysis. In the second step, we remove candidates that may split a compound data structure, such as an array or a struct. The remaining candidates are used to partition the stack frame into memory blocks. Note that BinType only requires the first step while BPA and BinPointer require both first and second steps.

Collecting boundary candidates For a function, we analyze its code to determine what stack addresses instructions might use and collects those stack addresses as boundary candidates. The process is similar to how variable-like entities are recovered in previous systems [51, 53]; so we will discuss it only briefly with an example. At a high level, it is an intraprocedural analysis that infers the relationship between registers and the esp value at the function entry (stack top). At every point in the function, it computes a set of equations in the following form: $\{r_1 = \text{top} + c_1, ..., r_n = \text{top} + c_n\}$, meaning that r_1 equals the entry esp value (top) plus constant c_1, ..., and r_n equals top plus constant c_n. If a register does not point to the stack, then no equation for the register is included in the above set. Fig. 4.2 provides a simple example and the stack layout inference result for the first two instructions. At the entrance, esp points to the stack top; after "push ebp", esp gets to be top-4; after "mov ebp, esp", ebp also gets to be top-4. For the rest of the code, the equations for ebp and esp do not change and are not shown in the figure.

With the result of stack layout analysis, our infrastructure then goes through all instructions, extracts stack addresses from instruction operands, and uses them as boundary candidates. For the example in Fig. 4.2, instructions from line 6 to 9 all involve stack addresses. For example, instruction 6 accesses the address $\mathrm{top} - 8$, since ebp is $\mathrm{top} - 4$. Collectively, instructions 6 to 9 produce the following boundary candidates: $\{\mathrm{top} - 8, \mathrm{top} + 4, \mathrm{top} - 12\}$. These collected boundary candidates that can serve as four-byte stack memory-locations are sufficient for BinType's type-based analysis to infer stack-based alias analysis information, where details are explained in Sec. 7.3.1, but BPA and BinPointer require additional analysis to remove some boundary candidates to uphold the block memory model assumption for their pointer analysis.

Removing boundary candidates The boundary candidates collected in the previous step may be in the middle of compound data structures, such as an array or a struct, due to compiler optimizations. Those candidates would be fine for partitioning the stack frame into alocs in VSA, as it does not rely on the assumption that pointer arithmetic stays within alocs; it tracks offsets of pointers and can determine whether pointer arithmetic makes the resulting pointers point to different alocs. However, if BPA or BinPointer partitioned the stack frame with a boundary candidate that is in the middle of a compound structure, pointer arithmetic would cross the boundary of generated blocks, breaking the assumption of our block model. Therefore, such boundary candidates have to be removed for BPA and BinPointer. These systems apply different techniques for removing boundary candidates, which are detailed in Sec. 5.3.1 and Sec. 6.3 for BPA and BinPointer respectively.

4.3.2 Identifying Arguments and Parameters on x86 Binaries

In our assumed calling convention on x86 binaries, the return address is located at offset 0 from the initial stack top (i.e., the ESP value at the beginning of a function) and parameters are located at stack offsets greater than zero, with multiples of four. For example, the third parameter of the function is placed at offset 12. Meanwhile, the arguments of call instructions, including indirect calls, are sequentially pushed onto the stack in the basic block before the callsite. As a result, these arguments are stored in memory slots with consecutive offsets. This information enables us to infer the number

of arguments. In detail, we begin by tracking the offset of a stack pointer from the last instruction before the call instruction and proceed to examine the addresses of memory write accesses. By analyzing their offset values based on stack layout analysis, we determine the number of arguments and the argument's position (order).

4.3.3 Global Memory Region Partitioning

BPA and BinType partition global memory regions (i.e., the .DATA, .RODATA, and .BSS data sections) by analyzing related memory accesses; in contrast, BinPointer relies on symbol tables to partition global memory regions, where more details are explained in Sec. 6.3. Methodology-wise, it is also a two-step process; it discovers block boundary candidates in step 1; in step 2, candidates that are in the middle of compound structures are removed. However, since programs access global regions using patterns different from those accessing the stack, different global-memory partitioning techniques are needed.

Collecting boundary candidates In step 1, block boundary candidates are collected as follows: (i) extract all base addresses ($addr$) from memory-accessing instructions with indexed operands, in the form of "$addr + r$" (constant address plus a register), or "$addr + r * scale$" (constant address plus a register multiplied by a scale factor such as four), or "$addr + r + r' * scale$" (constant address plus a register and another register multiplied by a scale factor such as four); (ii) in addition, we extract constant addresses stored into registers and memory in the program; we call addresses extracted in (i) and (ii) *pointer-arithmetic-base* (PAB) addresses. Then all PAB addresses that fall into the address ranges of global memory regions are regarded as boundary candidates; the intuition is that such an address is likely to be the start address of a compound data structure. This process is similar to how IDAPro decides on the boundary of alocs in global regions.

We next present an example adapted from `445.gobmk` in SPEC2k6 to illustrate step 1. For easy understanding, we present (a) source code and (b) assembly code (Fig. 4.3) of the example (even though our infrastructure works on binaries). The code has an array of three structs; each has a function pointer and an integer flag. The `discard_moves` function iterates through the array, calls each function pointer, and accesses the flag. Then

```c
1  struct discard_rule {
2    void (*condition)();
3    int flag;
4  } rules[] = {
5  {fptr1, 0}, {fptr2, 1}, {fptr3, -1}
6  };
7  void discard_moves(...){
8    int i, user_f;
9    for (i = 0; i < 3; i++){
10     rules[i].condition();
11     user_f = rules[i].flag;
12   }
13   rules[2].condition();
14   gtp(rules);
15 }
16 void gtp(struct discard_rule rules[]){
17   int j = rules[1].flag;
18 }
```

(a) Source code

```asm
[01] <discard_moves>:
[02] mov  ebp, esp
[03] mov  [ebp-4], 0      ; i=0
[04] mov  [ebp-8], 0
[05] Loop:
[06]   mov  edx, [ebp-4]    ; get i
[07]   mov  eax, [$1000+(edx*8)]
[08]   call eax ; call rules[i].condition
[09]   mov  ecx, [$1004+(edx*8)]
[10]   mov  [ebp-8], ecx
[11]   add  [ebp-4], 1
[12]   cmp  [ebp-4], 1
[13]   jl   Loop
[14] mov  eax, [$1016]
[15] call eax    ; call rules[2].condition
[16] push $1000 ; argument passing
[17] call <gtp>
[18]
[19] <gtp>:
[20] mov  ebp, esp
[21] mov  eax, [ebp+8] ; load from parameter
[22] mov  edx, [eax+12] ; load rules[1].flag
[23] mov  [ebp-4], edx ; store to variable j
```

(b) Assembly code

Figure 4.3: Global region partitioning example.

it calls the function pointer at index 2 of the array; finally, it calls `gtp` with argument `rules`, and the flag at index 1 from `rules` is stored into `j`. In correspondence, the assembly code uses pointer arithmetic to access function pointers and flags in the array, starting from global addresses 1000 and 1004, respectively. Also, the constant global address 1000 is stored into an argument. As a result, the step 1 would collect these two addresses as boundary candidates. Note that 1016 is not collected as a candidate even if in instruction [14] (Fig. 4.3 (b)) there is a memory access using that constant address, because it is accessed without any pointer arithmetic and the address is not stored into a register or memory.

However, the boundary candidates produced in step 1 might still be in the middle of compound structures. This may happen due to compiler optimizations. In the example, the compiler decides to use different base addresses to access the two fields in the struct. If we used those two base addresses (1000 and 1004) in step 1 as block boundaries, it would partition the array into two blocks: b_1 with the first function pointer (`fptr1`), and b_2 for the rest of the array. If such blocks were used in determining the targets of the indirect call at instruction [8], our later steps would decide that the indirect-call targets are read from block b_1. The reason is that instruction [7] loads the target for the indirect call through an operand with the base address 1000 and the address is associated with block b_1. Thus, with the assumption that pointer arithmetic stays within blocks, we would determine that only `fptr1` can be the target of the indirect call, since `fptr2` and `fptr3`

would be in a different block. This would be unsound.

Removing boundary candidates In step 2, we filter out addresses that may split a compound data structure. The idea is to estimate for each boundary candidate an address range that starts with the boundary candidate and that should fall into the same data structure. Any boundary candidate that is strictly within another candidate's estimated range should be eliminated, because such a candidate must be in the middle of a compound data structure. Essentially, the range estimation is a data-flow analysis to estimate a set of possible values that can be computed by pointer arithmetic from each boundary candidate.

If the boundary candidate is a category (i) PAB address (see earlier discussion on collecting boundary candidates), the analysis estimates the possible values of the index register used in an instruction with the PAB address. For the example, it decides that the index register edx in instructions [7] and [9] have at least values of 0 and 1 since they are in a loop. As a result, instruction [7] accesses at least the range of $[1000, 1007]$ and instruction [9] accesses at least the range of $[1004, 1011]$. Because 1004 is in the middle of the first range, it is eliminated as a block boundary. For the example, the end result is that only 1000 is used as a block boundary, treating the whole array as a single block. When the boundary candidate is a category (ii) PAB address, we apply data-flow analysis to track pointer arithmetic on the PAB address to related memory accesses, and uses the result to estimate the range. For the example in Fig. 4.3 (b), starting with 1000 in instruction [16], we perform data-flow analysis on registers and discovered stack memory locations (initial candidates) from Sec. 4.3.1 to compute where the global addresses are stored to. After knowing that eax in instruction [22] points to 1000, we generate a filtering range for the data structure to be $[1000, 1015]$; note that even though the instruction accesses only $[1012, 1015]$, we treat it as part of a bigger data structure that starts from the base address 1000. By this address range, 1004 is again selected to be removed from the boundary candidates.

4.3.4 Heap Partitioning

We partition the heap using the allocation-site approach, used in many static analysis (e.g., [66]) for analyzing the heap. In particular, all memory allocated at a particular

allocation site (i.e., a call to memory allocation functions such as *malloc*, *calloc*, and *realloc*) are put into one memory block. Each such block is identified with a unique allocation site ID, determined by the address of the call instruction to a memory allocation function that allocates the block. This means we assume the binary is dynamically linked and stripped. Further dividing heap memory blocks could increase precision, but would come at the cost of analysis overhead.

Chapter 5

BPA: Block-based Pointer Analysis for Identifying Indirect Call Targets

High-precision CFG generation is the key to improving CFI's security strength [3–5]. However, research on CFG construction for stripped binaries is still lacking. Hence, we propose a block-based pointer analysis (BPA) to construct high-precision CFGs on stripped binaries by precisely detecting indirect call targets. It assumes a block memory model and leverages the memory blocks generated by partitioning techniques introduced in Sec. 4.3, so that BPA can achieve a balance among soundness, precision, and scalability. BPA performs a block-based value tracking analysis, the core of BPA for pointer aliases, which relies on the block memory model to perform a fixed point computation to resolve targets of indirect calls.

5.1 BPA's Block Memory Model Assumptions for Pointer Analysis

When BPA adopts a block memory model, a pointer arithmetic is allowed within one block, but the block model assumes that it is not possible to make a pointer to one block point to a different block via pointer arithmetic. For instance, if p points to block b_1, for any offset o, the result of pointer arithmetic $p + o$ must also point to block b_1, not any other block b_2. Therefore, BPA by design eliminates the consideration of pointer arithmetic during points-to analysis. Conceptually, the block model makes two blocks

separate by an infinite amount of space; so adding a finite offset to a pointer cannot make it point to a different block. This infinite-separation view is sound for points-to analyses if blocks are formed properly, for the following reason. Memory blocks should delineate the boundaries of where legal pointer accesses should be; out-of-block accesses imply memory errors (e.g., accessing an array out of bound). Since program behavior after a memory error is undefined, a points-to analysis should capture only those points-to relations in executions without memory errors. Therefore, when performing a points-to analysis, we can safely assume pointer arithmetic does not make a pointer go outside of a block. This assumption of memory-safe executions is also in previous work of formalizing points-to analysis [67, 68].

One key benefit of the block model is that it enables BPA to not track offsets during its points-to analysis. Since adding an offset to a pointer does not change the block the resulting pointer points to, for a pointer BPA tracks only what blocks it might point to, not the offsets the pointer has from the beginnings of blocks. By not tracking offsets, BPA treats a block as a whole, meaning that memory reads and writes through pointers to the block at different offsets are not distinguished. This enables BPA to be much more scalable than previous binary-level points-to analysis such as VSA. Furthermore, this design also accommodates CFG construction with good precision, as in general it matches well with how code addresses are stored and retrieved from memory. As an example, suppose a program stores into a memory block two function pointers: fp_1 stored at $p + o1$ and fp_2 stored at $p + o2$, assuming p points to the block; the program then performs an indirect call using a function pointer retrieved from $p + o3$. BPA's analysis then ignores all those offsets and decides the block has two function pointers (fp_1 and fp_2) inside and the indirect call can target either one of them. This overapproximation yields good results on stripped binaries, as our experiments show.

5.2 BPA's System Overview

BPA is designed for resolving indirect-branch targets in stripped binaries by a block-based pointer analysis. It takes as input a binary program, and performs a disassembly to produce a CFG that contains targets for direct branches, but not indirect branches. The targets of indirect branches are then determined by BPA's pointer analysis. Fig. 5.1 describes the workflow of BPA, which consists of three major components: input processing,

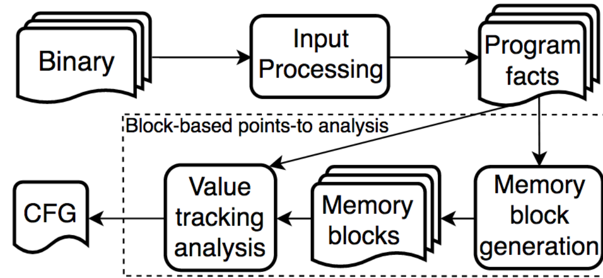

Figure 5.1: BPA's System flow.

memory-block generation, and value-tracking analysis.

Input processing This component is described in Sec. 4.1. To summarize, it generates facts encoded in Datalog to represent the input disassembly. It requires the input disassembly being represented in a Direct Control-Flow Graph (DCFG), which includes edges for direct branches but not indirect branches. This component first translates the input's assembly instructions into an intermediate representation. Then, it identifies function boundaries. Based on function boundaries, functions whose addresses are taken are identified, which can be considered candidates of indirect call targets. This component's outputs serve as inputs for memory block generation and value tracking analysis components.

Memory-block generation As discussed in Sec. 4.3, this component performs a memory-access analysis to generate memory blocks from the DCFG in all memory regions, including data sections, function stack frames, and the heap. It computes block boundaries to partition each memory region. For different memory regions, we partition them at different granularity levels, for balancing between precision and scalability. In addition to the techniques mentioned in Sec. 4.3, BPA adds more techniques for stack and global region memory, which is detailed in Sec. 5.3.1.

Block-based value tracking analysis This component takes as input the DCFG, the address-taken functions, and the generated memory blocks to add new control-flow edges discovered for indirect branches into the CFG. In detail, this component tracks values over abstract locations (registers and memory blocks) according to the input CFG and determines the targets of indirect branches. First, it transforms the program into

a memory-block access intermediate representation. Then, it performs a fixed point computation that recursively interleaves three steps. The first step is to transform the use of registers into the Single-Static-Assignment form based on a CFG by adding ϕ-instructions; the second step performs pointer analysis to infer points-to relations based on the current CFG with the SSA form; the final step discovers targets of indirect branches (indirect calls and jumps) and adds them to the CFG. Since both SSA transformation and pointer analysis depend on the CFG, the three steps are mutually recursive. We implement all three steps by Datalog rules and Datalog's execution engine computes a fixed point until no new points-to relations and CFG edges are discovered. As the total number of indirect-branch targets is finite, there exists a fixed point to guarantee termination. Note that BPA is a context-insensitive interprocedural pointer analysis. It supports flow sensitivity on registers through SSA, but is flow insensitive on memory locations.

5.3 Block-based Value Tracking Analysis

BPA's value tracking analysis component takes as inputs the preprocessed disassembly facts such as a DCFG with RTL instructions and address taken functions, and memory blocks. Then, it performs a fixed point computation that recursively interleaves three steps. The first step is to transform the use of registers into the Single-Static-Assignment form based on a CFG by adding ϕ-instructions. Note that registers in the instructions are SSA formed. The second step performs interprocedural pointer analysis to infer points-to relations based on the current CFG with the SSA form. The final step discovers targets of indirect branches (indirect calls and jumps) and adds them to the CFG. Since both SSA transformation and pointer analysis depend on the CFG, the three steps are mutually recursive. We implement all three steps as Datalog rules and Datalog's execution engine computes a fixed point until no new points-to and CFG edges are discovered.

5.3.1 Memory Block and Chunk Generation

BPA uses the memory block boundary generation techniques discussed in Sec. 4.3 to partition the memory regions of stack, global, and heap regions. Upon the mentioned techniques, this subsection further discusses how BPA removes stack boundary candidates

from the collected ones using the techniques in Sec. 4.3.1 to generate memory blocks for stack region. As previously discussed, BPA requires coarse-grained memory block boundaries for sound pointer analysis. Here we also introduce a memory chunk concept that is only applied for global regions and aims to help increasing BPA's precision for identifying indirect call targets.

Removing boundary candidates for stack regions BPA adopts the following set of heuristics for deciding what boundary candidates from the first step should be kept: (1) $\mathrm{top} + 0$ is always a boundary as the return address is stored from top to $\mathrm{top} + 3$ in x86; (2) $\mathrm{top} + c$, where c is positive, is a boundary as in most calling conventions of x86 such stack slots are used to pass parameters; (3) $\mathrm{top} + c$, where c is negative, is kept only if it is stored into a general-purpose register or a memory location. Category (3) addresses are kept because programs often store the base address of a compound structure into a general-purpose register or a memory location, before accessing the components in that structure; addresses extracted in this way approximate the base addresses of compound structures. We note that IDAPro [51] also uses this observation to perform type recovery (although it considers only movement into registers, not memory).

For the example in Fig. 4.2, only instruction at line 9 is moving a stack address to a register; therefore, only $\mathrm{top} - 12$ is in category (3). This example was adapted from `403.gcc` in SPEC2k6. By inspecting the source code, we find that instructions 6 to 8 are accessing different fields of the same struct and $\mathrm{top} - 12$ is the base address of the struct.

At last, the remaining boundary candidates are used to partition the stack frame. Note that our stack partitioning is conservative in a few ways. First, the stack layout analysis in the step that collects boundary candidates may not discover all possible stack accesses in instructions; e.g., it is intraprocedural and tracks data flow of stack addresses through registers, but not memory. Second, the heuristics in the step for removing boundary candidates may fail to discover boundaries of some compound data structures; as a result, two compound data structures may be put into the same memory block. Being conservative harms the precision and scalability but not soundness, as it leads to a more coarse-grained partitioning; coarse-grained partitioning has less chance to violate the pointer-arithmetic assumption of the block memory model.

Memory chunks for global regions For global regions, with block boundaries determined by remaining PAB addresses that are generated by using the techniques introduced in Sec. 4.3.3, BPA further splits blocks into *memory chunks*. The motivation of introducing memory chunks is based on the fact that there are many read-only constants in the global region and programs often use constant addresses to directly access these read-only constants. To increase analysis precision, BPA uses these constant addresses to partition blocks into chunks. Reading from a chunk would return values only from the chunk, when the surrounding block is read-only. For example, the precision improvement can be seen from instructions [14-15] of Fig. 4.3 (b); it reads from a memory chunk, and thereby the indirect call at instruction [15] only calls fptr3, which increases precision compared to reading from the entire block that contains the chunk. Note that pointer arithmetic can access different chunks within a block; so memory chunks do not carry the pointer-arithmetic assumption. This makes value set tracking more complicated; as we will discuss in Sec. 5.3.4, when a memory block gets updated, the analysis needs to update the values of all memory chunks inside the block.

5.3.2 Memory-block Access Transformation

BPA translates the RTL program into a memory-block access intermediate representation (MBA-IR). This translation improves the runtime efficiency of the next step, value tracking analysis, in two ways. First, memory accesses in the RTL program are converted into accesses to memory blocks/chunks, avoiding repeating the process that determines the corresponding memory blocks/chunks for memory accesses. Second, since BPA's value tracking analysis tracks only part of the program state, the translation also abstracts away unnecessary RTL instructions and expressions. For example, since BPA avoids path sensitivity for scalability, it does not track path conditions. As a result, instructions, expressions, and RTL locations that are related to path conditions are translated away. Moreover, constants that are not function start addresses or memory block/chunk start addresses are abstracted away, for the reason that an internal constant address of a block cannot be used to compute the start address of a different block, by the pointer-arithmetic assumption of the block memory model.

MBA-IR's syntax is defined in Fig. 5.2. MBA-IR has five types of instructions:

(1) register-update instructions, $reg \leftarrow Exp$;

(2) memory-location-update instructions, $mLoc \leftarrow Exp$;

(3) indirect memory-update instructions, $*reg \leftarrow Exp$;

(4) nondeterministic memory-update instructions, $(mLoc \ \vee \ *reg) \leftarrow Exp$;

(5) and no-ops, SKIP.

A register-update instruction computes the value of Exp and stores it into reg. Note that we also treat the program counter as a reg. A memory-location-update instruction has a similar effect except that the destination is a memory block/chunk; recall that BPA uses memory chunks to increase the analysis precision on global data regions. An indirect memory-update instruction ($*reg \leftarrow Exp$) stores the computed value of Exp to a memory location pointed to by reg. A nondeterministic memory-update instruction directly updates a memory location or indirectly updates a memory location through a register; such nondeterminism is necessary to achieve soundness of the MBA-IR translation. Note that a nondeterministic memory-update instruction cannot directly update more than one memory block, which will be explained during the description of translation rules. An expression, Exp, can be (1) a register, which produces the value stored in the register when evaluated, (2) a memory location, which produces the value stored in the memory location when evaluated, (3) a dereference operation on a register ($*reg$), which loads the value stored in the memory location pointed to by reg, (3) the memory address of a memory location ($\&mLoc$), (4) a function start address ($\&func$), or (5) a nondeterministic choose expression ($Exp \ \vee \ Exp$) to represent a nondeterministic expression evaluation for achieving translation soundness.

RTL to MBA-IR The core idea is to translate address expressions used in RTL memory instructions into MBA-IR memory locations (i.e., memory blocks and chunks). This translation is written as $\mathrm{TransAddr(a)}$ and proceeds by pattern matching address a. Before we explain the rules, we first discuss notation. We represent the results of memory block generation as four functions: gblk(c), gchk(c), sblk(f, c), and hblk(c). In particular, gblk(c) returns the global memory block for global memory address c; gchk(c) returns the global memory chunk for global memory address c; sblk(f, c) returns the stack memory block for function f at its stack frame offset c; and hblk(c) returns the heap memory block for a `malloc`-family function call site c. In addition, we use func(c) for mapping the start address c of function $func$ to $\&func$.

$$
\begin{array}{rcl}
\textit{func} & \in & \text{Functions} \\
\textit{gBlk} & \in & \text{Global memory blocks} \\
\textit{sBlk} & \in & \text{Stack memory blocks} \\
\textit{hBlk} & \in & \text{Heap memory blocks} \\
\textit{gchk} & \in & \text{Global memory chunks} \\
\textit{reg} & \in & \text{Registers and program counter} \\
\textit{mLoc} & := & \textit{mBlk} \mid \textit{sBlk} \mid \textit{hBlk} \mid \textit{gchk} \\
\textit{Exp} & := & \textit{reg} \mid \textit{mLoc} \mid *\,\textit{reg} \mid \&\textit{mLoc} \mid \&\textit{func} \\
 & \mid & \textit{Exp} \vee \textit{Exp} \\
\textit{ins} & := & \textit{reg} \leftarrow \textit{Exp} \mid \textit{mLoc} \leftarrow \textit{Exp} \mid *\,\textit{reg} \leftarrow \textit{Exp} \\
 & \mid & (\textit{mLoc} \vee *\textit{reg}) \leftarrow \textit{Exp} \mid \text{SKIP}
\end{array}
$$

Figure 5.2: The syntax of the MBA-IR.

Table 5.1 formalizes the translation. We write $\mathrm{TransAddr(a)}$ for translating address expressions; it is designed according to address a's pattern. In total, there are four patterns, which correspond to the four cases of x86 address-mode operands: (1) c, (2) $c + \textit{reg}$, (3) $c + \textit{reg} * \textit{sc}$, and (4) $c + \textit{reg} + \textit{reg}' * \textit{sc}$. The first pattern is a constant c, which may represent a global memory address, a function address, or a value that is not tracked. So the translation relies on the result of memory block generation to either map it to a global memory location or a unique function ID, or treat it as an unnecessary value. When translating $c + \textit{reg}$, we consider two possibilities: (1) c is used as the base address of a memory block and $\textit{reg}$ is used as the offset to the block; (2) $\textit{reg}$ is used as the base address and c as the offset. Therefore, the translation translates $c + \textit{reg}$ to a nondeterministic choice "$\textit{mLoc} \vee *\textit{reg}$", assuming c is mapped to $\textit{mLoc}$ (either a global memory block or a global memory chunk) during global memory partitioning. The translation for constant c is deterministic because one constant address can be the start address of only one memory block/chunk. The actual memory locations associated with the second choice $*\textit{reg}$ will be known during value tracking analysis. Note that $*\textit{reg}$ may yield multiple memory locations. For example, a register may point to either a stack block or a heap block, depending on which path the program is taking. For the third pattern, $c + \textit{reg} * \textit{sc}$, it shares the same translation rule as pattern c, since the part of $\textit{reg} * \textit{sc}$ in the pattern is assumed to stay within the memory block associated with the constant c, according to the block memory model. Thus, it recursively applies $\mathrm{TransAddr(c)}$. The same reasoning applies to pattern $c + \textit{reg} + \textit{reg}' * \textit{sc}$; it applies $\mathrm{TransAddr}(\mathrm{c} + \textit{reg})$.

Table 5.1: MBA-IR transformation rules.

Transformation rules for RTL address expressions:
$\text{TransAddr(c)} :=$
if c is defined in gblk then gblk(c)
elif c is defined in func then func(c)
else None
$\text{TransAddr}(\text{c} + \textit{reg}) :=$
if $\text{TransAddr(c)} = \text{None}$ then $*\textit{reg}$
else $\text{TransAddr(c)} \ \vee \ *\textit{reg}$
$\text{TransAddr}(\text{c} + \textit{reg} * \text{sc}) := \text{TransAddr(c)}$
$\text{TransAddr}(\text{c} + \textit{reg} + \textit{reg}' * \text{sc}) := \text{TransAddr}(\text{c} + \textit{reg})$
Transformation rules for RTL instructions:
$\text{TransIns}(\textit{reg} = \text{e}) :=$
if $\text{TransExp(e)} = \text{None}$ then SKIP
else $\textit{reg} \leftarrow \text{TransExp(e)}$
$\text{TransIns(Mem[a]} = \text{e)} :=$
if $\text{TransExp(e)} = \text{None}$ then SKIP
else $\text{TransAddr(a)} \leftarrow \text{TransExp(e)}$
$\text{TransIns(IF e DO ins)} := \text{TransIns(ins)}$
Transformation rules for RTL expressions:
$\text{TransExp(load_loc(}\textit{reg}\text{))} := \textit{reg}$
$\text{TransExp(load_mem(a))} := \text{TransAddr(a)}$
$\text{TransExp(arith(_, e}_1\text{, e}_2\text{))} :=$
if $\text{TransExp(e}_1) = \text{None}$ then $\text{TransExp(e}_2)$
elif $\text{TransExp(e}_2) = \text{None}$ then $\text{TransExp(e}_1)$
else $\text{TransExp(e}_1) \ \vee \ \text{TransExp(e}_2)$
$\text{TransExp(bitvec(_, c))} :=$
if $\text{TransAddr(c)} = \text{None}$ then None
else $\&\text{TransAddr(c)}$

With TransAddr(a) defined, the rest of translation is straightforward. We next discuss $\text{TransIns}(-)$. The location-update instruction $\textit{reg} = e$ is translated to $\textit{reg} \leftarrow \text{TransExp(e)}$. During expression translation, it generates a "None" value for a constant that is not tracked. Thus, if $\text{TransExp(e)} = \text{None}$, it converts the instruction into a SKIP instruction. Finally, a memory-update instruction $\text{Mem}[a] = e$ is translated to $\text{TransAddr(a)} \leftarrow \text{TransExp(e)}$. As an optimization, during expression translation, values that are not tracked are removed; an RTL instruction that uses only non-tracked values is considered unnecessary and removed from the CFG. At last, for an if-do instruction, IF e DO ins, the translation ignores the condition e and recursively applies

$\mathrm{TransIns}(-)$ on instruction ins.

$\mathrm{TransExp(e)}$ translates RTL expression e into an MBA-IR expression. A location-load expression is directly converted to the corresponding register in MBA-IR. For a memory-load expression, $\mathrm{load_mem}(a)$, it applies the address expression translation ($\mathrm{TransAddr(a)}$) to map it to a memory location or a dereference operation; an RTL bit-vector is treated in the same way as an address expression. For arithmetic expressions, $\mathrm{arith}(bvop,\ exp,\ exp)$, it ignores the operator and conservatively and recursively translates each operands into values we track and use the nondeterministic choose expression to connect them. Such a translation is valid because we only track base addresses for functions and memory blocks. Either operand could be a base address; and the operator is ignored to be conservative.

5.3.3 SSA Transformation

In this step, BPA performs SSA (Single Static Assignment) transformation on the program so that every variable is defined only once. It is easier to analyze programs in the SSA form. For example, performing a flow-insensitive analysis on the program after SSA is equivalent to a flow-sensitive analysis on the program before SSA. Since every variable is defined only once, SSA transformation essentially renames all variables in a program by adding indices to variable names based on the CFG. As a result, the control flow information is recorded by the indexing system. During SSA transformation, ϕ-instructions are introduced into the program to represent the merge points of multiple control flows. For example, in the following simple C code,

if(x == 1) x = x + 1; **else** x = x + 2;

the variable x after the execution of the if-else statement may hold either the value of x in the then branch or the value of x in the else branch. In SSA transformation, an index is introduced for every variable and a ϕ-instruction is introduced after the statement. In detail, the same program in SSA form is as follows:

if(x_0 == 1) $x_1 = x_0 + 1$; **else** $x_2 = x_0 + 2$;
$x_3 = \phi\ (x_1, x_2)$;

The ϕ-instruction selects either x_1 or x_2, depending on whether the control flow reaches the instruction through the then branch or the else branch.

BPA performs SSA on the MBA-IR, where it treats only the registers as variables. We design our SSA transformation in Datalog by adapting an efficient SSA algorithm [69]. Here, for brevity, we do not elaborate on the theoretical details of the original algorithm but only summarize our adaption. The key step of our approach is to first perform the SSA transformation for each function according to the intraprocedural edges in the input CFG. Then, we utilize the interprocedural control-flow information to connect intraprocedural SSA forms into a complete SSA form for the whole program. When constructing ϕ-instructions, BPA applies only the first phase (called the really crude phase) to incrementally add them instead of starting from scratch when new edges are detected. This kind of incremental analysis is significantly more scalable [70, 71]. As a result, our SSA transformation achieves better scalability compared with the original algorithm [69] while maintaining the correctness.

5.3.4 Value Tracking Analysis

After transforming MBA-IR code into its SSA form, BPA performs an interprocedural value tracking analysis. The analysis is implemented by Datalog logic rules on instructions in the MBA-IR SSA form. A Datalog engine then scans through all instructions to perform the value tracking analysis according to the rules. To ease the explanation of our full value tracking rules, we first show the formalization of a basic value tracking analysis that does not consider global memory chunks as memory locations. After that, we will show how to incorporate global memory chunks and an instruction reachability detection module, which improves analysis precision.

Basic value tracking analysis To formalize the basic value tracking analysis, we design rules for the five types of MBA-IR instructions. In the rules, we use the notation $ireg$ to represent an indexed register (i.e., a register with an index) after the SSA transformation. For readability we present Datalog rules in the inference rule notation, which puts the conclusion below a horizontal bar and the assumptions above the bar.

The rules for the register-update instructions and nondeterministic memory-update instructions are presented in Fig. 5.3. Further, for brevity, we do not list the formal

$\text{VSet}(\mathit{ireg}) \coloneqq \{v \mid \text{AlocVal}(\mathit{ireg}, v)\}$
$\text{VSet}(\mathit{mLoc}) \coloneqq \{v \mid \text{AlocVal}(\mathit{mloc}, v)\}$
$\text{VSet}(*\mathit{ireg}) \coloneqq$
$\quad \{v \mid \text{AlocVal}(\mathit{ireg}, \mathit{mLoc}) \wedge \text{AlocVal}(\mathit{mloc}, v)\}$
$\text{VSet}(\&\mathit{mLoc}) \coloneqq \{\&\mathit{mLoc}\}$
$\text{VSet}(\&\mathit{func}) \coloneqq \{\&\mathit{func}\}$
$\text{VSet}(\mathit{Exp}_1 \vee \mathit{Exp}_2) \coloneqq \text{VSet}(\mathit{Exp}_1) \cup \text{VSet}(\mathit{Exp}_2)$

Table 5.2: Definition of $\text{VSet}(-)$.

$$\textsc{AddrMloc}\ \frac{\text{Reachable}(\mathit{ireg} \leftarrow \&\mathit{mLoc})}{\text{AlocVal}(\mathit{ireg}, \&\mathit{mLoc})}$$

$$\textsc{AddrFunc}\ \frac{\text{Reachable}(\mathit{ireg} \leftarrow \&\mathit{func})}{\text{AlocVal}(\mathit{ireg}, \&\mathit{func})}$$

$$\textsc{Ireg}\ \frac{\text{Reachable}(\mathit{ireg} \leftarrow \mathit{ireg}') \quad \text{AlocVal}(\mathit{ireg}', \mathit{val})}{\text{AlocVal}(\mathit{ireg}, \mathit{val})}$$

$$\textsc{Mloc}\ \frac{\text{Reachable}(\mathit{ireg} \leftarrow \mathit{mLoc}) \quad \text{AlocVal}(\mathit{mLoc}, \mathit{val})}{\text{AlocVal}(\mathit{ireg}, \mathit{val})}$$

$$\textsc{DIreg}\ \frac{\text{Reachable}(\mathit{ireg} \leftarrow *\mathit{ireg}') \quad \text{AlocVal}(\mathit{ireg}', \&\mathit{mLoc}) \quad \text{AlocVal}(\mathit{mLoc}, \mathit{val})}{\text{AlocVal}(\mathit{ireg}, \mathit{val})}$$

$$\textsc{AltMloc}\ \frac{\text{Reachable}(\mathit{ireg} \leftarrow \mathit{Exp}_1 \vee \mathit{Exp}_2) \quad \&\mathit{mLoc} \in \text{VSet}(\mathit{Exp}_1) \vee \&\mathit{mLoc} \in \text{VSet}(\mathit{Exp}_2)}{\text{AlocVal}(\mathit{ireg}, \&\mathit{mLoc})}$$

$$\frac{\text{Reachable}(\mathit{ireg} \leftarrow \mathit{Exp}_1 \vee \mathit{Exp}_2) \quad \&\mathit{func} \in \text{VSet}(\mathit{Exp}_1) \vee \&\mathit{func} \in \text{VSet}(\mathit{Exp}_2)}{\text{AlocVal}(\mathit{ireg}, \&\mathit{func})}\ \textsc{AltFunc}$$

$$\frac{\text{Reachable}(\mathit{mLoc}' \vee *\mathit{ireg} \leftarrow \mathit{Exp}) \quad \mathit{val} \in \text{VSet}(\mathit{Exp}) \quad \mathit{mLoc} = \mathit{mLoc}' \vee \text{AlocVal}(\mathit{ireg}, \&\mathit{mLoc})}{\text{AlocVal}(\mathit{mLoc}, \mathit{val})}\ \textsc{ToAlt}$$

$$\frac{\text{Reachable}(*\mathit{ireg} \leftarrow \mathit{Exp}) \quad \mathit{val} \in \text{VSet}(\mathit{Exp}) \quad \text{AlocVal}(\mathit{ireg}, \&\mathit{mLoc})}{\text{AlocVal}(\mathit{mLoc}, \mathit{val})}\ \textsc{UpdMloc}$$

$$\frac{\text{Reachable}(\mathit{ireg} \leftarrow \phi(\mathit{ireg}_1, \ldots, \mathit{ireg}_n)) \quad \bigvee_{i=1,\ldots,n} \text{AlocVal}(\mathit{ireg}_i, \mathit{val})}{\text{AlocVal}(\mathit{ireg}, \mathit{val})}\ \textsc{Phi}$$

Figure 5.3: Representative rules of the basic value tracking analysis.

rules for memory-location-update instructions and indirect memory-update instructions, because they can be adapted from rules for register-update and nondeterministic memory-update instructions, which we will explain; and SKIP is omitted since it has no influence on the value tracking.

We use the predicate $\text{AlocVal}(\mathit{aloc}, \mathit{val})$ to record the value set information for *abstract locations*; an abstract location is defined to be either an indexed register or a memory location. Also, only start addresses of functions and memory blocks/chunks

are tracked. For example, $\mathrm{AlocVal}(\mathrm{EAX}_1, 1000)$ means that register EAX with index 1 holds value 1000. To shorten the rules, we introduce $\mathrm{VSet}(-)$ in Table 5.2 and use the notation $val \in \mathrm{VSet}(Exp)$ to mean that val belongs to the value set of Exp. We use $\mathrm{Reachable}(ins)$ for the predicate that determines the existence of an instruction in the current CFG. Thanks to SSA, BPA's value tracking analysis is flow insensitive. The input program is viewed as a collection of instructions; knowing if an instruction is in the CFG is sufficient for the analysis.

Rules ADDRMLOC and ADDRFUNC are the base cases and initialize all abstract locations with the values that are tracked. Rules IREG and MLOC capture the value flow to an indexed register from another indexed register or a memory location by making the destination indexed register have the same set of values as the source abstract location. Rule DIREG is about the dereference operation via an indexed register. The semantics of the dereference is to first retrieve the memory blocks stored in the indexed register and then transfer the values of the retrieved memory blocks into the destination indexed register. Therefore, in the rule, $\mathrm{AlocVal}(ireg', \&mLoc)$ tells that the $ireg'$ stores a memory location $\&mLoc$; $\mathrm{AlocVal}(mLoc, val)$ states that the memory location $mLoc$ holds value val. As a result, the destination $ireg$ should also hold val. Rules ALTMLOC and ALTFUNC are for a register-update instruction with a nondeterministic choose expression. Essentially, if one of the operands holds a memory location $mLoc$ or a function address $func$, as a result, the destination indexed register also holds the same memory location or function address. Rule TOALT is for a nondeterministic memory-update instruction. The rule finds a possible destination and makes it to hold the value val, which is evaluated from the source expression. Rule PHI deals with ϕ-instructions introduced by SSA transformation. The idea is similar to rule ALTMLOC; if one of the n operands of the ϕ-instruction holds value val, the destination $ireg$ should also hold val. The notation $\bigvee_{i=1,\ldots,n}$ represents a sequence of logical or operations on n predicates. Next, we show how to adapt the rules of register-update instructions to construct rules for the memory-location-update instructions and indirect-mloc-update instructions. In detail, for memory-location-update instructions, we can simply substitute all $ireg$ with $mLoc$ in each rule's conclusion and also in the $\mathrm{Reachable}(ins)$ assumption. For indirect memory-update instructions, we need to add $\mathrm{AlocVal}(ireg', \&mLoc)$ as a new assumption to each rule in Fig. 5.3 and replace $ireg$ in the rule's conclusion with $mLoc$. Note that the interprocedural ϕ-instructions capture the data flows between functions for return values as well as function parameters;

```
0. sub esp, 8
1. lea eax, esp - 4
2. mov [eax], $1000
3. mov edx, [eax]
4. call edx
```
(a) Assembly code

(1) $\mathrm{eax}_1 \leftarrow$ &stk_-12
(2) *(eax_1) $\leftarrow$ $1000
(3) $\mathrm{edx}_1 \leftarrow$ *(eax_1)
(4) call edx_1

(b) Processed program after memory block generation

AlocVal(eax_1, &stk_-12)
AlocVal(stk_-12, \$1000)
AlocVal(edx_1, \$1000)

(c) Value set outputs

Figure 5.4: Toy example for demonstrating BPA's mechanisms.

so our analysis is interprocedural.

Illustration through an example Fig. 5.4 is an example to illustrate BPA's core mechanisms. (a) represents an assembly code of an input binary program; (b) is the processed program after input processing and memory block generation on (a). For example, stack layout analysis determines that a memory address to its corresponding stack memory block is loaded at instruction (1). Also, registers are SSA formed. (c) shows the value-set output by each abstract location, after performing value tracking analysis on (b). Initially, value tracking analysis determines that the register eax_1 holds a memory address to the memory block $\mathrm{stk_}{-12}$. Then, the memory block $\mathrm{stk_}{-12}$ is inferred at memory access at instruction (2). Next, the source value (\$1000) of (2) is stored into this memory block. Then, instruction (3) loads the value from this memory block, and stores it to edx_1. Eventually, indirect call instruction (4) loads the code address from edx_1, and then a new edge will be added from instruction (4)'s basic block to the basic block of code address \$1000. It is not shown in the example, but as a result, instruction (4) adds new ϕ-instructions of registers to the starting basic block of function at \$1000.

Supporting global memory chunks As discussed in Sec. 5.3.1, BPA introduces global memory chunk concept for precision increase. To consider global memory chunks as memory locations, a new assumption needs to be added to every rule in Fig. 5.3. For brevity, we use DIREG as a representative to show how the rules can be adapted. The new rule is shown in Fig. 5.5. First, how a memory-location load is treated is unchanged. That is, it should still load the values of the target memory location, no matter whether it is a memory block or a global memory chunk. Second, rules for memory location

$$\frac{\text{Reachable}(aloc' \leftarrow *ireg) \qquad \text{AlocVal}(ireg, \&mLoc) \\ aloc' \equiv aloc \qquad \text{AlocVal}(mLoc, val)}{\text{AlocVal}(aloc, val)}$$

Figure 5.5: New rule for $*ireg$ to support memory chunks.

updates should now consider more locations to update; since no pointer offsets are tracked, an update to a memory block should also update all the memory chunks inside, to overapproximate the update effect. Furthermore, an update to a memory chunk should also change the memory block that holds the chunk, since the analysis needs to track all possible values in the block. Therefore, we introduce an aloc-equivalence relation to determine if two alocs are equivalent. The formal definition of aloc equivalence is shown as follows:

Definition (Aloc Equivalence)**.** We write $aloc_1 = aloc_2$ if the two alocs are the same. An abstract location $aloc_1$ is covered by another abstract location $aloc_2$, written as $aloc_1 \ @ \ aloc_2$, if $aloc_1$ is a global memory chunk, $aloc_2$ is a global memory block, $aloc_1$ is inside $aloc_2$. Then, two allocs are equivalent, written as $aloc_1 \equiv aloc_2$, iff $aloc_1 = aloc_2 \ \vee \ aloc_1 \ @ \ aloc_2 \ \vee \ aloc_2 \ @ \ aloc_1$

As we argued, every aloc update instruction should update all equivalent alocs of its original target aloc. Therefore, we simply add the aloc-equivalence relation into the rule as a condition; in this way, memory chunk accesses are supported.

Instruction reachability detection An optimization of BPA is to perform analysis only over reachable instructions. Given a CFG of a binary, BPA computes instructions that are reachable in the CFG from the entry point of the program. BPA analyzes only reachable instructions and resolves the targets of only reachable indirect branches. As a result, the target set of an unreachable indirect branch (in dead code) is empty, which increases BPA's analysis precision. To enable this feature, $\text{Reachable}(ins)$ in the rules is replaced with $\text{Reachable}(ins)$, which determines the reachability of an instruction in the CFG.

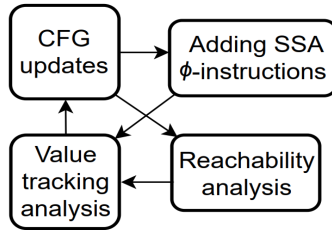

Figure 5.6: Workflow of fixed point recursion for updating a CFG. The components are recursively performed until no more indirect branch targets are found.

5.3.5 Discovering Indirect Branch Targets

BPA relies on the value tracking result to add targets for indirect branches. In this section, we explain how BPA resolves targets for each kind of indirect branch instructions.

Indirect calls The operand of an indirect call in MBA-IR is an aloc. Thus, BPA checks the inferred value set of the aloc operand by its value tracking analysis. All address-taken functions in the value set are treated as targets of the indirect call; BPA uses address taken functions to filter the value set for the targets of indirect calls. One could use other strategies such as arity-matching or type-matching to further improve precision. We will discuss that BPA and arity-matching can be combined to achieve even higher precision of indirect-call targets resolution in Sec. 5.4.

Indirect jumps and return instructions In BPA's input processing component, it performs a heuristic-based indirect-jump target detection to identify the function boundaries and refine the DCFG. Although this detection mechanism resolves jump targets that use jump tables, it does not resolve the targets of indirect tail-call jumps. Since the tail-call optimization is supported by BPA, we keep a record for all indirect calls that are transformed from indirect tail-call jumps during DCFG refinement (Sec. 4.1.2) and use this record to convert such calls back to indirect jumps. These tail-call indirect jumps then take the targets of the corresponding indirect calls in the refined DCFG. For return instructions, BPA constructs a call graph according to the CFG and uses that call graph to resolve the targets of return instructions.

Fixed point recursion Intuitively, the CFG construction, which focuses on resolving indirect branch instructions' targets, and value-tracking analysis should be mutually

recursive: the initial CFG contains only targets of direct branches. Value-tracking analysis then incrementally discovers indirect branch targets and adds those edges to the CFG; the addition of new CFG edges triggers the addition of new $\mathrm{Reachable}(ins)$ facts, which in turn triggers more computation of value-tracking analysis. This process should continue until reaching a fixed point, where a final CFG is generated. To realize the above intuition, after the input DCFG RTL program is converted into MBA-IR, BPA's design makes the rest of the components mutually recursive: (1) SSA transformation depends on the current CFG; (2) value tracking analysis relies on SSA transformation and uses predicates such as $\mathrm{Reachable}(ins)$, which depends on the CFG; (3) after new indirect call targets are discovered, they are added to the CFG, which triggers incremental SSA to add new interprocedural ϕ-instructions or add more operands to existing ϕ-instructions; (4) the new ϕ-instructions enable the value tracking analysis to discover new values in alocs; (5) those new values enable the discovery of new targets of indirect calls. The workflow of this mutually recursive process is shown in Fig. 5.6. The beauty of Datalog is that it allows all these components to be separately specified as logic rules; the Datalog engine then performs a fixed point calculation to compute the final outcome in an iterative fashion, without programmer involvement.

5.4 Evaluation

Our evaluation aims to answer a few major questions: (1) how much does BPA improve over previous techniques in terms of precision? (2) Are the CFGs generated by BPA sound? An unsound CFG (with missing edges) would prevent the execution of legitimate control-flow transfers. (3) how efficient is BPA in CFG generation?

Intuitively, indirect call precision reflects the effectiveness on refining indirect call targets, which is the key to CFI's security guarantee. In general, more precise CFGs (with less spurious indirect-branch edges) provide less freedom for attackers to discover gadgets for exploit generation. Most previous CFI papers rely on graph-based metrics, such as AIR (average indirect-target reduction) and AICT (average indirect-call targets) for evaluations. We follow this tradition to compute AICT for evaluating the precision of BPA's CFGs. In addition, another way of evaluating precision we adopt is to use profiling to collect the actual targets of indirect calls under a set of test cases and treat that as the pseudo ground truth. Based on that, we can approximate precision rates for

indirect calls. Furthermore, the targets of indirect calls collected during profiling are used to validate the soundness of CFGs, by checking whether all targets during profiling are actually included in BPA's CFGs. We note that during our evaluation we focus on indirect calls, since BPA uses similar heuristic-based algorithms as others for resolving indirect jumps' targets and return targets are resolved by call graph construction.

Benchmarks We applied BPA on SPEC2k6's C benchmarks, except for `429.mcf`, `470.lbm`, and `462.libquantum`, which do not have any indirect calls in the compiled binaries. For further evaluation, we also collected five security-critical applications: `thttpd-2.29`, `memcached-1.5.4`, `lighttpd-1.4.48`, `exim-4.89`, and `nginx-1.10`. To prepare the input disassembly, we used RockSalt [1] to disassemble x86 ELF binaries, compiled by GCC-9.2 and Clang-9.0 with optimization levels **-O0**, **-O1**, **-O2**, and **-O3**. We then ran BPA on binaries by different compilers (GCC and Clang) and different optimization levels and collected statistics for all settings. Since similar results and trends are observed from different settings, for brevity, the following sections provide only data for binaries compiled by GCC-9.2.

5.4.1 Recovering Arity Information

As discussed earlier, Address-Taken (AT) functions are considered as possible indirect-call targets in BPA. Indirect call sites are expected to target at only these functions. Besides AT functions, the arity information can be utilized to further refine the set of indirect-call targets.

TypeArmor [29] recovers arity information from a stripped binary to perform arity matching to resolve indirect-call targets. To our best knowledge, this is the state-of-the-art technique that statically detects indirect-call targets for stripped binaries. To compare with BPA and also check whether combining arity matching with BPA would improve precision, we implemented the conservative matching mechanism proposed by TypeArmor [29]. With the recovered arity information, we construct an arity-based CFG for each binary. Further, we refine BPA's output CFG by removing caller-callee pairs that do not satisfy the conservative arity-matching criterion; we call it the *hybrid* strategy.

Principles for Arity We explain the conservative matching mechanism of TypeArmor. First, targets of indirect calls can only be AT functions. Second, an indirect call site that

prepares n arguments is allowed to target functions that expect n or fewer parameters. Third, among all functions that satisfy the first two criteria, an indirect call site that expects a return value should only target the ones that do provide return values. These principles are designed to be conservative to prevent false negatives when identifying indirect call targets.

Implementing arity-based CFGs TypeArmor is designed for x64 binaries, which has a different calling convention from x86. Thus, we designed our own arity information recovery algorithm for x86 stripped binaries to facilitate the conservative arity-matching based CFG construction.

Our design is similar to TypeArmor's approach except for the argument count analysis due to a different calling convention for x86, where call-site arguments and function parameters are passed through the stack. We utilize our stack layout analysis and techniques for identifying arguments and parameters mentioned in Sec. 4.3.2 to infer the number of expected parameters of each AT function. For example, we find the maximum positive stack memory offset that has been used to read from a stack location in each AT function and divide this offset by four to infer the number of parameters.

5.4.2 AICT Comparison

We evaluate different techniques' CFG precision by measuring the average indirect call target (AICT). Table 5.3 shows the AICT statistics of different techniques on the benchmarks, for all optimization levels. To save space, we exclude `458.sjeng` and `433.milc` as all techniques precisely achieve the same results. It also lists the number of x86 instructions and the number of indirect call instructions in the binaries. In the group of AICT columns, AT is a system that allows an indirect call to target all address-taken functions [47]; Arity is the conservative arity-matching based CFG construction, which is our implementation of TypeArmor [29] for x86; BPA is our system; Hybrid resolves indirect-call targets by a combination of BPA and Arity.

As BPA does not consider unreachable indirect calls, some indirect call sites do not have any targets in their results. Thus, some AICT numbers by BPA or the Hybrid approach are less than 1.0. Also, for `456.hmmer` and `memcached`, BPA produces significantly better results than Arity, and the results are at the same level or even better than

Program	Opt Level	Instrs	I-Calls	AICT			
				AT	Arity	BPA	Hybrid
401.bzip2	O0	21K	20	2.0	1.0	2.0	1.0
	O1	11K	20	2.0	1.0	2.0	1.0
	O2	11K	20	2.0	1.0	2.0	1.0
	O3	15K	20	2.0	1.0	2.0	1.0
482.sphinx3	O0	45K	8	6.0	2.3	1.3	1.3
	O1	33K	8	6.0	2.4	1.3	1.3
	O2	35K	7	6.0	1.9	0.7	0.7
	O3	39K	7	6.0	1.9	0.7	0.7
456.hmmer	O0	88K	9	22.0	22.0	2.9	2.9
	O1	58K	11	22.0	22.0	4.3	4.3
	O2	60K	10	22.0	22.0	2.8	2.8
	O3	69K	9	14.0	14.0	1.0	1.0
464.h264ref	O0	161K	369	39.0	30.6	4.3	3.3
	O1	100K	353	39.0	28.7	4.1	3.2
	O2	100K	352	39.0	28.9	26.4	17.3
	O3	164K	355	39.0	28.5	18.0	15.6
445.gobmk	O0	213K	44	1790.0	1395.7	884.6	846.3
	O1	154K	44	1786.0	1392.0	1334.8	1191.5
	O2	157K	44	1788.0	1413.3	1297.2	1198.7
	O3	189K	44	1785.0	1460.3	1376.5	1270.1
400.perlbench	O0	306K	139	721.0	580.4	400.3	328.2
	O1	221K	139	721.0	560.9	364.2	282.2
	O2	226K	110	721.0	536.6	363.7	261.8
	O3	273K	237	718.0	523.7	453.4	322.1
403.gcc	O0	969K	459	1211.0	650.0	534.8	323.5
	O1	652K	473	1207.0	566.6	491.0	250.6
	O2	647K	450	1208.0	581.3	427.8	209.8
	O3	763K	727	1198.0	518.6	544.3	247.7
thttpd	O0	18K	1	17.0	17.0	8.0	8.0
	O1	13K	1	17.0	17.0	8.0	8.0
	O2	13K	1	17.0	17.0	14.0	14.0
	O3	14K	1	17.0	17.0	14.0	14.0
memcached	O0	37K	75	24.0	20.8	1.0	1.0
	O1	25K	75	24.0	21.1	1.3	1.3
	O2	26K	72	25.0	21.6	1.4	1.3
	O3	29K	78	24.0	20.5	0.7	0.7
lighttpd	O0	66K	126	52.0	27.8	35.5	17.1
	O1	46K	126	52.0	26.4	35.5	16.5
	O2	48K	109	52.0	24.7	33.9	14.6
	O3	54K	120	52.0	24.3	34.2	14.6
exim	O0	168K	89	85.0	40.4	31.1	17.3
	O1	135K	89	85.0	41.4	29.6	16.2
	O2	139K	106	85.0	38.0	30.6	17.6
	O3	155K	181	85.0	42.1	40.6	23.0
nginx	O0	232K	409	753.0	441.6	444.0	253.8
	O1	151K	409	753.0	422.2	463.4	251.9
	O2	153K	331	753.0	420.5	525.1	274.6
	O3	164K	365	754.0	432.4	511.0	273.8

Table 5.3: AICT evaluation results (for GCC 9.2).

those produced by a source-type-based approach [8, 46] based on our manual inspection. Through these AICT numbers, we compare precision improvement for four situations: (1) from AT to Arity (2) from AT to BPA, (3) AT to Hybrid, and (4) Arity to Hybrid. We choose to conduct these four comparisons to show BPA's abilities on improving precision over the state-of-the-art techniques. For each comparison, we compute the geometric

mean of a precision improvement rate by each benchmark listed in Table 5.3.

From (1) to (4), the geometric mean of AICT reduction rates are 20.1%, 37.0%, 62.7% and 34.5% respectively, considering all the benchmarks with all optimization levels in Table 5.3, excluding `458.sjeng` and `433.milc` where all techniques precisely resolve the targets. These rates show that BPA achieves higher AICT reduction rate (precision increase rate) than Arity when comparing the 20.1% precision increase rate of AT-to-Arity and the 37.0% precision increase rate of AT-to-BPA. Also, the combination of Arity with BPA can further increase the precision rate substantially.

5.4.3 Profiling-based Precision and Recall

Next we present a method of evaluating CFG construction precision and soundness based on runtime profile data. The idea is to profile a benchmark under a set of test cases and collect the targets of indirect calls. Then intuitively the precision rate is the percentage of CFG-predicted targets that actually appear as targets during profiling, and the recall rate is the percentage of those targets appearing in profiling that are predicted by the CFG. Note that since the set of targets with respect to test cases underapproximates the set of targets with respect to all possible inputs, the profiling-based precision rate is a lower bound of the real precision rate, while the profiling-based recall rate is an upper bound of the real recall rate.

For profiling, we collected a set of runtime traces by using Intel's Pin tool [72] on SPEC binaries. We used the extensive reference input datasets of SPEC2k6 to collect the runtime traces; we did not perform this for those security-critical benchmarks because they did not come with reference input datasets. We then used the following formula to calculate the average precision rate over all indirect calls:

$$Pc = \frac{1}{n}\sum_{i=1}^{n} Pc_i \;\; where \;\; Pc_i = \frac{TP_i}{TP_i + FP_i}$$

TP_i and FP_i are the numbers of true positives and false positives at indirect call site i, respectively. A true positive is a target that is predicted by CFG and also appears during profiling; a false positive is a target that is predicted by CFG but does not appear during profiling.

Table 5.4 shows profiling based precision rates for different techniques. In summary,

Program	Opt Level	Precision (%)			
		AT	Arity	BPA	Hybrid
401.bzip2	O0	30.0	60.0	30.0	60.0
	O1	30.0	60.0	30.0	60.0
	O2	30.0	60.0	30.0	60.0
	O3	30.0	60.0	30.0	60.0
458.sjeng	O0	85.7	85.7	85.7	85.7
	O1	85.7	85.7	85.7	85.7
	O2	85.7	85.7	85.7	85.7
	O3	85.7	85.7	85.7	85.7
433.milc	O0	100.0	100.0	100.0	100.0
	O1	100.0	100.0	100.0	100.0
	O2	100.0	100.0	100.0	100.0
	O3	100.0	100.0	100.0	100.0
482.sphinx3	O0	4.2	66.7	80.0	80.0
	O1	4.2	54.2	80.0	80.0
	O2	2.4	59.5	88.6	88.6
	O3	2.4	59.5	88.6	88.6
456.hmmer	O0	4.5	4.5	90.5	90.5
	O1	4.5	4.5	90.5	90.5
	O2	4.5	4.5	90.5	90.5
	O3	7.1	7.1	100.0	100.0
464.h264ref	O0	0.9	1.7	14.6	14.9
	O1	0.8	1.1	11.9	12.1
	O2	0.8	1.1	3.1	3.4
	O3	0.8	1.1	3.1	3.4
445.gobmk	O0	1.8	2.3	37.3	37.7
	O1	1.8	2.3	22.2	22.7
	O2	1.8	4.3	24.5	24.5
	O3	1.8	2.3	17.6	17.7
400.perlbench	O0	0.4	0.7	29.2	29.5
	O1	0.4	0.8	30.3	30.7
	O2	0.5	0.9	34.7	35.1
	O3	0.3	0.5	24.4	24.6
403.gcc	O0	0.1	0.2	27.6	27.7
	O1	0.1	0.2	30.6	32.2
	O2	0.1	0.2	33.3	34.7
	O3	0.1	0.2	27.5	31.7

Table 5.4: Profiling based precision rates (for GCC 9.2).

AT, Arity, BPA, and Hybrid have the arithmetic mean of precision rates of 25.3%, 35.1%, 54.0% and 57.6% respectively, over all benchmarks and optimization levels. Thus, on average, BPA achieves a 18.9% higher precision rate than Arity.

As noted earlier, profiling based precision rates are underapproximated, as reference datasets may not trigger all program behavior, resulting in incomplete ground truth.

The formula for calculating the average recall rate is:

$$Rc = \frac{1}{n}\sum_{i=1}^{n} Rc_i \;\; where \;\; Rc_i = \frac{TP_i}{TP_i + FN_i}$$

TP_i and FN_i are the numbers of true positives and false negatives at indirect call site i, respectively. A false negative is a target that appears during profiling but is not predicted by CFG. By our experiments, BPA and AT achieve 100% recall rates, and Arity achieves 99.8% on perlbench, 98.9% on gcc, and 100% on all other benchmarks. BPA's 100% recall rate provides a validation that its generated CFGs are sound and enforcing them via CFI does not prevent the legitimate execution of an application.

Table 5.5: Execution time by BPA and VSA. ∞ means timeout (exceeding 10 hours).

	Opt	Execution runtime (s)													
		bzip2	sjeng	milc	sphinx3	hmmer	h264ref	gobmk	perlbench	gcc	thttpd	memcached	lighttpd	exim	nginx
BPA	O0	17	158	35	24	62	350	1221	6919	33658	19	43	81	2554	2656
BPA	O1	8	116	32	31	68	332	1946	3756	23573	13	91	95	2121	2027
BPA	O2	8	131	33	36	79	379	1933	4006	27619	15	113	112	2728	2793
BPA	O3	12	152	34	42	75	1118	2290	4903	25246	17	131	151	2892	2855
VSA [49]	O0–O3	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

5.4.4 Performance Evaluation

We conducted our experiments on Ubuntu 18.04 with 500GB of RAM and 32-cores CPU (Intel Xeon Gold 6136 with 3.00GHz). The machine has a large amount of RAM for the reason that Souffle [73] (v2.0.2), the Datalog engine we use, requires a large memory consumption. We collected memory usage data from the large benchmarks with more than 50K assembly instructions. Table 5.6 shows the results. Not surprisingly, `403.gcc` from SPEC2k6 consumes the largest amount of memory. Except for `403.gcc`, less than 64GB of RAM is sufficient for evaluation of other benchmarks.

Table 5.6: Memory consumption by BPA for O2.

Prog	hmmer	h264ref	gobmk	perlbench	gcc	exim	nginx
Mem (GB)	0.6	3.6	28	57	352	48	24

Points-to analysis is known to be a hard problem, and the scalability issue rises even at the source level [12, 74]. As previously discussed, binary-level points-to analysis is even more challenging to scale due to the limited amount of information and the complexity

```
1. static int (*qcmp)();
2. int hit_comparison(_, _) {...}
3. void FullSortTophits(_ {
4.  specqsort(_, _, _, hit_comparison);
5. }
6. void specqsort(_, _, _, int (*compar)())
7. { qcmp = compar;
8.   if ((*qcmp)(j, lo) > 0) ...
9. }
```

Figure 5.7: Simplified code snippet from `456.hmmer`.

of assembly code. Table 5.5 presents the execution time of BPA for the benchmarks we use. Given that there is currently no practical points-to analysis framework at the binary level for resolving indirect-call targets, we believe the runtime results are acceptable and demonstrate BPA's scalability. Similar to memory usage, `403.gcc` takes the longest time (around 9.3 hours) to finish, due to its large size and its complexity in the usage of control-relevant data.

Execution time comparison with VSA We chose BAP's [49] VSA implementation for the scalability comparison; this decision was motivated by a previous system called BDA [15]. The BDA paper claims that BAP-VSA is the only publicly available VSA framework for complicated benchmarks; other frameworks with VSA such as CodeSurfer [50] and ANGR [14] are either not publicly available or not suitable for complicated benchmarks. For example, ANGR only supports intraprocedural analysis, and therefore not suitable for inferring indirect call targets by points-to analysis, where interprocedural analysis is necessary in most cases. We tested BAP-VSA on our benchmarks; as Table 5.5 shows, none of the benchmarks terminated within 10 hours. The results were consistent with the results reported by the previous paper [15]; when they ran their experiments with BAP-VSA on SPECINT2k (not SPEC2k6), only `181.mcf` terminated in 10.9 hours and others did not terminate within 12 hours. Note that runtime data from the original VSA paper [13] cannot be directly compared with our results, mainly due to the unsoundness in that implementation. The paper lists multiple reasons for possible unsound issues of their analysis, including the failure to resolve indirect call and jump edges that prevent further analysis.

5.4.5 Case Studies

According to previous results, BPA can sometimes generate significantly better results than the arity-based method. To understand in what kinds of scenarios this happens, we did manual inspection on selected benchmarks.

We next discuss a typical case in `456.hmmer`. According to Sec. 5.4.2 and Sec. 5.4.3, BPA generates much higher precision CFG on `456.hmmer` than Arity. It turns out that all the address-taken functions in `456.hmmer` are non-void functions and expect exactly two arguments; as a result, the arity-based method cannot refine the set of targets for an indirect call. In contrast, BPA's block-based points-to analysis achieves high precision. Consider the code snippet adapted from `456.hmmer` in Fig. 5.7. In function `FullSortTophits`, there is a function call to `specqsort` with `hit_comparison`'s address passing to an argument in instruction 4. Then, instruction 7 assigns the argument's function address to a global variable `qcmp`, which is used in instruction 8 to load the function pointer. BPA precisely performs points-to analysis on this code pattern with well partitioned stack and global memory blocks. This is where BPA shows better results.

Chapter 6

BinPointer: Offset Sensitive Pointer Analysis

BPA's goal is to generate precise and sound CFGs by interprocedural points to analysis, but it has not been evaluated for general purpose pointer analysis. Hence we propose BinPointer, a general-purpose interprocedural pointer analysis framework that infers the targets of memory-access instructions at the binary level. It is an extended pointer analysis on BPA; it still adopts the block memory model but its key insight is to track offsets within memory blocks. The offset-sensitive pointer analysis mechanisms improve the granularity of abstract locations during pointer analysis to achieve higher precision than BPA, while being conservative for the pursuit of soundness.

It is challenging to evaluate the precision and soundness of a general-purpose binary-level pointer analysis, since collecting ground truth for large benchmarks is difficult. To mitigate the challenges, we propose dynamic analysis mechanisms to collect and convert accessed memory addresses of the binaries into a format that can be used to evaluate static pointer analysis systems, including but not limited to BinPointer and BPA.

6.1 BinPointer's System Overview

Fig. 6.1 presents the workflow of BinPointer. Its input preprocessing with disassembly and memory block generation steps are similar to BPA except BinPointer applies slightly different block partitioning algorithms to generate more conservative memory blocks.

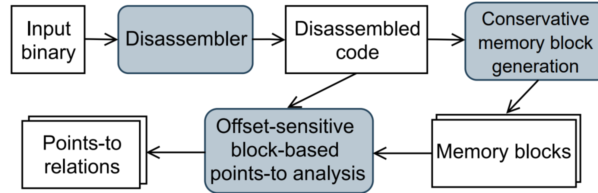

Figure 6.1: BinPointer's system flow.

BinPointer's key contribution is to perform an offset-sensitive pointer analysis, using memory blocks with offsets as abstract locations. This is to achieve higher precision on pointer analysis, while maintaining soundness and reasonable efficiency. Other than this offset tracking mechanism, the offset-sensitive block-based points-to analysis parts of BinPointer are similar to BPA, such as supporting SSA on registers, and the mutual recursion of performing value tracking analysis and CFG updates until fix-point by adding ϕ-instructions of resolved edges of indirect branch. At the end, BinPointer outputs a set of pointer relations, each of which is a tuple (i, b, o) meaning that the memory-access instruction i may access memory block b with an offset o.

6.2 Pointer Analysis

BinPointer's key contribution is to compute offsets within memory blocks to achieve higher precision on pointer analysis, while maintaining soundness. This section discusses how its pointer analysis is performed, assuming a set of memory blocks have already been generated (discussed in Sec. 6.3). The pointer analysis performs value tracking analysis to track values of abstract locations. As noted earlier, this is part of a fixed point computation that also performs SSA and discovers new targets of indirect branch instructions.

6.2.1 Motivating Example

Tracking all offsets would be too costly for pointer analysis as it would maintain a value set for every abstract location $b[o]$, with b being the ID of a block and o being an offset; o can be an integer or $\top$ (indicating any offset within the block). Hence BinPointer adopts a so-called *0-base abstraction*, which precisely tracks only those offsets resulting from pointer arithmetics using $b[0]$ as base addresses but overapproximates other kinds of

```
(1) call malloc                  (4) add edx, 4
(2) mov edx, eax                 (5) mov [edx], 1
(3) mov [edx+4], 0               (6) add edx, 4
```

Figure 6.2: Motivating example for the 0-base abstraction.

offsets to $\top$.

We illustrate this abstraction with an example in Fig. 6.2. In this code, a heap block for allocation site at (1) is created and the address $b[0]$ is stored in eax, assuming b is the allocation-site ID for (1). Then $b[0]$ is stored into edx at (2), followed by a pointer arithmetic $\mathrm{edx} + 4$ at (3); since this pointer arithmetic's base address $b[0]$ has offset 0, its result is tracked precisely and determined to be $b[4]$. Similarly, at instruction (4), the pointer arithmetic is also performed precisely and determined to be $b[4]$. At instruction (5), though edx does not hold a 0-offset address, there is no pointer arithmetic and BinPointer determines it is writing to the abstract location $b[4]$. However, at (6), there is a pointer arithmetic with a non-zero-offset base address; here BinPointer loses precision and edx's value gets to be $b[\top]$.

The motivation for this design choice is to balance between precision and scalability. Many memory operations in programs use 0-offset base addresses to access memory. E.g., imagine a memory block holds a C struct and a field can be accessed through the base address of the struct with some offset; similarly, if a memory block holds an array, an element can be accessed through the base address of the array with some offset. Our 0-base abstraction can reason about these patterns precisely. For each address resulting from a 0-base pointer arithmetic, BinPointer treats it as an abstract location and tracks its value set separately. Furthermore, our abstraction avoids creating too many offsets, especially when a pointer arithmetic occurs in a loop. Imagine a loop is used to access elements in an array with p initialized to be the base address and each iteration uses a pointer arithmetic to move p to the next array element; in this case, BinPointer would not create an abstract location for every array element and would treat p as having offset $\top$ after two iterations.

$$
\begin{array}{rcl}
i, c & \in & \text{Integers} \\
f & \in & \text{Library functions} \\
reg & := & \text{ESP} \mid \text{EBP} \mid \text{EAX} \mid \ \ldots \\
bvop & := & + \mid - \mid \times \mid \ldots \\
ireg & := & reg_i \\
VExp & := & c \mid ireg \mid \text{Mem}[AExp] \\
 & \mid & \text{arith}(VExp,\ bvop,\ VExp) \\
 & \mid & \text{libcall}_{@c}(f, arg_1, \ldots, arg_n) \\
AExp & := & c \mid c + ireg \mid c + ireg * c' \\
 & \mid & c + ireg + ireg' * c' \\
instr & := & ireg = VExp \mid \text{Mem}[AExp] = VExp \\
 & \mid & ireg = \phi(ireg_1, \ldots, ireg_n)
\end{array}
$$

Figure 6.3: Simplified RTL syntax in the SSA form.

6.2.2 Offset-sensitive Value Tracking Analysis

We next formalize the core of BinPointer's pointer analysis. A binary is first disassembled and translated into RTL [1], an IR with a RISC-like instruction set. The semantics of an x86 instruction is captured by a series of RTL instructions. Fig. 6.3 shows a simplified RTL syntax necessary for BinPointer. Also, the syntax is in the SSA form for registers. We write $ireg$ for an indexed register after the SSA transform; it is a register with an integer index. ϕ-functions are also introduced to capture data flows among indexed registers.

$VExp$ is the type of expressions that evaluate to values. One case is $\text{libcall}_{@c}(f, arg_1, \ldots, arg_n)$, which evaluates to the return value of the call to an external library function f (e.g., malloc); further, the call is at address c, which will be used to identify the heap block returned by a call to a malloc-like function. $AExp$ is the type of expressions that represent memory addresses. They include four x86 address modes: (1) c represents a constant memory address, (2) $c + reg$ represents either an address c plus an offset in reg, or an address with reg as the base address and c as the constant offset, (3) $c + reg * c'$ represents an address c plus an offset in $reg * c'$, where c' is a scale, and (4) $c + reg + reg' * c'$ has two possibilities as in case (2): c is a base address and the rest are an offset, or reg is a base address and the rest are an offset.

There are three types of instructions: assignments to an indexed register ($ireg = VExp$), memory writes ($\text{Mem}[AExp] = VExp$), and ϕ instructions ($ireg = \phi(ireg_1, \ldots, ireg_n)$).

$\text{GBMap}(c)$	returns the start address of the global block that global address c belongs to.
$\text{SBMap}(f, o)$	returns the start address of the stack block that contains the offset o in f's stack frame.
$\text{FNMap}(c)$	returns a function name f, if f's start code address is c.
$\text{StkOffMap}(ireg)$	returns (f, o), where f is the function that contains $ireg$ and o the definite offset from $ireg$ to the initial stack top of f, if $ireg$ points to the stack.

Table 6.1: A set of maps assumed by pointer analysis.

BinPointer abstracts away other kinds of instructions (e.g. CPU flag updates) that are not memory related.

Recall that our pointer analysis assumes memory blocks have been generated. We write GB_a for the ID of a global block that starts at global address a; we write $\text{SB}_{(f,a)}$ for the ID of a stack block of function f with the starting offset being a (the offset from the initial stack top of f); and we write HB_c for the heap block with allocation site c. Information about global and stack memory block boundaries are encoded in a pair of auxiliary functions: GBMap and SBMap, explained in Table 6.1. We further assume FNMap, which tells the function a code address belongs to. In addition, our pointer analysis assumes a preprocessing step: a stack layout analysis that analyzes a function and tracks the offset from registers to the stack top. This is a must analysis and produces a StkOffMap, which is explained in Table 6.1. For instance, if esp_0 is the initial stack top and $\text{esp}_1 = \text{esp}_0 - 32$, then $\text{StkOffMap}(\text{esp}_1) = -32$; and if further $\text{eax}_1 = \text{esp}_1 + 4$, then $\text{StkOffMap}(\text{eax}_1) = -28$.

With the help of those maps, Fig. 6.4 and Table 6.2 define the following relations and functions for pointer analysis:

- Relation $\text{AlocVal}(loc, val)$ holds if an abstract location loc is determined to hold a value val.

- Relation $\text{Reachable}(instr)$ holds if an instruction $instr$ is reachable in the CFG being built along with the value tracking analysis, which enables BinPointer to avoid analyzing dead code. Note that BinPointer incrementally updates the CFG

$$\frac{\text{Reachable}(\mathit{ireg} = \mathit{VExp}) \qquad \mathit{val} \in \text{ValSet}(\mathit{VExp})}{\text{AlocVal}(\mathit{ireg}, \mathit{val})} \text{ TOIREG}$$

$$\frac{\begin{array}{c}\text{Reachable}(\text{Mem}[\mathit{AExp}] = \mathit{VExp}) \\ \mathit{val} \in \text{ValSet}(\mathit{VExp}) \qquad \mathit{loc}' \in \text{LocSet}(\mathit{AExp}) \\ \mathit{loc}' \between \mathit{loc}\end{array}}{\text{AlocVal}(\mathit{loc}, \mathit{val})} \text{ TOMEM}$$

$$\frac{\begin{array}{c}\text{Reachable}(\mathit{ireg} = \phi(\mathit{ireg}_1, \ldots, \mathit{ireg}_n)) \\ \text{AlocVal}(\mathit{ireg}_i, \mathit{val}) \text{ for some } i\end{array}}{\text{AlocVal}(\mathit{ireg}, \mathit{val})} \text{ PHI}$$

Figure 6.4: Value tracking rules defined over instructions.

when new indirect branch targets are found.

- Function $\text{LocSet}(\mathit{AExp})$ takes an address expression AExp and returns its set of abstract locations.
- Function $\text{ValSet}(\mathit{VExp})$ takes an expression VExp and returns its value set.

Fig. 6.4 shows a set of flow-insensitive rules that track value sets for different kinds of instructions[1]. Rule TOIREG transfers the value set of VExp to the left-hand side of the assignment. Rule TOMEM transfers the value set of VExp to any abstract location represented by AExp. This rule uses relation $\mathit{loc} \between \mathit{loc}'$ when two abstract locations may overlap in their concrete address sets. E.g., $b[3]$ and $b[\top]$ satisfy the relation since $\top$ covers all offsets, including offset 3. When there is a memory write to an abstract location, for soundness all overlapping abstract locations should contain the written value. Rule PHI transfers the value sets of argument indexed registers to the destination indexed register.

Definition (Location overlapping relation)**.** Two abstract locations $\text{b}[o]$ and $\text{b}'[o']$ are overlapping, written as $\text{b}[o] \between \text{b}'[o']$, if (1) $b = b'$ and (2) $o = o' \ \vee \ o = \top \ \vee \ o' = \top$.

Definition $S \oplus c$ in Table 6.2 models pointer arithmetics based on BinPointer's 0-base abstraction. S is a set of abstract locations and c is a constant. When an abstract location represents the base address of a block (with offset 0), the definition tracks the

[1]Similar to BPA, since SSA is applied to registers, it achieves flow sensitivity on value sets in registers even with flow-insensitive rules.

$S \oplus c \coloneqq$
  $\{b[c] \mid b[0] \in S\} \ \cup \ \{b[\top] \mid b[o] \in S \wedge o \neq 0\}$

$\mathrm{LocSet}(c) \coloneqq \{\mathrm{GB}_a[c-a] \mid a = \mathrm{GBMap}(c)\}$
$\mathrm{LocSet}(c + \mathit{ireg}) \coloneqq$
  if ireg in $\mathrm{StkOffMap}$ then
   let O = $\mathrm{StkOffMap}(\mathit{ireg})$ in
   $\{\mathrm{SB}_{(f,a)}[c+o-a] \mid (f,o) \in O \wedge a = \mathrm{SBMap}(f, c+o)\}$
  else if $c = 0$ then $\{b[o] \mid \mathrm{AlocVal}(\mathit{ireg}, b[o])\}$
  else let S = $\{v \mid \mathrm{AlocVal}(\mathit{ireg}, v)\} \oplus c$ in
   if c in GBMap then $\{\mathrm{GB}_a[\top] \mid a = \mathrm{GBMap}(c)\} \ \cup \ S$
   else S
$\mathrm{LocSet}(c + \mathit{ireg} * c') \coloneqq \{\mathrm{GBMap}(c)[\top]\}$
$\mathrm{LocSet}(c + \mathit{ireg} + \mathit{ireg}' * c') \coloneqq$
  if ireg in $\mathrm{StkOffMap}$ then
   let O = $\mathrm{StkOffMap}(\mathit{ireg})$ in
   $\{\mathrm{SB}_{(f,a)}[\top] \mid (f,o) \in O \wedge a = \mathrm{SBMap}(f, c+o)\}$
  else let S = $\{b[\top] \mid \mathrm{AlocVal}(\mathit{ireg}, b[o])\}$ in
   if c in GBMap then $\{\mathrm{GB}_a[\top] \mid a = \mathrm{GBMap}(c)\} \ \cup \ S$
   else S

$\mathrm{ValSet}(c) \coloneqq$
  if c in FNMap then $\{\mathrm{FNMap}(c)\}$
  else if c in GBMap then $\{\mathrm{GB}_a[c-a] \mid a = \mathrm{GBMap}(c)\}$
  else $\emptyset$
$\mathrm{ValSet}(\mathit{ireg}) \coloneqq$
  if ireg in $\mathrm{StkOffMap}$ then $\emptyset$ else $\{v \mid \mathrm{AlocVal}(\mathit{ireg}, v)\}$
$\mathrm{ValSet}(\mathrm{Mem}[\mathit{AExp}]) \coloneqq$
  $\{v \mid \mathit{loc} \in \mathrm{LocSet}(\mathit{AExp}) \wedge \mathrm{AlocVal}(\mathit{loc}, v)\}$
$\mathrm{ValSet}(\mathrm{arith}(\mathit{VExp}_1,\ \mathit{bvop},\ \mathit{VExp}_2)) \coloneqq$
  match VExp_1, bvop, VExp_2 with
  | c, $+$, c' -> $\{b[o] \mid b[o] \in \mathrm{ValSet}(c + c')\}$
  | c, $-$, c' -> $\{b[o] \mid b[o] \in \mathrm{ValSet}(c - c')\}$
  | c, $+$, _ -> $(\mathrm{ValSet}(\mathit{VExp}_2) \oplus c) \cup \{b[\top] \mid b[o] \in \mathrm{ValSet}(c)\}$
  | _, $+$, c -> $(\mathrm{ValSet}(\mathit{VExp}_1) \oplus c) \cup \{b[\top] \mid b[o] \in \mathrm{ValSet}(c)\}$
  | c, $-$, _ -> $\{b[\top] \mid b[o] \in \mathrm{ValSet}(c)\}$
  | _, $-$, c -> $\mathrm{ValSet}(\mathit{VExp}_1) \oplus -c$
  | _, $+$, _ -> $\{b[\top] \mid b[o] \in \mathrm{ValSet}(\mathit{VExp}_1) \cup \mathrm{ValSet}(\mathit{VExp}_2)\}$
  | _, $-$, _ -> $\{b[\top] \mid b[o] \in \mathrm{ValSet}(\mathit{VExp}_1)\}$
  | _, _, _ -> $\emptyset$
$\mathrm{ValSet}(\mathrm{libcall}_{@c}(f, arg_1, \ldots, arg_n)) \coloneqq$
  if $f \in \{malloc, calloc\}$ then $\{\mathrm{HB}_c[0]\}$
  else if $f = realloc$ then $\{\mathrm{HB}_{c'}[0] \mid \mathrm{AlocVal}(arg_1, \mathrm{HB}_{c'}[0])\}$
  else $\emptyset$

Table 6.2: Definitions of $\mathrm{ValSet}(\mathit{VExp})$ and $\mathrm{LocSet}(\mathit{AExp})$.

new address's offset precisely. In other cases, the resulting address is approximated with offset $\top$.

$\mathrm{LocSet}(-)$ returns a set of memory abstract locations for an $AExp$. We assume that only global data sections can be accessed in a memory operation using a constant base address (with some possible offset), since heap blocks and stack frames are dynamically allocated and cannot be safely accessed via addresses with constant base addresses. Therefore, the case of $\mathrm{LocSet}(c)$ returns a global abstract location using GBMap to compute the global block ID and "$c - \mathrm{GBMap}(c)$" as the offset, assuming c is in the domain of GBMap. The case of $\mathrm{LocSet}(c + \mathit{ireg})$ considers three main cases: (1) if ireg must point to the stack according to $\mathrm{StkOffMap}$, it returns a stack block based on information in SBMap; (2) if c is a base address of a global block, it returns the global block with a $\top$ offset for approximation, and (3) if ireg holds a base address of some block, then pointer arithmetic is performed using c as the offset. The other two cases of $\mathrm{LocSet}(-)$ follow a similar design as the first two cases.

$\mathrm{ValSet}(\mathit{VExp})$ returns a set of values for VExp. For scalability, the pointer analysis tracks only two kinds of values: (1) a memory address in the form of $b[o]$, and (2) the start code address of some function f in the form of $\&f$. For the case of $\mathrm{ValSet}(c)$, it returns $\&f$ if c is the start code address of some function f according to FNMap, and returns a global location if c belongs to GBMap; if neither satisfies, it returns the empty set as the pointer analysis does not track other kinds of values including constants. The definitions of $\mathrm{ValSet}(\mathit{ireg})$ and $\mathrm{ValSet}(\mathrm{Mem}[\mathit{AExp}])$ are self explanatory. For $\mathrm{ValSet}(\mathrm{arith}(-))$, the goal is to track memory addresses that are computed through pointer arithmetics via either the plus or the minus operator. For other operators, the definition returns the empty set as memory addresses should not be computed through those operators. We next discuss the case of $\mathrm{ValSet}(\mathrm{arith}(c,\ +,\ _))$ and other cases are similar. Definition $\mathrm{ValSet}(\mathrm{arith}(c,\ +,\ _))$ considers two possibilities: (1) c used as a base address of a global block, and (2) VExp_2 used as a base address. In case (1), it returns the global block with offset $\top$. For case (2), BinPointer's abstract $\oplus$ operator is performed on the value set of VExp_2 and c. Finally, the case for $\mathrm{libcall}_{@c}(f, arg_1, \ldots, arg_n)$ models how heap blocks are created.

6.3 Implementation

As noted earlier, BinPointer's major components including its pointer analysis are implemented in Datalog, a declarative language that enables a modular implementation and efficient fixed point computation. We use the Datalog engine Souffle [73], and BinPointer's core analysis components are implemented by approximately 3,600 lines of Datalog code.

Another component in BinPointer's implementation is its memory block generation. For heap memory blocks, BinPointer adopts the same allocation-site ID approach as BPA, introduced in Sec. 4.3.4. For partitioning stack and global data sections, BinPointer has different designs for pursuing soundness.

Partitioning global data sections as discussed in Sec. 4.3.3, BPA leverages a data-flow analysis and heuristics to partition global data sections; this method, however, may result in memory blocks that violate the pointer-arithmetic assumption. BinPointer instead relies on symbol tables to identify block boundaries for globals. Symbol tables are generated by compilers to contain information about symbols (e.g., function and variable names) from source code and are embedded in binaries. From symbol tables BinPointer can know global symbols' base addresses, which are used to partition a global data section into memory blocks. Since compiler-generated symbol tables are used by linkers for static/dynamic linking and debuggers for debugging, we can assume the correctness of symbol tables. Therefore, this method of generating memory blocks is trustworthy. Since BinPointer's focus is its offset-sensitive pointer analysis, we rely on symbol tables to ease the preprocessing step for global block boundary generation. To support stripped binaries we can generate global boundaries without relying on symbol tables. One way is to use the heuristic-based methods suggested in Sec. 4.3.3, though it may decrease precision/recall rates depending on the input programs. Generating sound and precise global block boundaries without symbol tables is an interesting research direction.

Stack partitioning BinPointer partitions stack frames into memory blocks in two steps as mentioned in Sec. 4.3.1: (1) gather a set of boundary candidates by a stack layout analysis and (2) remove candidates that may split a compound data structure. BinPointer takes a similar but more conservative approach than BPA in step (2), meaning that less

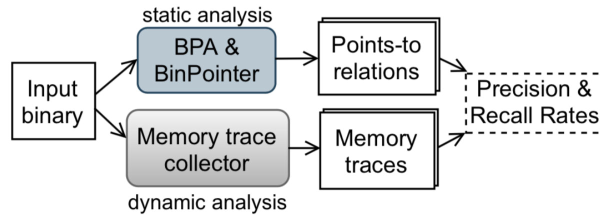

Figure 6.5: Runtime-data driven evaluation strategy.

boundary candidates are kept as the final boundaries. In detail, for final boundaries BinPointer takes $\mathrm{top} + c$, where c is 0 or positive, which is similar to BPA's methods as mentioned in Sec. 5.3.1, but in addition to this BinPointer only considers the address mode that has a scale component is used to determine final boundaries. For example, in BinPointer, the stack address computed from esp-420 in the memory write instruction "`mov [esp+(esi*4)-420], 0`" is a final boundary. In contrast, boundaries considered by BPA may result in an unsound general-purpose pointer analysis on optimized code. For example, BPA considers the stack address computed from esp-416 in the address-loading instruction "`lea eax, esp-416`" as a boundary, which could be taking the address of the second element of an array starting at esp-420.

For a fair comparison with BPA, during evaluation we compare BinPointer with a version of BPA that uses the same memory block generation design as BinPointer's.

6.4 Evaluation Strategy

It is challenging to evaluate the precision and soundness of a binary-level pointer analysis, since collecting ground truth for large benchmarks is difficult. Motivated by previous work [2, 46] that relies on profiling to evaluate static CFG construction, we collect runtime memory accesses triggered by test inputs to validate BinPointer's soundness and to evaluate its precision; the workflow is visualized in Fig. 6.5. To adapt the evaluation to the block memory model, our runtime memory access collection converts the target address of a memory read/write into a pair (b, o), where b is the ID of a memory block, and o is an offset into the block.

We use Intel's Pin [72] to construct a dynamic analysis tool, which collects and converts accessed memory addresses; we call this tool the dynamic collector. In Sec. 6.4.1, we explain how the tool dynamically converts memory addresses into a format that can

be used to evaluate pointer analysis. In Sec. 6.4.2, we introduce a runtime-data based precision metric. Moreover, to ameliorate the incompleteness problem of runtime memory accesses, in Sec. 6.4.3 we propose another metric that relies only on the points-to relations of BinPointer and BPA to compare their degrees of overapproximation.

6.4.1 Dynamic Conversion of Memory Addresses

The key step of our dynamic collector is to convert memory addresses into a format that can be used to evaluate static pointer analysis, including but not limited to BPA and BinPointer. At runtime the information for a memory-access instruction is of the form (i, a), where i is a memory-access instruction and a is a memory address in the flat memory model. However, BinPointer produces results of the form (i, b, o), where i is a memory-access instruction, b the block, and o the offset. Therefore, there is a need to decompose a flat memory address into a block and an offset.

For a memory address in a global data section, the decomposition is straightforward and can be statically performed since a global data section is partitioned via a set of static global memory addresses. In contrast, the decomposition cannot be performed statically for memory addresses to stack frames and dynamically allocated heap regions. For example, if a is an address to a heap region, the base address of the heap region is needed to perform the decomposition; but the base address cannot be known statically and the set of valid base addresses changes during program execution due to memory allocation/deallocation. In general, our Pin-based tool tracks the memory state dynamically (e.g., what heap regions are allocated and their start addresses and sizes) to perform this decomposition.

Converting heap memory accesses Our dynamic collector tracks the set of currently allocated heap memory regions and their memory ranges. In particular, for a call instruction to *malloc*/*calloc*, the collector records the instruction's code address h, the start address of the allocated heap memory region h_{start}, which is the return value of the call, and the size of the allocated heap memory region h_{size}, which can be retrieved from the argument of the call. That is, h is the allocation site and $[h_{start}, h_{start} + h_{size})$ is the allocated memory region range. If a *malloc*/*calloc* at the same allocation site is invoked multiple times in one run, we distinguish them by indices. Thus, a region allocated at

site h is represented by h^n and its range is $[h^n_{start}, h^n_{start} + h^n_{size})$. For a call to *realloc*, which resizes a buffer, the collector retrieves the start address of the resized buffer from the first argument of the call and the new size from the second argument. Then the start address is used to determine the corresponding h^n; and the stored range information of h^n is updated to reflect the new size. Finally, for a call to *free*, its argument holds the start address of a heap memory region that is to be freed. Thus, the collector discards the range information of h^n if h^n_{start} matches the start address.

Information maintained by the collector partitions the heap memory into a set of disjoint heap regions, at any moment during execution. With that information, if a memory instruction i accesses address a, the collector can determine the range $[h^n_{start}, h^n_{start}+h^n_{size})$ that contains a and generates a triple $(i, h, a - h^n_{start})$, where allocation site h is used to identify the static memory block (which represents all regions allocated at that site).

Converting stack memory accesses The collector maintains a stack of function contexts. In detail, for a function call, the collector pushes the callee's name and the base address of the callee's stack frame as a pair (f, f_{base}) onto the stack maintained by the collector; and the collector pops a pair off the stack when it sees a function return. Tail calls are also considered; they are jump instructions that target functions. In effect, a tail call does not add a new stack frame to the call-stack. However, the context changes. Thus, to reflect the context switch, the collector pops the top pair off its stack and pushes the callee's name and the base address of its stack frame onto the stack. Thus, each adjacent pair on the collector's stack defines the range of a stack frame. As a result, when the collector encounters an instruction at code address i accessing a memory address a within a stack frame f, it retrieves the base address f_{base} of the stack frame and converts the address a into an offset into the stack frame $(i, f, a - f_{base})$. If the stack frame of f is partitioned to blocks, $(i, f, a - f_{base})$ is further converted to be of the form (i, b, o), where b is a block in the f's stack frame; this conversion can be performed statically, since the block partitioning of a stack frame does not depend on dynamic information.

6.4.2 Runtime-data based Evaluation Metrics

We use set D for the set of triples (i, b, o) collected during runtime (discussed in Sec. 6.4.1); in these triples, an offset o is a concrete integer. We further use set S

for the set of points-to relations (i, b, o) computed by a static pointer analysis that is based on the block memory model; in these triples, an offset o is an abstract value. E.g., in BinPointer's output, o can be either $\top$ or an integer; in BPA's output, o must be $\top$ as BPA tracks only blocks but not offsets in its pointer analysis.

By comparing D and S, we define notions of recall and soundness. We say a triple $(i, b, o) \in D$ is *covered* by S if either $(i, b, o) \in S$ or $(i, b, \top) \in S$; i.e., the concrete result is covered by the predicted result in S. Then the *recall rate* of S relative to D is the percentage of triples in D that are covered by S. When the recall rate is 100%, we say S is *sound* relative to D, meaning every triple in D is covered by S.

Soundness, however, is only part of the story. For instance, if a pointer analysis's result includes $(i, b, \top)$ for every i and b, it is trivially sound. We therefore introduce a notion of *precision*, relative to runtime data. We first discuss the intuition through examples. Imagine runtime data D contains $(i, b, 0)$ and $(i, b, 1)$. In one case, imagine the pointer analysis result contains $(i, b, 0)$, $(i, b, 1)$, and $(i, b, 2)$; in this case, we say the precision is $2/3$ for (i, b), since two of three predicted outcomes by pointer analysis appear in runtime data. In the previous example, suppose we change the pointer analysis result to be $(i, b, \top)$; since $\top$ represents every possible offset in block b, we say the precision is $2/\mathrm{SizeOf}(b)$, where $\mathrm{SizeOf}(b)$ is the size of block b in terms of bytes. Note that sizes of heap memory blocks can be obtained during runtime data D, while sizes of stack and global memory blocks can be statically determined by the memory block boundaries.

With the precision for a particular pair (i, b) defined, we can define the precision of S relative to D to be the average precision across all blocks and all instructions. We next introduce notations and define it formally. We write $\mathrm{Instrs}(D)$ for the instructions in D, defined as $\{i \mid \exists b \exists o,\ (i, b, o) \in D\}$. We write $\mathrm{Blocks}(D, i)$ for the blocks associated with i in D, defined as $\{b \mid \exists o,\ (i, b, o) \in D\}$. We write $D(i, b) = \{o \mid (i, b, o) \in D\}$ for the set of offsets associated with (i, b) in D; similarly, we write $S(i, b) = \{o \mid (i, b, o) \in S\}$. Then the precision of S relative to D is defined as

$$P(S, D) = \frac{1}{|\mathrm{Instrs}(D)|} \sum_{i \in \mathrm{Instrs}(D)} \frac{\sum_{b \in \mathrm{Blocks}(D,i)} P_{i,b}(S, D)}{|\mathrm{Blocks}(D, i)|},$$

where $P_{i,b}(S, D)$ is the precision for (i, b), defined as:

$$P_{i,b}(S, D) = \begin{cases} \frac{|D(i,b) \cap S(i,b)|}{|S(i,b)|}, & \text{if } (i, b, \top) \notin S. \\ \\ \frac{|D(i,b)|}{\text{SizeOf}(b)}, & \text{if } (i, b, \top) \in S \end{cases}$$

6.4.3 Overapproximation Degree

One weakness of the runtime data based precision metric is that it is relative to runtime data D and the quality of the metric depends on how complete D is. We describe another metric, which does not rely on runtime data and quantitatively measures the overapproximation of a system by calculating the percentage of overapproximated points-to relations in its output. A higher percentage indicates a larger overapproximation; we call this metric *overapproximation degree*. Specifically, if $(i, b, \top) \in S$, we say the pointer analysis output S is overapproximated for the pair of (i, b). A more precise points-to analysis should have less overapproximated (i, b) pairs in its output.

Therefore, we calculate the number of overapproximated (i, b) pairs in the output S to measure the overapproximation degree (OD) of a system, which is defined as:

$$OD(S) = |\{(i, b) \mid (i, b, \top) \in S\}|.$$

6.5 Evaluation

BinPointer is evaluated on a set of SPEC CPU 2k6 C benchmarks and also `nginx` (a web server). The binaries are compiled by GCC-9.2 with optimization levels of -O0 to -O3. To save space, we report only the data of -O0 (unoptimized) and -O2 (most commonly used). We exclude `445.gobmk`, `400.perlbench`, and `403.gcc` as BinPointer does not scale to them. Also, since `nginx` does not come with standard reference inputs, we exclude it from recall and precision evaluation, which requires memory-access traces triggered by reference inputs. Our evaluation aims to answer three major questions: (1) how scalable is BinPointer? (2) how much precision is improved by BinPointer's offset-tracking abstraction? (3) Does BinPointer produce any false negatives?

Program	Details		Runtime-based precision results (%)					
	Opt Level	Instrs	Stack		Global		Heap	
			BPA	BinP	BPA	BinP	BPA	BinP
mcf	O0	3.3K	18.9	100.0	27.5	71.9	n/a	n/a
	O2	2.4K	26.3	100.0	27.0	85.7	n/a	n/a
lbm	O0	6.5K	17.9	100.0	41.7	100.0	n/a	n/a
	O2	2.2K	22.3	99.5	73.1	100.0	n/a	n/a
lib-quantum	O0	10K	65.6	100.0	100.0	100.0	7.0	7.0
	O2	9.6K	47.9	100.0	100.0	100.0	6.9	6.9
bzip2	O0	21K	38.8	97.5	46.6	46.6	1.4	4.5
	O2	11K	16.9	93.2	51.7	51.7	3.6	21.8
sjeng	O0	32K	24.8	94.3	53.7	53.8	n/a	n/a
	O2	22K	32.7	97.5	55.1	55.6	n/a	n/a
milc	O0	31K	40.0	97.7	74.7	91.1	22.4	22.4
	O2	23K	49.4	99.4	81.2	88.9	23.7	23.7
hmmer	O0	88K	30.0	99.9	80.7	80.7	8.0	40.7
	O2	60K	38.0	99.9	76.4	76.4	7.6	11.5
h264ref	O0	161K	28.9	97.8	6.7	69.3	22.7	38.3
	O2	100K	35.3	97.3	6.2	65.5	24.2	40.8

Table 6.3: Precision evaluation results of BPA and BinPointer (-O0 and -O2).

6.5.1 Soundness and Precision

Through dynamic analysis introduced in Sec. 6.4, we collected runtime memory traces in the benchmark programs. When running those programs, we used SPEC CPU 2k6 benchmarks' extensive reference inputs. BinPointer achieves a 100% recall rate, as defined in Sec. 6.4.2. That means BinPointer's pointer-analysis results for our benchmarks are sound relative to runtime memory traces we collected. As discussed in Sec. 6.3, we put additional efforts in making block boundary generation more coarse grained to pursue soundness.

For evaluating precision, Table 6.3 shows the precision results of BPA and BinPointer according to the runtime-trace based metric defined in Sec. 6.4.2. We exclude results for `482.sphinx3`, as the reference inputs by SPEC CPU 2k6 triggered only few memory references. Also, `n/a` in the table means no memory references were triggered for that type of memory blocks. As shown in the table, BinPointer's precision for stack blocks is close to 100%, since BinPointer recovers most stack memory locations through stack layout analysis and precise offset tracking. In contrast, BPA's precision is much lower as it does not distinguish memory accesses with different offsets within the same stack block. The precision difference between BinPointer and BPA for global blocks is relatively smaller. However, there are notable differences for several benchmarks

Memory Region	Overapproximation reduction rate (%)							
	mcf	lbm	libq	bzip2	sjeng	milc	hmmer	h264ref
Stack	79.5	99.1	56.1	30.3	81.1	20.4	33.1	18.7
Global	93.9	89.6	98.8	7.4	10.4	49.4	54.1	15.7
Heap	21.0	50.0	28.7	0.7	1.1	17.9	29.7	12.6

Table 6.4: Overapproximation degree reduction rate of BinPointer over BPA (-O2).

such as `464.h264ref`, showing the effectiveness of BinPointer's offset tracking. For `464.h264ref`, by our manual investigation, the major reason is due to instruction patterns similar to the one below:

```
(1) mov eax, $0x100
(2) mov edx, [eax+$0x8]
```

For this example, BinPointer determines instruction (2) accesses an address with a block starting at 0x100 and offset 0x8. However, BPA does not distinguish offsets in this case; it assumes that eax at (2) points to every offset within the global block of start address 0x100.

Table 6.4 shows the overapproximation degree (OD) reduction results of BinPointer over BPA on binaries compiled by -O2 (other optimization levels' results are generally similar to those of -O2). As an example, for the stack blocks of mcf, BinPointer produces 79.5% less $(i, b, \top)$ triples compared to BPA. In general, BinPointer achieves high OD reduction rates in stack and global blocks. In contrast, BinPointer achieves relatively lower reduction results on heap blocks. For example, the OD rate on `401.bzip2` is 0.7%. Our investigation on `401.bzip2`'s results indicates that BinPointer produces many false positive results when analyzing heap blocks.

6.5.2 Performance Evaluation

BinPointer scales to real world binaries including `nginx` and `464.h264ref` with reasonable computing resources: 128GB of RAM with a 32-core CPU (Intel Xeon Gold 6136 with 3.00GHz) on Ubuntu 18.04. Table 6.5 shows performance results on BPA and BinPointer on the three largest binaries in our benchmark sets. The binaries were compiled at level -O2. Around 6.5 and 2 hours were spent on `464.h264ref` and `nginx` by BinPointer, respectively. Benchmarks not shown in the table terminated within ten to a few hundred seconds by BinPointer. In general BinPointer's runtime is significantly

Program	Runtime (s)		Memory (GB)		# of relations	
	BPA	BinP	BPA	BinP	BPA	BinP
hmmer	54	510	0.6	1.8	7M	12M
h264ref	612	23020	3.3	8.8	52M	109M
nginx	1723	7692	23	41	374M	525M

Table 6.5: Performance evaluations of BPA and BinPointer on large binaries (-O2).

higher than BPA's. We also present the size of $\mathrm{AlocVal}(-,-)$ relations computed by Datalog for both BPA and BinPointer, where M represents millions. In general, BinPointer produces 1.4x $\sim$ 2.0x more relations than BPA, resulting in more memory consumption (about 3x).

The data shows that BinPointer is not as scalable as BPA; the reason is that BPA tracks one value set per memory block, while BinPointer tracks separate value sets possibly at different offsets. On the other hand, BinPointer achieves higher precision, shown in Sec 6.5.1.

6.5.3 Comparison with `angr`'s VSA

We compared BinPointer with the VSA implementation of `angr` [14], a popular open-source binary-analysis framework. To collect `angr`'s VSA output, we constructed the value flow graph (VFG) interface for each benchmark. In the VFG construction, we chose the static binary CFG construction approach as the basis, 3 as the context-sensitivity level, and 4 as the inter-function level. Then each VFG node constructed for user-defined functions was traversed to collect the registers' state (VSA output) in the node. However, we found two issues regarding the VFG results for the benchmarks we considered. First, `angr` produced VSA information for VFG nodes only in the `main` functions. Second, only a few instructions were associated with VSA information in the output; i.e., `angr`'s VSA output was incomplete.

Considering these issues, we turned to micro-benchmarks instead; we used those that were proposed to evaluate SVF [37, 75], an IR-level pointer analysis. We aimed at the `array` directory and the `struct` directory in SVF's GitHub repository[2] and we selected 7 micro-benchmarks that did not crash `angr`'s VSA and that did not have any user-defined functions other than `main`. Then we compared the VSA results of `angr` and BinPointer

[2] https://github.com/SVF-tools/Test-Suite/tree/master/src/non_annotated_tests/

for mov and add instructions that involved memory accesses in the main functions of the 7 micro-benchmarks. Table 6.6 shows the results. For each micro-benchmark, the left-side number is the number of mov and add instructions whose memory references are precisely resolved; and the right-side number is the total number of the mov and add instructions with memory references.

Prog	array1	array2	array4	array5	stride	ben3	ben5
angr-VSA	0/10	0/3	0/4	2/9	1/5	0/3	0/16
BinPointer	10/10	3/3	4/4	9/9	4/5	3/3	16/16

Table 6.6: VSA results by ANGR and BinPointer on SVF micro-benchmarks (-O0).

In all, angr-VSA can precisely generate the VSA information for 3 out of 50 instructions. In fact, the resolved results are straightforward stack-access instructions through rbp. Though angr-VSA did produce result for 2 of the 4 instructions in array4, the output was incorrect. In contrast, BinPointer outputs precise VSA information for 49 out of 50 instructions. The only imprecision is one memory access to a global array within a loop in stride, where BinPointer produces an over-approximated result, while angr-VSA gives no information for this memory access.

6.5.4 Downstream Applications

Resolve indirect-call targets Static binary-level CFG construction has a wide range of applications in security and software engineering, such as control-flow integrity and binary debloating, where resolving the targets of indirect calls is the major challenge. Since BinPointer is superior to BPA in terms of pointer analysis precision, BinPointer should also be more precise in resolving indirect-call targets. BPA measures the precision of a CFG by computing the average indirect call target (AICT); it is the average number of targets across all indirect calls. Lower AICT means higher CFG precision.

We computed AICT metrics of the CFGs constructed by BPA and BinPointer for the benchmarks listed in Table 6.3 and nginx (v1.10). 464.h264ref and nginx are the only benchmarks where there are more than 20 indirect call sites and BinPointer achieves AICT reductions compared to BPA. For other benchmarks, both BPA and BinPointer achieve the same AICT numbers.

Table 6.7 shows the AICT results. As a result, 37.9% (-O0) and 25.7% (-O2) AICT reduction rates are observed on BinPointer over BPA on 464.h264ref. On nginx,

Pointer platform	464.h264ref		Nginx	
	O0	O2	O0	O2
BPA	6.6	30.7	337.8	519.6
BinPointer	4.1	22.8	99.5	465.2

Table 6.7: AICT results for large benchmarks.

```
(1) int *b[100];
 ...
(2) *b[20] = 300;
```

```
<1> mov ebp, esp
...
<2> mov eax, [ebp-736]
<3> mov [eax], 300
```

Figure 6.6: Instructions causing a segmentation fault in `array5`

70.5% (-O0) and 10.5% (-O2) AICT reduction rates are observed on BinPointer over BPA. Moreover, we validated the soundness of the constructed CFGs for `464.h264ref` with runtime indirect call targets; however, we did not validate the soundness of CFGs for `nginx` because `nginx` does not provide reference inputs.

Detect uninitialized memory read We found an uninitialized memory read in the `array5` benchmark in SVF, which causes a segmentation fault. Fig. 6.6 shows `array5`'s simplified source code (left side) and assembly instructions (right side). At runtime, there is no instruction that writes any value to the memory location `[ebp-736]` before instruction (2). As a result, an arbitrary value is transferred into `eax` at (2). Thus, instruction (3) accessing memory through `eax` can cause a segmentation fault. This error can be statically detected by BinPointer, because it produces an empty value set (the $\perp$ value in our lattice) for `eax` at instruction (3). In contrast, BPA's underlying pointer analysis cannot detect this error due to its over-approximation on stack memory accesses. This result shows a potential application of BinPointer to detecting uninitialized memory accesses.

Chapter 7

BinType: Type-based approach for Identifying Indirect Call Targets

Both BPA and BinPointer suffer from scalability issues over large binaries (e.g. consuming a large amount of memory), which is a general limitation of pointer analysis. Hence we propose BinType, a new type based binary-level CFG construction framework. It performs equivalence-class based techniques to infer different types of arity-related alocs, which are used to improve the granularity of a type-system for more precise indirect call target detection. Also it identifies the exact parameter numbers of functions to improve the type-based refinement. Similar to BPA and BinPointer, the techniques are formalized by inference rules and implemented in Datalog, which enables us to achieve modularity and conveniently switch between different components of arity-based approach.

7.1 BinType's System Overview

BinType is designed to resolve indirect call targets over binary programs by utilizing two new techniques to infer types and arity of functions and indirect calls. Fig. 7.1 shows

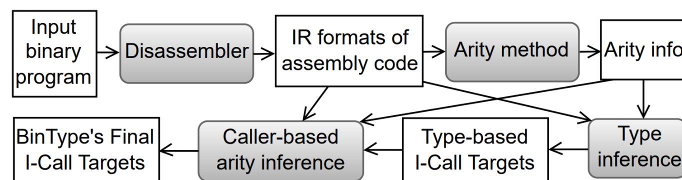

Figure 7.1: System flow.

BinType's workflow. Similar to BPA and BinPointer, BinType's core (type inference) techniques are performed on the Datalog formatted facts of RTL instructions and DCFG information introduced in Sec. 4.1.

BinType improves over TypeArmor [29], which utilizes the arity of functions and indirect calls to resolve indirect call targets; we call this the arity method. As an initial step, BinType implements the arity method for x86 binaries. Then, we perform *equivalence-class based type inference* to infer the types of functions and indirect calls, making matching indirect call targets more fine grained. Last, we apply a caller-based approach to determine the exact number of parameters for applicable functions. We call this technique *caller-based arity inference*. The difference between this arity inference technique and TypeArmor's is that TypeArmor infers arity based on how parameters are used in a function, while our technique first identifies certain caller of the function and uses the caller's supplied number of arguments to determine the function's arity.

Arity inference for x86 binaries TypeArmor's implementation is on x64 binaries. BinType supports only x86 binaries, which follow a different calling convention from that of x64 binaries, and therefore we implement our own arity method, which are required for the later analyses of BinType. We will explain the principles of the arity method, and describe our implementation details in Sec. 7.2.

Equivalence class based type inference Inferring the type of each parameter of the callee function can help increase precision of identifying indirect call targets through the arity-type based signature matching techniques, but type information is unavailable at the binary level. Therefore, we introduce an equivalence-class construction mechanism to infer coarse-grained types of parameters for refining indirect call targets, which are detailed in Sec. 7.3.

Caller-based arity inference At the binary level, the number of a callee function's parameters cannot be statically known without knowing its caller, and therefore under-approximation is applied to calculate the number of parameters; more details are explained in Sec. 7.2.1. Due to this reason, the arity method must over-approximate the indirect call target set to pursue soundness, which decreases the precision. Therefore, we propose new techniques to detect the callee functions' callers to reduce over-

approximation, where the details will be explained in Sec. 7.4.

7.2 Arity Background and Implementations

In this section, we give background information for the principles of TypeArmor's arity-matching technique and our implementations for arity on x86 binaries.

7.2.1 Background: Arity Matching

The arity-matching technique for indirect call target detection introduced by TypeArmor [29] is a binary-level analysis and follows a few principles as mentioned in Sec. 5.4.1. First, the candidates of indirect calls' target functions should be address-taken functions, whose addresses are loaded in the program. Second, indirect calls that supply n number of arguments should target functions that expect n or *fewer* number of parameters. This is mostly because TypeArmor detects the arity of functions based on how parameters are used, which can result in undercounting of arity. For example, if a parameter is unused by a function, TypeArmor's inference would not detect the presence of the parameter, resulting in inferring a smaller arity than the actual one. Furthermore, even if a parameter is used by a function at the source-code level, optimized binary code can remove such uses, leading TypeArmor to infer a smaller arity. Third, indirect calls that expect return values (with a non-void return type) should target only functions that do return values. In contrast, indirect calls that do not expect return values (with the void return type) are allowed to target functions that do or do not return values; this is because a call to a non-void function can elect to ignore the return value.

7.2.2 Counting Arity

As mentioned in Sec. 4.3.2, we can count the number of arguments and parameters on x86 binaries by identifying them through stack layout analysis. Note that this method of counting the arity of a function can under-count, as explained in Sec. 7.2.1.

Variadic functions The number of parameters of variadic functions cannot be statically known. However, under-counting still works over variadic functions and would not violate the soundness of the arity matching technique. Explicit parameters before variadic

parameters are still accessed through stack memory accesses. For example, a variadic callee function `foo(int a, int b, ...)` can take two or more parameters. As long as the first two parameters' memory accesses are represented in the function `foo`, we can determine it has two parameters so that the function can still be considered as the target of indirect calls supplying two or more arguments. Identifying a variadic function is not required for computing the number of parameters but required for our caller-based arity inference's indirect call target detection step in Sec. 7.4.1, which describes how BinType identifies a variadic function.

7.2.3 Identifying Void/Non-void Return Types

To determine whether an indirect call expects a return value or not, we analyze the indirect call's following instructions. We search for an instruction that uses or writes to eax. Upon locating the first such instruction, if eax is used, we conclude the call site expects a return value and the target function should be of a non-void return type. If it is being written or no such instruction is found, we conclude that the callsite expects no return value and the target function should be of a void return type.

For identifying the return type of an address-taken function, our approach is similar to TypeArmor's [29], but we adopt a more conservative approach to pursue soundness. We perform a backward search analysis, starting at the function exit point, which corresponds to the return instruction. During this process, we seek out the return-value assignment site, defined as either: 1) a call instruction, or 2) an instruction utilizing or modifying the 'eax' register according to certain patterns. Upon encountering either of these instructions during the backward search, we stop the process and ascertain the callee's type at these sites. The search is performed sequentially, inspecting each instruction starting from the return instruction until one of the two conditions (1 or 2) is met. For condition 1, when a library call is identified, we determine the callee's type by tracing the library call's return type. Note that for a C standard library call, its return type is manually encoded into the analysis. For direct calls (condition 1), the return type is inferred from the callee's return type of the direct call, allowing for recursive analysis.

In the case of condition 2, if the search finds an instruction that writes to the eax register, we conclude that the function's return type is a non-void type. Otherwise, if eax is loaded from a source operand without accessing memory, we classify the function

as a void function. However, if eax is used to access memory, we do not determine its type here and the search continues until an appropriate stopping point for either condition 1 or 2 is encountered. For example, an instruction `mov [eax],edx` is not considered as condition 1 or 2 but `mov eax,edx` is considered as condition 1. This is a distinction from TypeArmor for pursuing soundness; to our best interpretation of TypeArmor paper [29], eax accessing memory is also considered to satisfy condition 1. According to our experiments, assuming the callee is a void function when eax is used to access memory can break soundness of O2-optimized binaries. Hence, we opt for a more conservative approach.

7.3 Equivalence Class based Type Inference

BinType performs type inference according to a coarse-grained type system to enhance the precision of resolving indirect call targets. This approach involves three steps to generate the final results. 1) abstract location discovery: as a preprocessing step, BinType applies the static single assignment (SSA) form to registers, converting them into indexed registers. Also, it leverages stack layout analysis from Sec. 4.3.1 to infer stack memory locations from memory accesses. These identified memory locations, referred to as alocs, serve as the basis for type determination. 2) type initialization: this component involves analyzing expression patterns of instructions to initialize types for alocs, which refer to register or memory locations. Note that this component can initialize only a small portion of alocs. Our type system consists of two different types, $\mathrm{PtrType}$ and $\mathrm{NonPtrType}$. 3) equivalence class construction: in this step, BinType constructs equivalence relations between alocs. By doing so, it identifies which alocs can share the same type, thus establishing connections between different alocs. Thus, the initialized types can be propagated to more alocs.

7.3.1 Preprocessing

BinType's type inference performs on the Datalog-formatted Intermediate Representation (IR), which is the Register Transfer Language (RTL), as discussed in Sec. 4.1. We further transform the IR to use the Static Single Assignment (SSA) form on registers, ensuring that each register has a unique definition and becomes an indexed register. The

$$
\begin{array}{rcl}
i, c \in \text{Integer} & & \\
f \in \text{CalleeFun} & & \\
r \in \text{Reg} & := & \textbf{ESP} \mid \textbf{EBP} \mid \cdots \mid \text{PC} \mid \text{Flag} \\
\mathit{ireg} \in \text{IReg} & := & r_{(f,i)} \\
\mathit{bvop} \in \text{Bvop} & := & + \mid - \mid \times \mid \text{and} \mid \text{shl} \mid \ldots \\
\mathit{VExp} \in \text{VExp} & := & c \mid \mathit{ireg} \mid \text{Mem}[\mathit{AExp}] \\
& \mid & \text{arith}(\mathit{VExp},\ \mathit{bvop},\ \mathit{VExp}') \\
\mathit{AExp} \in \text{AExp} & := & c \mid \mathit{ireg} + c \mid c + \mathit{ireg} * c' \\
& \mid & c + \mathit{ireg} + \mathit{ireg}' * c' \\
\mathit{instr} \in \text{Instr} & := & \mathit{ireg} = \mathit{VExp} \mid \text{Mem}[\mathit{AExp}] = \mathit{VExp} \\
& \mid & \text{DCall}(f) \mid \text{ICall}(\mathit{ir}) \\
& \mid & \mathit{ireg} = \phi(\mathit{ireg}_1, \ldots, \mathit{ireg}_n); \text{intraprocedural} \\
& \mid & \mathit{ireg} = \Phi_{\mathit{ireg}_1, \ldots, \mathit{ireg}_n}; \text{interprocedural}
\end{array}
$$

Figure 7.2: Simplified RTL syntax with the SSA form.

syntax of RTL with indexed registers is presented in Fig. 7.2. Also, we utilize stack layout analysis' boundary candidates, described in Sec. 4.3.1, to generate stack memory locations to use them during type inference. In detail, all identified boundary candidates from stack memory accesses are used to create four bytes of stack memory locations. For instance, when an instruction is in the form of $r_1 := [r_2 - 16]$ in the function `foo` and if r_2 holds stack offset value $\text{top} - 4$ computed by stack layout analysis, $\text{top} - 20$ will be created as an initial boundary candidate, where BinType infers a stack memory location `foo_-20`. In addition, we consider global memory locations represented by memory access through a single constant memory address (e.g., $AExp$ in Fig. 7.2 represented by c). These indexed registers and memory locations are called alocs, which will be used in type inference in Sec. 7.3.3 and Sec. 7.3.4.

7.3.2 Coarse-grained Types

BinType infers two kinds of types for arguments and return values: 1) a pointer type (PtrType) and 2) a non-pointer type (NonPtrType). PtrType corresponds to the set of values that are used as addresses to access memory locations. NonPtrType contains all other values that are not in PtrType. BinType aims to detect these types for arguments, parameters, and return-value alocs. This results in more information than just the arity and improves the precision of identifying indirect call targets by matching type signatures.

Table 7.1: Definitions of $\mathrm{InferTypeFromExp}(\mathit{VExp})$ and $\mathrm{GetIReg}(\mathit{VExp})$, in OCaml-like syntax.

$\mathrm{InferTypeFromExp}(\mathit{VExp}) :=$ match VExp with | c when $\neg\mathrm{GlobalRgn}(c)$ and $c \neq 0$ –> $\mathrm{NonPtrType}$ | $\mathrm{arith}(\mathit{ireg},\ +,\ c)$ when $\mathrm{StackRgn}(\mathit{ireg})$ or $\mathrm{HeapRgn}(\mathit{ireg})$ –> $\mathrm{PtrType}$ | _ –> None
$\mathrm{GetIReg}(\mathit{VExp}) :=$ match VExp with | $\mathrm{arith}(\mathit{ireg},\ +,\ \mathit{ireg}')$ when $\mathit{ireg} = \mathit{ireg}'$ –> ireg | $\mathrm{arith}(\mathit{ireg},\ +,\ c)$ when $\neg\mathrm{GlobalRgn}(c)$ –> ireg | $\mathrm{arith}(c,\ +,\ \mathrm{arith}(\mathit{ireg},\ +,\ \mathrm{arith}(\mathit{ireg}',\ *,\ c')))$ –> ireg | _ –> None

7.3.3 Type Initialization

Type initialization in BinType is performed independently for each instruction without considering the data flow between instructions, and thus it is performed on a per-instruction basis. The process involves analyzing expressions in each instruction to identify which alocs can be initialized with an appropriate type. Note that only applicable alocs can be initialized with a type; otherwise, no action is taken on those alocs, meaning that they do not get any inferred types at this stage.

Fig. 7.3 shows the inference rules for type initialization. We next define input relations, functions, and output relations used in Fig. 7.3.

Input relations:

- $\mathrm{InstrAt}(\mathrm{I},\ \mathrm{A})$: there is an instruction I defined in Fig. 7.2 at address A.
- $\textsc{KnownMemLoc}(\mathit{AExp},\ \mathrm{memloc})$: a memory address expression AExp represents a memory location memloc. This includes memory locations of arguments and parameters. It can also be a memory location for a global-region address, where AExp is a single global address constant.
- $\mathrm{ArgMemLoc}(\mathrm{CS}, \mathrm{memloc}, \mathrm{Nth})$: the stack memory location memloc is the Nth argument of the callsite CS, which can be either a direct or an indirect call.
- $\mathrm{LibCallArgType}(\mathrm{f}, \mathrm{Nth}, \mathrm{type})$: the Nth argument of a library function f has type. Note that we manually encode the types in this relation for library functions.

$$\frac{\begin{array}{c}(\mathrm{InstrAt}(\mathrm{Mem}[\mathit{AExp}] = _,\ _)\ \texttt{or} \\ \mathrm{InstrAt}(_ = \mathrm{Mem}[\mathit{AExp}],\ _)) \qquad \mathit{AExp} = \mathit{ireg} + \mathrm{c} \qquad \neg\mathrm{GlobalRgn}(\mathrm{c})\end{array}}{\mathrm{InitType}(\mathit{ireg}, \mathrm{PtrType})}\ \textsc{MemExp1}$$

$$\frac{\begin{array}{c}(\mathrm{InstrAt}(\mathrm{Mem}[\mathit{AExp}] = _,\ _)\ \texttt{or} \\ \mathrm{InstrAt}(_ = \mathrm{Mem}[\mathit{AExp}],\ _)) \qquad \mathit{AExp} = \mathrm{c} + \mathit{ireg} * \mathrm{c}' \qquad \mathrm{c}' \neq 1\end{array}}{\mathrm{InitType}(\mathit{ireg}, \mathrm{NonPtrType})}\ \textsc{MemExp2}$$

$$\frac{\begin{array}{c}(\mathrm{InstrAt}(\mathrm{Mem}[\mathit{AExp}] = _,\ _)\ \texttt{or} \\ \mathrm{InstrAt}(_ = \mathrm{Mem}[\mathit{AExp}],\ _)) \qquad \mathit{AExp} = \mathrm{c} + \mathit{ireg} + \mathit{ireg}' * \mathrm{c}' \qquad \mathrm{c}' \neq 1\end{array}}{\begin{array}{c}\mathrm{InitType}(\mathit{ireg}, \mathrm{PtrType}) \\ \mathrm{InitType}(\mathit{ireg}', \mathrm{NonPtrType})\end{array}}\ \textsc{MemExp3}$$

$$\frac{\mathrm{InstrAt}(_ = \mathrm{arith}(\mathit{ireg},\ +,\ \mathit{ireg}'),\ _) \qquad \mathit{ireg} = \mathit{ireg}'}{\mathrm{InitType}(\mathit{ireg}, \mathrm{NonPtrType})}\ \textsc{LdSrc1}$$

$$\frac{\begin{array}{c}\mathrm{InstrAt}(_ = \mathrm{arith}(\mathrm{c},\ +,\ \mathrm{arith}(\mathit{ireg},\ *,\ \mathrm{c}')),\ _) \\ \mathrm{c}' \neq 1\end{array}}{\mathrm{InitType}(\mathit{ireg}, \mathrm{NonPtrType})}\ \textsc{LdSrc2}$$

$$\frac{\begin{array}{c}\mathrm{InstrAt}(_ = \mathrm{arith}(\mathrm{c},\ +,\ \mathrm{arith}(\mathit{ireg},\ +,\ \mathit{VExp})),\ _) \\ \mathit{VExp} = \mathrm{arith}(\mathit{ireg}',\ *,\ \mathrm{c}') \qquad \mathrm{c}' \neq 1\end{array}}{\begin{array}{c}\mathrm{InitType}(\mathit{ireg}, \mathrm{PtrType}) \\ \mathrm{InitType}(\mathit{ireg}', \mathrm{NonPtrType})\end{array}}\ \textsc{LdSrc3}$$

$$\frac{\begin{array}{c}\mathrm{InstrAt}(\mathit{ireg} = \mathit{VExp},\ _) \qquad \mathrm{type} \neq \mathrm{None} \\ \mathrm{type} = \mathrm{InferTypeFromExp}(\mathit{VExp})\end{array}}{\mathrm{InitType}(\mathit{ireg}, \mathrm{type})}\ \textsc{LdValue1}$$

$$\frac{\begin{array}{c}\mathrm{InstrAt}(\mathrm{Mem}[\mathit{AExp}] = \mathit{VExp},\ _) \\ \mathrm{type} \neq \mathrm{None} \\ \mathrm{type} = \mathrm{InferTypeFromExp}(\mathit{VExp}) \qquad \textsc{KnownMemLoc}(\mathit{AExp},\ \mathrm{memloc})\end{array}}{\mathrm{InitType}(\mathrm{memloc}, \mathrm{type})}\ \textsc{LdValue2}$$

$$\frac{\begin{array}{c}\mathrm{InstrAt}(\mathrm{DCall}(\mathrm{CalleeFun}),\ \mathrm{callsite}) \\ \mathrm{ArgMemLoc}(\mathrm{callsite}, \mathrm{memloc}, \mathrm{Nth}) \qquad \mathrm{LibCallArgType}(\mathrm{CalleeFun}, \mathrm{Nth}, \mathrm{type})\end{array}}{\mathrm{InitType}(\mathrm{memloc}, \mathrm{type})}\ \textsc{LibCall}$$

Figure 7.3: Type initialization.

- $\text{GlobalRgn}(\text{c})$: a constant value c is within the address range of global regions (i.e., .BSS, .DATA, .RODATA).

- $\text{StackRgn}(ireg)$: an indexed register $ireg$ has been identified as a pointer to a stack memory address by stack layout analysis.

- $\text{HeapRgn}(ireg)$: an indexed register $ireg$ holds the returned memory address of a heap allocation library call (i.e., malloc, calloc, realloc).

Helper function:

- $\text{InferTypeFromExp}(VExp)$: it takes an expression $VExp$ and returns a type, defined in Table 7.1.

Output relation:

- $\text{InitType}(\text{aloc}, \text{type})$: an abstract location aloc has type, which is either NonPtrType or PtrType.

The first three rules in Fig. 7.3, MEMEXP1 to MEMEXP3, analyze expressions used in a memory operation. MEMEXP1 refers to the scenario where there is a memory read or write with an address expressed through a register added by a constant. If the constant is within the global region, that constant can be a base address and the register be an offset rather than a memory address; in this case, we do not initialize any type since we cannot infer the type by looking at only this instruction. Otherwise, in $ireg + \text{c}$, register $ireg$ must represent some memory address and so its type is PtrType. Note that $ireg + \text{c}$ includes the case when c is 0.

Rules MEMEXP2 and MEMEXP3 handle cases when expressions use an indexed register multiplied by a scale (used in x86 addressing modes "index*scale+disp" and "base+index*scale+disp"). More specifically, when the indexed register is multiplied by a constant, it should not be considered as a memory address, as compilers should not allow memory addresses multiplied by a constant. Consequently, the indexed register is classified to be of type NonPtrType. An exception is when the scale is one, in which case the index register could contain a memory address. Note that our analysis is tailored for binaries compiled by regular compilers.

Rules LDSRC1 to LDSRC3 deal with cases of computing arithmetic expressions. LDSRC1 applies when the source operand expression is an arithmetic operation involving

two identical registers. In such instances, both registers should be constants, implying a type of $\mathrm{NonPtrType}$; compilers should not allow the addition of two identical memory addresses. Rules LDSRC2 and LDSRC3 mirror MEMEXP2 and MEMEXP3, respectively, but they pertain to situations memory operations are not involved.

Rules LDVALUE1 and LDVALUE2 use the function $\mathrm{InferTypeFromExp}(_)$, defined in Table 7.1, to analyze the instruction's source operand expression and return its type. For example, when the function's input source expression (VExp) is a constant, it should represent a non-memory address unless it is a constant value 0 or within the global region, and thus the function returns $\mathrm{NonPtrType}$ for this expression. Note that the constant value 0 at assembly level can represent either an integer value 0 or NULL of a C program, and thus we cannot determine its type at an initialization step. Similarly, when the expression is an arithmetic operation using ireg as an element of $\mathrm{StackRgn}(_)$ or $\mathrm{HeapRgn}(_)$, then it is a memory address, and the function returns $\mathrm{PtrType}$. For all other patterns of expressions, we do not infer any type and return None. When the returned type is not None from the function, we initialize the instruction's destination ireg or memory location in rules LDVALUE1 and LDVALUE2 respectively with the returned type.

Rule LIBCALL enables us to infer types when there is a direct call to a library function. Each argument of the library call has an expectation of either a memory address or a non-memory address. Thus, we can initialize the argument's stack memory location with its respective type, based on the expected type from the library function.

7.3.4 Equivalence Relations Construction

In this subsection we explain how we establish an equivalence relation between alocs, meaning that they share the same type. The inference rules to determine this relation are presented in Fig. 7.4. We first introduce new input relations, a helper function, and an output relation used in Fig. 7.4.

- $\mathrm{CallerSavedReg}(\mathit{ireg})$: an indexed register ireg is a caller-saved register, i.e., EAX, ECX, and EDX, according to the assumed calling convention.

- $\mathrm{ParaMemLoc}(\mathrm{f}, \mathrm{memloc}, \mathrm{Nth})$: a stack memory location memloc is the Nth parameter of function f.

$$\frac{\begin{array}{c}\text{InstrAt}(ireg' = VExp,\ _)\\ ireg = \text{GetIReg}(VExp) \qquad ireg \neq \text{None}\end{array}}{\text{Equiv}(ireg, ireg')}\ \textsc{IRegToIReg}$$

$$\frac{\begin{array}{c}\text{InstrAt}(\text{Mem}[AExp] = VExp,\ _)\\ ireg = \text{GetIReg}(VExp) \qquad ireg \neq \text{None}\\ \textsc{KnownMemLoc}(AExp,\ \text{memloc})\end{array}}{\text{Equiv}(\text{memloc}, ireg)}\ \textsc{IRegToMLoc}$$

$$\frac{\text{InstrAt}(ireg = \text{Mem}[AExp],\ _) \qquad \textsc{KnownMemLoc}(AExp,\ \text{memloc})}{\text{Equiv}(ireg, \text{memloc})}\ \textsc{MLocToIReg}$$

$$\frac{\begin{array}{c}\text{InstrAt}(\text{DCall}(\text{Fun}),\ \text{callsite})\\ \text{ArgMemLoc}(\text{callsite}, \text{memloc}, \text{Nth}) \qquad \text{ParaMemLoc}(\text{Fun}, \text{memloc}', \text{Nth})\end{array}}{\text{Equiv}(\text{memloc}, \text{memloc}')}\ \textsc{DCallArg}$$

$$\frac{\text{InstrAt}(ireg = \phi(ireg_1, \ldots, ireg_n),\ _)}{\text{Equiv}(ireg, ireg_1), \ldots, \text{Equiv}(ireg, ireg_n)}\ \textsc{IntraPhi}$$

$$\frac{\begin{array}{c}\text{InstrAt}(ireg = \Phi_{ireg_1,\ldots,ireg_n},\ _)\\ \text{CallerSavedReg}(ireg)\end{array}}{\text{Equiv}(ireg, ireg_1), \ldots, \text{Equiv}(ireg, ireg_n)}\ \textsc{InterPhi}$$

Figure 7.4: Equivalence class construction rules.

Below is a new helper function:

- $\text{GetIReg}(VExp)$: it takes an expression $VExp$ and returns an indexed register $ireg$ within the expression. This function is defined in Table 7.1.

Then the output relation is:

- $\text{Equiv}(\text{aloc}, \text{aloc}')$: the abstract locations aloc and aloc' are equivalent, meaning that they share the same type.

Fig. 7.4 presents the rules for establishing the equivalence relation (in addition to typical rules of reflexivity, symmetry, and transitivity, not shown in the figure). Note that Rule IREGTOIREG makes the indexed registers of source and destination operand equivalent. This rule uses the function $\text{GetIReg}(_)$ that takes as input the source expression $VExp$ to get the indexed register inside. For instance, when the source operand is an arithmetic operation of $ireg + c$, the function returns $ireg$; an exception is when the

$$\frac{\begin{array}{c}\text{InstrAt}(\text{ICall}(_),\ \text{callsite}) \qquad \text{ArgMemLoc}(\text{callsite}, \text{memloc}, \text{Nth}) \\ \text{Equiv}(\text{memloc}, \text{aloc}) \\ \text{InitType}(\text{aloc}, \text{type})\end{array}}{\text{TypeArgument}(\text{callsite}, \text{Nth}, \text{type})}\ \text{ICALLTYPE}$$

$$\frac{\begin{array}{c}\text{ParaMemLoc}(\text{Fun}, \text{memloc}, \text{Nth}) \\ \text{Equiv}(\text{memloc}, \text{aloc}) \\ \text{InitType}(\text{aloc}, \text{type})\end{array}}{\text{TypeParameter}(\text{Fun}, \text{Nth}, \text{type})}\ \text{ATTYPE}$$

Figure 7.5: Type inference from the equivalence relations

constant c is within a global region, which would make the function return None and the rule is not triggered. Rule IREGTOMLOC is similar to IREGTOIREG, except it deals with situations where writes are performed to known memory locations. In this case, the equivalence relation is established between the indexed register in the source operand and the known memory location at the destination. Rule MLOCTOIREG constructs an equivalence relation between a known memory location in the source operand and the destination indexed register. Note that these three rules hold when the instruction has not been used to generate a type (e.g. rules conflicting with Fig. 7.3). This ensures that unnecessary equivalence relations are not constructed, thereby optimizing the inference process. To simplify the formalization we have not illustrated this feature in the inference rules.

Rule DCALLARG generates equivalence relations between argument memory locations at the callsite of a direct call and parameter memory locations of the callee function. For an intra-procedural ϕ-instruction (generated by SSA for indexed registers), rule INTRAPHI makes equivalent the destination register and the source registers. On the other hand, rule INTERPHI creates equivalence relations from ϕ-instructions between the caller and callee at the entry point of the callee function for caller-saved registers. These registers are used as arguments in optimized code on x86 binaries.

7.3.5 Type Inference

The combination of type initialization in Sec. 7.3.3 and equivalence relation generation in Sec. 7.3.4 enables the type inference of arguments and parameters through the rules in Fig. 7.5. Below are two newly introduced output relations:

$$\frac{\mathrm{ICallArgCount(callsite, count_callsite)} \quad \mathrm{ATFuncParaCount(Fun, count_callee)} \quad \mathrm{count_callee} <= \mathrm{count_callsite}}{\mathrm{ArityTarget(callsite, Fun)}}$$

Figure 7.6: Resolving indirect call targets based on TypeArmor's arity matching technique.

- $\mathrm{TypeArgument(CS, Nth, type)}$: the Nth argument of indirect callsite CS has type.
- $\mathrm{TypeParameter(f, Nth, type)}$: the Nth parameter of address-taken function f has type.

Rules ICALLTYPE and ATTYPE in Fig. 7.5 determine types of indirect calls' arguments and functions' parameters respectively. These results will be used to find indirect call targets next.

7.3.6 Type-based Indirect Call Target Refinement

Based on the inferred types in Sec. 7.3.5, we apply type-signature based techniques to detect indirect call targets. Our type-matching approach is built on top of TypeArmor's arity technique, which is formalized in Fig. 7.6. Our rules for refining indirect call targets through inferred types are presented in Fig. 7.7. We first discuss the new relations introduced in Fig. 7.6 and Fig. 7.7.

- $\mathrm{ATFuncParaCount(f, N)}$: The address taken function f has N number of parameters.
- $\mathrm{DCallArgCount(CS, N)}$: At a direct callsite CS, N number of arguments are supplied.
- $\mathrm{ICallArgCount(CS, N)}$: At an indirect callsite CS, N number of arguments are supplied.
- $\mathrm{ArityTarget(CS, f)}$: indirect callsite CS can target function f. This is the output of TypeArmor's arity method.

$$\frac{\begin{array}{c}\text{TypeParameter}(\text{Fun}, \text{Nth}, \text{type}) \\ \text{TypeArgument}(\text{callsite}, \text{Nth}, \text{type}) \qquad \text{ArityTarget}(\text{callsite}, \text{Fun})\end{array}}{\text{MatchNthArg}(\text{callsite}, \text{Nth}, \text{Fun})} \textsc{ExactMatch}$$

$$\frac{\begin{array}{c}(\neg\text{TypeParameter}(\text{Fun}, \text{Nth}, _)|| \\ \neg\text{TypeArgument}(\text{callsite}, \text{Nth}, _)) \\ \text{ArityTarget}(\text{callsite}, \text{Fun}) \qquad \text{ArgMemLoc}(\text{callsite}, _, \text{Nth})\end{array}}{\text{MatchNthArg}(\text{callsite}, \text{Nth}, \text{Fun})} \textsc{Undetected}$$

$$\frac{\begin{array}{c}\text{ATFuncParaCount}(\text{Fun}, 0) \\ \text{ArityTarget}(\text{callsite}, \text{Fun})\end{array}}{\text{TypeArgTarget}(\text{callsite}, \text{Fun})} \textsc{ZeroParam}$$

$$\frac{\text{MatchNthArg}(\text{callsite}, \text{Nth}, \text{Fun}) \qquad \text{Nth} = 1}{\text{MatchUntilNArgs}(\text{callsite}, \text{Nth}, \text{Fun})} \textsc{Init}$$

$$\frac{\begin{array}{c}\text{MatchNthArg}(\text{callsite}, \text{Nth} + 1, \text{Fun}) \\ \text{MatchUntilNArgs}(\text{callsite}, \text{Nth}, \text{Fun})\end{array}}{\text{MatchUntilNArgs}(\text{callsite}, \text{Nth} + 1, \text{Fun})} \textsc{Iterate}$$

$$\frac{\begin{array}{c}\text{ICallArgCount}(\text{callsite}, \text{N}) \\ \text{MatchUntilNArgs}(\text{callsite}, \text{N}, \text{Fun})\end{array}}{\text{TypeArgTarget}(\text{callsite}, \text{Fun})} \textsc{FindTargets}$$

Figure 7.7: Indirect call targets detection based on BinType's type signature matching.

- $\text{MatchNthArg}(\text{callsite}, \text{Nth}, \text{f})$: the type of the Nth argument of callsite matches the type of the Nth parameter of the function f. This is an intermediate relation for generating $\text{TypeArgTarget}(_, _)$.
- $\text{MatchUntilNArgs}(\text{callsite}, \text{N}, \text{f})$: the first N arguments and parameters all match in terms of types between the callsite and the target function f. This is an intermediate helper relation for generating $\text{TypeArgTarget}(_, _)$.
- $\text{TypeArgTarget}(\text{CS}, \text{f})$: an indirect callsite CS can target function f by our type matching method.

In Fig. 7.7, rule ExactMatch matches the type of a callsite's Nth argument with the type of the corresponding parameter of a target function. This matching process utilizes the relations $\text{TypeParameter}(_, _, _)$ and $\text{TypeArgument}(_, _, _)$ generated from Fig. 7.5. The relation $\text{ArityTarget}(_, _)$ provides a pool of indirect call target

candidates, provided by the arity matching method. The goal is to reduce false positives of the arity method by leveraging our type matching technique.

Rule UNDETECTED follows a conservative approach in cases where the type of either the callsite's Nth order of argument or the callee's Nth order of parameter has not been detected. One of the reasons for undetected types is the limitation of the equivalence relation construction method in BinType due to the lack of memory-level alias analysis. For instance, BinType does not infer memory locations (alocs) of memory accesses to heap regions, resulting in incomplete memory alias analysis. This eventually leads to the absence of the generation of equivalence relations for those corresponding missing alocs. In this case, since exact type matching is not possible, we fall back to the arity-matching method.

Rule ZEROPARAM matches a function with zero parameters to any indirect call; type inference cannot contribute to improve precision over such functions. Next, rules INIT, ITERATE, and FINDTARGETS together iterate through parameters and arguments to perform type matching and find the targets of indirect calls. In detail, rule INIT matches the type of the first argument of a call site and the type of the first parameter of a function; rule ITERATE matches the first N+1 argument types of a callsite and parameter types of a function based on the recursive result of matching the first N argument/parameter types. Then, rule FINDTARGETS detects indirect call targets of a callsite based on the type matching result.

This approach leverages type-based refinement, enhancing the precision of the arity analysis. For instance, consider a callsite with two arguments. If the types of the two arguments cannot be detected, the result has to be overapproximated by allowing all address-taken functions that expect two or fewer parameters. However, if the type of the second argument is detected as $\mathrm{NonPtrType}$, then functions with their second arguments' types as $\mathrm{PtrType}$ are not considered as possible targets.

Matching return types As previously stated in Sec. 7.2.1, a void callsite can target functions with both non-void and void return types whereas a non-void callsite can target only functions with a non-void return type. Our type refinement technique can further refine the matching by considering the memory and non-memory return types. Similar to the argument matching technique introduced in Sec. 7.3.6, return type matching disallows a callsite to target a function whose return type does not match the callsite's

expected return type, assuming both types can be inferred. Once we take the intersection of $\text{TypeArgTarget}(_,_)$ and the result of matching return types, we generate the final output of indirect call targets. Note that these indirect call targets are not used to generate more equivalence relations by constructing more call edges, as false positives in the resolved targets could lead to over-approximation in the generation of equivalence relations.

7.4 Caller-based Arity Inference

BinType's second major contribution is a new technique for determining the exact arity of some functions. TypeArmor [29]'s method of determining the arity of a function is approximate since it is based on uses of parameters in the function. However, when some parameters are unused in a function, they infer a smaller arity than the actual one. Our idea is to formulate a *must analysis* that determines *definite callers* of a function and, if there is such a definite caller, we can use the caller's supplied number of arguments to infer the exact arity of the function. One easy case is when a function has a caller that invokes the function via a direct call; such a direct call is a definite caller. However, many address-taken functions, the candidates of indirect call targets, are often invoked through only indirect calls in real world programs. To address this issue, BinType infers some indirect calls' definite targets through understanding how code addresses flow into indirect calls. For example, if immediately before the indirect call through a register, the program loads into the register a code address to the beginning of function f, then we can infer that the indirect call *must* call f.

To realize the idea, we detect definite targets of indirect calls by applying algorithms similar to the equivalence class method introduced in Sec. 7.3.4. Then, we determine the exact number of parameters of functions for whom we have identified definite callers. Last, we refine indirect call targets for these callee functions using this information.

7.4.1 Indirect Call Target Detection

To detect the definite targets of indirect calls we employ an algorithm similar to the equivalence-relation based type inference in Sec. 7.3. In detail, we apply the same equivalence-relation construction rules in Fig. 7.4 but also introduce new code address

initialization rules. The combination of these new initialization rules and equivalence-relation construction rules enables us to identify indirect call definite targets. Then, these targets' exact number of parameters can be determined from the number of arguments prepared for their callers. We can then use this result to further refine indirect call targets upon the type-inference based results in Sec. 7.3.6. Below are new input relations for this approach:

- $\mathrm{FunCodeAddr(c, f)}$: the constant c is the beginning address of the function f.
- $\mathrm{GlobalMemBlk(start, end, A)}$: the global address A is within the range of start and end, which are used as boundaries to partition a global memory region and generate memory blocks. Note that this relation is generated by a method for global memory block generation introduced in Sec. 4.3.3.
- $\mathrm{ReadFromDataSec(glb, c)}$: the global address glb holds the code address value c from .data section.
- $\mathrm{VariadicFun(f)}$: the function f is a variadic function.

Below are output relations:

- $\mathrm{InitFunc(aloc, f)}$: an aloc holds the beginning address of the function f.
- $\mathrm{ExactParamCount(f, N)}$: for a callee function f, we have determined a definite caller with N number of arguments. Note that variadic functions are exempt from this relation.

Code address initialization As an initialization step, the inference rules in Fig. 7.8 focus on loading code addresses into alocs. Rule LDADDR loads code address from the source operand's expression, which is a single constant, and store it into a destination indexed register. Rule GLBSING loads a code address from .DATA section's global memory location, and stores it into an indexed register.

Similar to BPA, BinType uses the global memory block generation techniques mentioned in Sec. 4.3.3 to partition a global memory region's array-like data-structures into a set of memory blocks. Rule GLBBLK Fig. 7.8 deals with scenarios when an instruction loads from memory with an expression that may read from multiple global addresses.

Here, we load code addresses from all global memory addresses within the global memory block where `c`, used in the memory access's expression, is within the range of that memory block. For all rules in Fig. 7.8, similar rules can be derived on the instructions where the source expressions are the same but destinations of instructions are accessing memory; in this case, the only changing part is to infer the known memory locations from the address expression that is used to access memory, and store code addresses into the destination memory locations.

Identifying variadic functions When determining the exact arity of functions, we need to exclude variadic functions since they can take a variable number of parameters. To identify a variadic function, we rely on the calling convention for implementing variadic functions: a stack memory address of the caller of a variadic function is passed into the function so that it can access the caller's arguments. Threfore, a signature of a variadic function should have an instruction where 1) a positive stack memory address is written to eax, 2) there should not be any instruction with memory access through a stack memory address that is greater than or equal to that address in 1). For example, if there is an instruction `lea eax,ebp+8` where `ebp` holds a stack memory offset of four (obtained by stack layout analysis), so that `eax` is written by 12, as well as there is no other instructions using stack memory address (offset) of 12 or higher in the function, then we assume this function is a variadic funciton. This method is an over-approximation and may produce false positives, which can impact the precision of the analysis in Sec. 7.4 but should not impact its soundness.

Resolving targets to find the exact number of parameters The relations $\mathrm{InitFunc}(_,_)$ and $\mathrm{Equiv}(_,_)$ generated from Fig. 7.8 and Fig. 7.4 respectively can determine the indirect call's definite target functions. Here, variadic functions are excluded. Then we infer a target function's exact number of parameters by looking at an indirect call's expected number of arguments as shown in rule ICALL in Fig. 7.9. The target function of a direct call is known without extra analysis; so we learn the function's number of parameters through its callsite by rule DCALL in Fig. 7.9.

$$\frac{\textsc{Instr}(\mathit{ireg} = \mathrm{c}) \qquad \mathrm{FunCodeAddr}(\mathrm{c}, \mathrm{Fun})}{\mathrm{InitFunc}(\mathit{ireg}, \mathrm{Fun})} \ \textsc{LdAddr}$$

$$\frac{\begin{array}{c}\textsc{Instr}(\mathit{ireg} = \mathrm{Mem}[\mathrm{GlbAddr}]) \\ \mathrm{ReadFromDataSec}(\mathrm{GlbAddr}, \mathrm{CodeAddr}) \\ \mathrm{FunCodeAddr}(\mathrm{CodeAddr}, \mathrm{Fun})\end{array}}{\mathrm{InitFunc}(\mathit{ireg}, \mathrm{Fun})} \ \textsc{GlbSing}$$

$$\frac{\begin{array}{c}\textsc{Instr}(\mathit{ireg} = \mathrm{Mem}[\mathit{AExp}]) \\ (\mathit{AExp} = \mathit{ireg} + \mathrm{c} || \mathit{AExp} = \mathrm{c} + \mathit{ireg} * \mathrm{c}', \mathrm{c}' \neq 1) \\ \mathrm{GlobalMemBlk}(\mathrm{Lower}, \mathrm{Upper}, \mathrm{c}) \\ \mathrm{GlobalMemBlk}(\mathrm{Lower}, \mathrm{Upper}, \mathrm{GlbAddr}) \\ \mathrm{ReadFromDataSec}(\mathrm{GlbAddr}, \mathrm{CodeAddr}) \\ \mathrm{FunCodeAddr}(\mathrm{CodeAddr}, \mathrm{Fun})\end{array}}{\mathrm{InitFunc}(\mathit{ireg}, \mathrm{Fun})} \ \textsc{GlbBlk}$$

Figure 7.8: Initializing indexed registers with functions.

7.4.2 Indirect Call Target Refinement for BinType

The callee functions whose exact number of parameters have been identified can help reducing false positives of indirect call targets. We utilize the relation $\mathrm{ExactParamCount}(_,_)$ to refine indirect call targets of $\mathrm{ArityTarget}(_,_)$ in Fig. 7.10. Rule EXACT allows an indirect call expecting N number of arguments to have targets of functions with the same number of parameters only when those functions' exact number of parameters are detected; otherwise we follow the same convention of arity matching techniques introduced in Fig. 7.6 for those callee functions whose exact number of parameters are not detected by rule NONEXACT. Then, the inference rules of generating $\mathrm{ArityTarget}(_,_)$ allow the generation of new type inference based indirect call target refinement in Fig. 7.7, which is the final output of BinType.

7.5 Evaluations

Our experiments aim to answer several research questions: (1) how much precision increase is achieved by BinType compared to the traditional arity-based technique (as in TypeArmor) as well as other techniques? (2) In what scenarios does BinType improve precision over other techniques? (3) does BinType produce any false negatives (missing

$$\frac{\text{InstrAt(DCall(Fun), callsite)} \quad \text{DCallArgCount(callsite, N_count)} \quad \neg\text{VariadicFun(Fun)}}{\text{ExactParamCount(Fun, N_count)}} \ \textsc{DCall}$$

$$\frac{\text{InstrAt(ICall}(\textit{ireg}), \text{callsite)} \quad \text{ICallArgCount(callsite, N_count)} \quad \text{Equiv}(\textit{ireg}, \text{aloc}) \quad \text{InitFunc(aloc, Fun)} \quad \neg\text{VariadicFun(Fun)}}{\text{ExactParamCount(Fun, N_count)}} \ \textsc{ICall}$$

Figure 7.9: Caller-based arity inference.

$$\frac{\text{ICallArgCount(callsite, count)} \quad \text{ExactParamCount(Fun, count)}}{\text{ArityTarget(callsite, Fun)}} \ \textsc{Exact}$$

$$\frac{\text{ICallArgCount(callsite, count_callsite)} \quad \text{ATFuncParaCount(Fun, count_callee)} \quad \neg\text{ExactParamCount(Fun, _)} \quad \text{count_callee} <= \text{count_callsite}}{\text{ArityTarget(callsite, Fun)}} \ \textsc{NotExact}$$

Figure 7.10: Modified rules for arity matching with the enforcement of caller-based arity inference

valid indirect call targets)? (4) how efficient is BinType and does it scale to real-world binaries?

We evaluated BinType on a set of SPEC CPU 2006 C benchmarks and a few security-critical benchmarks including `exim-4.89` and `nginx-1.10`. The binaries are compiled by GCC-9.2 and with optimization levels of -O0 to -O3; this is the similar set of benchmarks evaluated by the other state-of-the-art techniques such as BPA and CALLEE [48]. To save space we present only results with optimization levels O0 (unoptimized) and O2 (most commonly used).

7.5.1 AICT Results

As discussed in Sec. 5.4, average indirect call target (AICT), which is the average of identified targets across all indirect calls in the input program, is the standard way of measuring the precision of techniques that resolve indirect call targets. This metric has

Table 7.2: AICT evaluation results of four different systems. The lower AICT number means the higher precision.

Program	Opt Level	Instrs	I-Calls	AICT			
				AT	Arity	TyInfer	BinType
401.bzip2	O0	21K	20	2.0	1.0	1.0	1.0
	O2	11K	20	2.0	1.4	1.4	1.4
458.sjeng	O0	32K	1	7.0	7.0	7.0	7.0
	O2	22K	1	7.0	7.0	7.0	7.0
433.milc	O0	31K	4	2.0	2.0	2.0	2.0
	O2	23K	4	2.0	2.0	2.0	2.0
482.sphinx3	O0	45K	8	6.0	2.3	2.3	2.3
	O2	35K	7	6.0	1.1	1.1	1.1
456.hmmer	O0	88K	9	22.0	22.0	22.0	22.0
	O2	60K	12	22.0	20.3	20.3	20.3
464.h264ref	O0	161K	369	39.0	34.6	34.3	26.8
	O2	100K	352	39.0	34.5	29.6	24.6
445.gobmk	O0	213K	44	1790.0	1394.3	1016.3	1008.5
	O2	157K	44	1788.0	1393.8	1125.7	1119.5
400.perlbench	O0	306K	139	721.0	585.6	584.3	520.5
	O2	226K	136	721.0	531.6	529.8	517.0
403.gcc	O0	969K	459	1211.0	680.3	668.4	579.8
	O2	647K	468	1208.0	666.5	657.3	579.4
exim	O0	168K	89	85.0	47.7	44.9	36.3
	O2	139K	106	85.0	42.8	39.5	33.0
nginx	O0	232K	409	753.0	474.8	435.1	407.7
	O2	153K	362	753.0	430.2	397.4	376.8

been used by the state of the art techniques at both source and binary level [2, 10, 48] for evaluations. When enforcing CFI based on the CFGs, intuitively the fewer the AICT, the less freedom is given to attackers. Hence, CFGs with a lower AICT number and without any missing legitimate call targets are desired for enforcing CFI.

Table 7.2 shows AICT results of four different systems. (1) AT refers to the system that resolves any indirect call's targets as all address-taken functions [47]. (2) Arity is the conservative arity-matching technique introduced by TypeArmor [29], which we have customized for x86 binaries as explained in Sec. 7.2. (3) TyInfer is our type-signature based indirect call target refinement techniques explained in Sec. 7.3. (4) BinType refers to our final prototype that further enforces arity type systems with the caller-based arity inference explained in Sec. 7.4. Note that type inference and caller-based arity inference are orthogonal techniques on top of the arity method but for evaluations we show the data in an incremental order by (3) and (4).

In the table, for each program the column `Instrs` contains the number of assembly instructions after compiling the program with a certain optimization level, and `I-Calls` contains the number of indirect calls. The table shows that on large binaries with more than 100K assembly instructions BinType shows precision improvements over other techniques. As BinType's goal is to achieve high scalability, in the rest of the section we focus on the data of large binaries including `464.h264ref`, `445.gobmk`, `400.perlbench`, `403.gcc`, `exim` and `nginx`. On these binaries, compared to Arity, BinType on average achieves 24.2% higher precision at the O0 level, and 21.1% higher precision at the O2 level. Also, Table 7.3 shows AICT precision increase from Arity to TyInfer and TyInfer to BinType. In general, caller-based arity inference shows higher precision increase than type inference except gobmk. The impacts of BinType's type-enforcement based refinement are detailed in Sec. 7.5.2.

7.5.2 Arity Result Details and Case Studies

In this subsection, we study under what circumstances BinType can enhance precision and analyze the impact of each of our proposed techniques.

Table 7.4 shows the number of address taken functions and indirect calls that expect N number of parameters/arguments. For example, we have identified that `400.perlbench` has 488 address taken functions and 40 indirect calls that expect 0 number of arguments. Also, we show the number of functions identified with the exact number of parameters from Sec. 7.4. For instance, BinType identified 6 functions that expect exactly 0 number of parameters.

BinType shows lower impacts in improving precision on `400.perlbench` compared to other benchmarks. One major reason is due to the fact that 488 address-taken functions expect 0 parameters in `400.perlbench`; they are considered as the targets of all indirect calls, where type inference techniques introduced in Sec. 7.3 are not helpful. In `400.perlbench`, our manual investigations show that there are many functions starting with the name `Perl_pp_*` that do not take any function parameter. In contrast, identifying exact number of parameters, ideally the ones that expect fewer number of parameters, is generally helpful in improving precision for BinType's type-inference techniques. As a result, our caller-based arity inference technique in Sec. 7.4 have more impacts on improving AICT than our type inference technique on large benchmarks with

Table 7.3: Precision increase (%) comparisons. TyInfer and BinType rows show AICT precision increase from Arity to TyInfer and TyInfer to BinType from Table 7.2 respectively.

Technique	Opt Level	Programs					
		h264ref	gobmk	perlbench	gcc	exim	nginx
TyInfer	O0	0.9	37.2	0.2	1.8	6.2	9.1
	O2	16.5	23.8	0.3	1.4	8.4	8.3
BinType	O0	28.0	0.8	12.3	15.3	23.7	6.7
	O2	20.3	0.6	2.5	13.4	20.0	5.5

many functions expecting 0 number of parameters such as `400.perlbench`, as shown in Table 7.2.

The caller-based arity inference technique has higher impacts in improving AICT on `403.gcc`; BinType increases precision by 15.3% and 13.4% on O0 and O2 binaries of `403.gcc` respectively over TyInfer, whereas TyInfer only increases precision by 1.8% and 1.4% over these binaries compared to Arity, as shown in Table 7.3. Our manual investigations reveal that BinType can more precisely identify many indirect call targets in `403.gcc` through the techniques introduced in Sec. 7.4 as the benchmark has many code patterns of using callback functions. Thus, we conclude that BinType in general can improve precision over large and complex binaries that frequently use callback functions.

Another common code pattern where indirect call targets can be more precisely resolved by our system is when transferring code address through global addresses. Consider the simplified assembly code in Fig. 7.11, which is adopted from `nginx`. $\$2000$ is a global address that accesses .BSS global memory region, which stores code addresses of $\$1000$, $\$1040$ and $\$1080$. Then these addresses are loaded from memory at instruction 10 and stored to eax, leading to call these code addresses at instruction 12. Such patterns at `nginx` allow finding more functions of `Detected` in Table 7.4.

Table 7.5 shows, at each position, the numbers of parameters/arguments whose types can be precisely inferred by BinType's type system. We say that a parameter/argument's type can be precisely inferred if only one type is inferred for the parameter/argument. Due to over-approximation in BinType, there are instances where both $\mathrm{NonPtrType}$ and $\mathrm{PtrType}$ are inferred for the same parameter/argument; those are not counted as precisely inferred instances. For example, on `445.gobmk`, BinType precisely identifies the first-parameter types of 72 address-taken functions as the pointer type and the first-parameter types of 1430 address-taken functions as the non-pointer type; also it identifies

Table 7.4: Arity argument count information statistics on large binaries (-O2). This table shows the number of address-taken functions (AT), indirect calls (ICall), and functions resolved by $\mathrm{ExactParamCount}(_,_)$ (Detected) that expect a specific count of arguments (N) where N = 1 to 5, or 6 or higher.

Program	Kind	Argument Counts						
		0	1	2	3	4	5	≥ 6
h264ref	AT	1	4	22	1	6	3	2
	ICall	0	8	31	0	2	305	6
	Detected	0	4	2	0	3	4	2
gobmk	AT	46	75	621	909	131	4	2
	ICall	1	6	5	1	23	2	6
	Detected	1	2	8	2	2	0	2
perlbench	AT	488	120	73	23	11	0	6
	ICall	40	33	33	10	7	1	12
	Detected	6	10	10	7	4	0	3
gcc	AT	80	286	477	250	51	39	16
	ICall	40	130	224	39	25	5	5
	Detected	72	41	133	101	55	18	11
exim	AT	30	18	10	7	4	4	12
	ICall	11	43	16	0	18	5	13
	Detected	9	4	4	0	8	0	2
nginx	AT	43	270	197	227	5	3	8
	ICall	34	141	81	97	7	2	0
	Detected	23	27	34	8	3	0	0

the first-argument types of 8 indirect calls as the pointer type and the first-argument types of 18 indirect calls as the non-pointer type.

These type inference results on `445.gobmk` have a significant impact on reducing AICT. E.g., if we consider the first-argument and first-parameter types, those 8 indirect calls that take the pointer type as the first-argument type cannot be matched with those 1430 functions that take the non-pointer type as the first-parameter type; this eliminates 1403 out of 1788 candidates (the number of address taken functions of `445.gobmk`'s O2 binary). Fig. 7.12 shows simplified source and assembly code of `445.gobmk` that illustrates how this result is achieved. The benchmark has around 1152 functions with the form of the name `autohelperpat<N>`, where <N> is an integer, and has the same code patterns of parameter types. At source level, instruction 5 shows that the first parameter of the function in Fig. 7.12 is an integer. $\mathrm{NonPtrType}$ can be identified at the assembly instruction by BinType as the first parameter's memory location at instruction 11 is equivalent to eax, which conveys the type of $\mathrm{NonPtrType}$ as it is used as a scale register, which fits to the rule MemExp2 in Fig. 7.3. In contrast, the indirect call at

Table 7.5: BinType's type inference statistics on large binaries (-O2). This table shows the number of address-taken functions (AT) whose parameters' types can be precisely inferred and the number of indirect calls (ICall) whose arguments' types can be precisely inferred. The results are in the form of (`l_n / m_n / r_n`), where `l_n` represents the number of instances for which the inferred type is the pointer type, and `m_n` presents the cases for which the inferred type is the non-pointer type, and `r_n` refers to the total number of arguments/parameters at a position.

Program	Kind	Nth parameters and arguments					
		1st	2nd	3rd	4th	5th	$\geq$ 6th
h264ref	AT	26/0/38	19/2/34	6/0/12	6/2/11	0/2/5	0/0/2
	ICall	30/0/352	30/0/344	2/0/313	2/288/313	0/294/311	0/0/6
gobmk	AT	72/1430/1742	5/8/1667	10/1/1046	3/5/137	2/0/6	0/0/2
	ICall	8/18/43	1/8/37	2/7/32	8/5/31	6/0/8	0/6/6
perlbench	AT	120/5/233	37/0/113	0/5/40	3/0/17	0/1/6	6/3/6
	ICall	18/1/99	11/0/66	6/4/30	6/1/20	0/2/13	3/2/12
gcc	AT	348/1/1128	81/19/833	36/1/356	9/0/106	15/0/55	0/0/16
	ICall	108/6/428	18/24/298	4/5/74	2/1/35	0/0/10	1/0/5
exim	AT	26/2/55	11/0/37	13/0/27	1/1/20	8/0/16	13/0/12
	ICall	38/26/95	18/4/52	11/2/36	10/6/36	5/3/18	2/6/13
nginx	AT	465/0/710	251/3/440	145/2/243	0/0/16	5/0/11	3/0/8
	ICall	151/2/328	37/30/187	5/27/106	0/1/9	2/0/2	0/0/0

instruction 28 expects a pointer type on the first argument, as the register eax (which holds the first argument's value) is used to access memory at instruction 25. This triggers the initialization rule MEMEXP2 in Fig. 7.3. Based on our manual analysis on the source code of `445.gobmk`, there are 8 indirect calls with this code pattern, allowing us to reduce AICT greater than the caller-based arity inference technique, as shown in Table 7.3.

Root cause of unresolved types BinType's equivalence class construction rules defined in Fig. 7.4 do not address all possible instruction patterns. For example, BinType does not generate an equivalence relation over an add or sub instruction with two different registers of source operands. The motivation of this design choice is that it can be incorrect to make any equivalence relation assumptions such as the destination register's type is equivalent to two source registers' types. For example, when each of two source registers has different types of PtrType and NonPtrType, the destination register's type is PtrType. On the other hand, when both registers have the same type NonPtrType, the destination register also has NonPtrType. Therefore, it is unsafe to infer any equivalence relations over such instructions. Meanwhile, our manual investigation reveals that not

```
 1. <ngx_http_addition_filter_init>:
 2.  mov    $1000, [$2000]
 3. <ngx_http_userid_init>:
 4.  movl   $1040, [$2000]
 5. ...
 6.
 7. <ngx_http_headers_filter_init>:
 8.  movl   $1080,[$2000]
 9.  ...
10.  mov    [$2000],%eax
11.  ...
12.  call   eax
```

Figure 7.11: Simplified assembly code snippet from nginx.

supporting add instructions during equivalence relation inference can result in many unresolved types over arguments or parameters. Consider the following simplified example from `445.gobmk`:

```
<autohelperpat1152>:
 1. mov ebp, esp
 ...
 2. mov eax, [ebp+12]
 3. add eax, edx
```

In this example, identifying the type of the second parameter at instruction 2 is prevented at the add instruction 3 where we do not generate any equivalence relation. The same code pattern appears in the benchmark `445.gobmk` with more than 1000 functions by the name `autohelperpat<N>` where <N> is an integer. In detail, these functions share the same code pattern as their second parameters are computed in the same way as the above example, where BinType cannot infer any type. Analysis on these functions have large impacts on the total results since these functions comprise a large portion of `445.gobmk`. As a result, BinType eventually achieves the lower number of resolved types on the second parameter in `445.gobmk` compared to the first parameter, as shown in Table 7.5. Overall, BinType's equivalence-relation based type inference algorithm is insufficient to capture such patterns, which would require a type inference algorithm that first builds a flow graph and then infers types based on the graph. We leave the supports for inferring

```
 1. // source code for callee
 2. extern int transform[..][8];
 3. #define TRANSFORM(.., trans, ..)\
 4. (transform[..][trans] + ..)
 5. int autohelperpat1152 (int trans, ..){
 6.   int a = TRANSFORM(.., trans, ..);
 7.
 8. // assembly code for callee
 9. <autohelperpat1152>:
10.   mov   ebp, esp
11.   mov   eax, [ebp+8]
12.   mov   edx, [$1000+(eax*4)]
13.
14. // source code for indirect call
15. void atari_atari_attack_callback
16. (.., struct pattern *pattern, ..) {
17.   int move = TRANSFORM(pattern->o, ..);
18.   ..
19.   if (!pattern->helper(pattern, ..))
20.
21. // assembly code for indirect call
22. <atari_atari_attack_callback>:
23.   mov   ebp, esp
24.   mov   eax, [ebp+12]
25.   mov   edx, [eax+148]
26.   ..
27.   push  eax
28.   call  ecx
```

Figure 7.12: Simplified source and assembly code snippet of 445.gobmk.

types over an add or sub instruction as interesting future work, which can help us resolve more types of arity.

7.5.3 Dynamic Analysis based Soundness Analysis

One important aspect of CFI is that its enforced CFG must be sound, meaning that the CFG does not miss any legitimate targets for an indirect call. To validate the soundness of CFGs produced by BinType, we have collected the pseudo-ground truth of indirect call targets using dynamic analysis, which is the same methodology as computing profiling based recall rates for BPA, discussed in Sec. 5.4.3. In detail we used Intel's Pin [72],

Table 7.6: AICT, runtime, and memory consumption evaluation results of BPA [2] and BinType on large binaries (-O2) with more than 100K instructions.

Program	AICT		Runtime (s)		Memory (GB)	
	BPA	BinType	BPA	BinType	BPA	BinType
h264ref	26.4	24.6	379	39	3.6	0.6
gobmk	1297.2	1119.5	1933	60	28	0.7
perlbench	363.7	517.0	4006	60	57	0.7
gcc	427.8	579.4	27619	325	352	7.3
exim	30.6	33.0	2728	36	48	0.4
nginx	525.1	376.8	2793	32	24	0.4

the dynamic binary instrumentation framework, to collect runtime traces of indirect call targets on SPEC CPU 2k6 C benchmarks using their extensive reference input datasets; we did not collect runtime traces for `nginx` and `exim` as they do not provide reference inputs.

Our experiments showed that BinType did not miss any target in the collected runtime traces; in other words, BinType's resolved indirect call targets are supersets of dynamically collected pseodu-ground truth data. Our additional conservative analysis in identifying non-void return types of callee functions mentioned in Sec. 7.2.3 enables us to avoid causing false negatives, resulting in a 100% recall rate; This shows BinType's capability of supporting the CFI application. However, note that this result is overapproximated, as there may be some legit indirect call targets missed by BinType as well as by Intel's Pin tool due to the lack of input coverage, which is a general limitation of dynamic analysis.

7.5.4 BinType's Efficiency and Comparison with BPA

Table 7.6 shows precision and efficiency comparison results between BinType and BPA over large binaries (with optimization level O2). We do not compare with CALLEE's results due to its nonsupport of CFI [48]. In detail, as the general limitation of deep learning based techniques, CALLEE does not achieve 100% recall rates that is required for CFI; it presents AICT results when the recall rates are 99.9%. In general, BinType shows comparable AICT results to BPA. While BinType achieves higher precision of 7.3%, 15.9% and 39.4% on `464.h264ref`, `445.gobmk` and `nginx` than BPA respectively, BPA achieves higher precision of 42.1%, 35.4% and 7.8% on `400.perlbench`, `403.gcc`

and `exim` than BinType respectively. However, BinType shows much higher efficiency over BPA. BinType achieves a 59.5X faster runtime on average than BPA, and a 59.3X reduction of memory consumption on average over BPA. We tested BinType on Ubuntu 18.04 with 32GB of RAM and 32-core CPU of Intel Xeon Gold 6136 with 3.00GHz. In contrast, BPA used large computing resources with 500GB of RAM and CALLEE used 768GB of RAM and 4 Nvidia GPUs (A100 PCIE) [48]. BinType does not require such large computing resources and it scales to real-world binaries. The evaluation shows that BinType is a practical tool for analyzing and detecting large binaries' indirect call targets.

Chapter 8

Discussions, Future Work and Conclusion

In this chapter, we discuss a few scenarios that can affect the soundness of our research. Next, we propose a few interesting future directions that can extend and improve the research discussed in this dissertation. Last, we summarize our work and conclude this dissertation.

8.1 Discussions

This section describes what aspects can affect the soundness of our pointer analysis frameworks (BPA and BinPointer) and type-based CFG construction approach (BinType).

8.1.1 Memory Blocks for BPA and BinPointer

The memory-block based pointer analysis systems' pointer arithmetic assumption is that a pointer to a memory block cannot be pointing to another memory block via pointer arithmetic. Therefore, memory blocks must be partitioned in a sound way to pursue sound pointer analysis. Our memory block boundary partitioning techniques aim to partition the memory into compound data structures such as arrays or structs, but the soundness bottleneck may remain for the stack and global regions. For example, BPA's global region partitioning techniques described in Sec. 4.3.3 are heuristic-based. These heuristics may not be sound under all circumstances, meaning that the generated blocks may violate the

pointer-arithmetic assumption of the block memory model. To avoid potential soundness violation issues, BinPointer utilizes symbol tables to recover block boundaries for global regions, meaning that it relies on source-level information. Therefore, principled binary-level solutions for generating global memory block boundaries are desired.

8.1.2 Calling Convention Assumptions for BinType

For correctness of identifying indirect call targets, BinType's approach should not under-approximate the number of arguments on indirect calls; in other words, the inferred number of arguments over call instructions should not be fewer than the actual number. Furthermore, BinType's caller-based arity inference requires that the number of arguments of some indirect calls should be correctly inferred when the indirect calls' target functions have been resolved by BinType's must analysis introduced in Sec. 7.4.1. As described in Sec. 4.3.1, we assume the classic **cdecl** x86 calling convention. BinType identifies arguments of call instructions by leveraging stack layout analysis under this calling convention assumption where arguments will be pushed onto stack in a consecutive order. A violation towards this assumption may cause BinType to produce false negatives in resolving indirect call targets. Meanwhile, it is also doable to adapt BinType to other calling conventions. For example, supporting x64 binaries would require BinType to identify general purpose registers (rdi, rsi, rdx, rcx, r8, and r9) as the first six integer or pointer arguments.

8.2 Framework Extensions

Our current infrastructure only supports C binaries compiled with the x86-32 bit architecture. Extension of the scope to support different kinds of binaries will make our system more applicable. We believe our techniques are implementable to support C++ and binaries of different architectures with engineering efforts.

8.2.1 Binaries of Different Architectures

BPA, BinPointer, and BinType utilize RockSalt [1] to convert binary code into an RTL IR. RockSalt supports only x86 binaries. Ddisasm [52] is a binary disassembly and reassembly framework implemented in Datalog and supports x86, x64, ARM32, ARM64,

and MIPS32 binaries. We believe this tool is suitable to our systems' extension for binaries of these architectures, as the core components of our systems are implemented in Datalog. Meanwhile, another note is that source code is less likely to be available for legacy x86 binaries, which highlights the importance of analyzing x86 binaries.

8.2.2 C++ Binaries

Our systems' current design and implementation target binaries compiled from C programs. We believe our block memory model concept still applies to C++ generated binaries, but supporting them may require a substantial amount of both conceptual and engineering efforts, particularly in tracking virtual tables (vtables) and object references. For vtables, we believe our global region partitioning method can be extended to generate memory blocks for vtables; one observation to support our belief is that for safety concerns vtables are encoded in the read-only data section, and pointers to vtables are stored at the beginning of the object's memory region. Further, BPA and BinPointer's value tracking analysis need to be extended to track the assignment of vtable pointers into heap memory blocks generated at allocation sites (i.e., call sites of the `new` function), and to track how object references are created and propagated. To our best knowledge, there is no practical points-to analysis for stripped binaries generated from C++ code. Overall, extending our frameworks to support C++ binaries is an interesting future work.

8.3 Generating More Fine-Grained and Sound Memory Block Boundaries

Block based pointer analysis systems including BPA and BinPointer rely on block memory model where fine-grained yet sound block boundaries are crucial for analysis. As mentioned in Sec. 8.1.1, more principles binary level solutions for partitioning the global region are desired. Also, BPA and BinPointer rely on the conservative memory block generation for the heap region (treating the whole heap allocation as one block). Although BinPointer tracks offsets within the block to increase precision, it over-approximates the results and assumes that the pointer can point to every offset within the block when precise offset tracking is difficult, especially on large and complicated binaries. Therefore, more fine grained yet conservative heap memory modeling is the key to improve both

pointer analysis platforms' precision, which is an interesting direction for future work.

8.4 More Precise Type Inference

BinType proposes equivalence construction techniques to infer what alocs on the instructions are aliased with each other. While we recover some stack and global memory locations, we do not consider heap memory locations. In general, precise heap memory modeling is challenging especially at the binary level. However, precisely resolving heap memory-locations should generate more equivalence relations, which can help inferring more precise types in Sec. 7.3 as well as detecting more indirect calls' caller information in Sec. 7.4. Thus, further memory-level alias analysis is desired for future work. One promising way is to utilize BinPointer's offset tracking mechanisms to infer heap memory locations of some memory accesses where precise analysis is possible by BinPointer.

BinType distinguishes types by $\mathrm{PtrType}$ and $\mathrm{NonPtrType}$, which is coarse-grained compared to source-level types. More sophisticated analysis may allow distinguishing more fine-grained types. For example, we could distinguish code address types independently from $\mathrm{NonPtrType}$. At an initialization step, we cannot statically determine whether the code address value being assigned in an instruction (e.g. matches with the beginning address of a function) is an authentic code address or an integer value that coincidentally matches the code address. However, further analysis (e.g. data flow analysis) on the code address value to figure out how it is used (e.g. it flows to a program counter at an indirect call) can determine its type. More fine-grained type inference while being conservative is an interesting research direction, which we leave as a future work.

8.5 Summary of Our Work

Our goal is to build high precision and practical binary pointer analysis and CFG construction platforms that scale to real world binaries. As the first prototype, we implemented BPA that relies on the block memory model for pointer aliases. In detail, it does not track offsets within memory blocks in order to increase efficiency. As a result, BPA scales to SPEC CPU 2k6 C binaries and achieve higher precision rates for CFG generation than the state-of-the-art technique.

As an extension of BPA, we also implemented BinPointer that tracks offsets within memory blocks. Our experiments show that BinPointer is suitable for general-purpose interprocedural analysis by achieving reasonable performance and higher accuracy than BPA. Last, our type based approach BinType performs equivalence relation generation methods to infer arity-types for indirect call target refinement. BinType achieves higher performance than BPA and comparable precision results as BPA. Our systems BPA, BinPointer, and BinType are compatible with binaries compiled by different compilers and optimization levels. Thanks to the block memory model and Datalog implementations, they produce precise results and scale to real-world binaries.

Bibliography

[1] MORRISETT, G., G. TAN, J. TASSAROTTI, J.-B. TRISTAN, and E. GAN (2012) "RockSalt: Better, Faster, Stronger SFI for the x86," in *Programming Language Design and Implementation (PLDI)*, pp. 395–404.

[2] KIM, S. H., C. SUN, D. ZENG, and G. TAN (2021) "Refining Indirect Call Targets at the Binary Level," in *Network and Distributed System Security Symposium (NDSS)*.

[3] EVANS, I., F. LONG, U. OTGONBAATAR, H. SHROBE, M. RINARD, H. OKHRAVI, and S. SIDIROGLOU-DOUSKOS (2015) "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *22nd ACM Conference on Computer and Communications Security (CCS)*, pp. 901–913.

[4] CARLINI, N., A. BARRESI, M. PAYER, D. WAGNER, and T. R. GROSS (2015) "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *24th Usenix Security Symposium*, pp. 161–176.

[5] FARKHANI, R. M., S. JAFARI, S. ARSHAD, W. ROBERTSON, E. KIRDA, and H. OKHRAVI (2018) "On the Effectiveness of Type-based Control Flow Integrity," in *Annual Computer Security Applications Conference*, pp. 28–39.

[6] ABADI, M., M. BUDIU, Ú. ERLINGSSON, and J. LIGATTI (2005) "Control-flow integrity," in *12th ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353.

[7] TICE, C., T. ROEDER, P. COLLINGBOURNE, S. CHECKOWAY, Ú. ERLINGSSON, L. LOZANO, and G. PIKE (2014) "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *23rd Usenix Security Symposium*.

[8] NIU, B. and G. TAN (2014) "Modular Control-Flow Integrity," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 577–587.

[9] GE, X., N. TALELE, M. PAYER, and T. JAEGER (2016) "Fine-Grained Control-Flow Integrity for Kernel Software," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 179–194.

[10] LU, K. and H. HU (2019) "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *26th ACM Conference on Computer and Communications Security (CCS)*, pp. 1867–1881.

[11] VAN DER VEEN, V., D. ANDRIESSE, M. STAMATOGIANNAKIS, X. CHEN, H. BOS, and C. GIUFFRDIA (2017) "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *24th ACM Conference on Computer and Communications Security (CCS)*, pp. 1675–1689.

[12] KHANDAKER, M. R., W. LIU, A. NASER, Z. WANG, and J. YANG (2019) "Origin-sensitive control flow integrity," in *28th Usenix Security Symposium*, pp. 195–211.

[13] BALAKRISHNAN, G. and T. REPS (2004) "Analyzing Memory Accesses in x86 Executables," in *International Conference on Compiler Construction (CC)*, pp. 5–23.

[14] SHOSHITAISHVILI, Y., R. WANG, C. SALLS, N. STEPHENS, M. POLINO, A. DUTCHER, J. GROSEN, S. FENG, C. HAUSER, C. KRUEGEL, ET AL. (2016) "SoK:(state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 138–157.

[15] ZHANG, Z., W. YOU, G. TAO, G. WEI, Y. KWON, and X. ZHANG (2019) "BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," *Proceedings of the ACM on Programming Languages*, **3**(OOPSLA), pp. 1–31.

[16] ZHANG, Z., Y. YE, W. YOU, G. TAO, W.-C. LEE, Y. KWON, Y. AAFER, and X. ZHANG (2021) "OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 813–832.

[17] REDINI, N., R. WANG, A. MACHIRY, Y. SHOSHITAISHVILI, G. VIGNA, and C. KRUEGEL (2019) "BinTrimmer: Towards static binary debloating through abstract interpretation," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, Springer, pp. 482–501.

[18] AGADAKOS, I., N. DEMARINIS, D. JIN, K. WILLIAMS-KING, J. ALFAJARDO, B. SHTEINFELD, D. WILLIAMS-KING, V. P. KEMERLIS, and G. PORTOKALIDIS (2020) "Large-scale debloating of binary shared libraries," *Digital Threats: Research and Practice*, **1**(4), pp. 1–28.

[19] ZHANG, H., M. REN, Y. LEI, and J. MING (2022) "One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 255–270.

[20] QUACH, A., A. PRAKASH, and L. YAN (2018) "Debloating Software through Piece-Wise Compilation and Loading," in *27th USENIX security symposium (USENIX Security 18)*, pp. 869–886.

[21] HE, L., H. HU, P. SU, Y. CAI, and Z. LIANG (2022) "FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2497–2512.

[22] WANG, G., S. CHATTOPADHYAY, I. GOTOVCHITS, T. MITRA, and A. ROYCHOUDHURY (2019) "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, **47**(11), pp. 2504–2519.

[23] COVA, M., V. FELMETSGER, G. BANKS, and G. VIGNA (2006) "Static detection of vulnerabilities in x86 executables," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, IEEE, pp. 269–278.

[24] HERNANDEZ, G., F. FOWZE, D. TIAN, T. YAVUZ, and K. R. BUTLER (2017) "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2245–2262.

[25] IJAZ, M., M. H. DURAD, and M. ISMAIL (2019) "Static and dynamic malware analysis using machine learning," in *2019 16th International bhurban conference on applied sciences and technology (IBCAST)*, IEEE, pp. 687–691.

[26] MING, J., D. XU, Y. JIANG, and D. WU (2017) "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 253–270.

[27] DUAN, Y., X. LI, J. WANG, and H. YIN (2020) "DeepBinDiff: Learning program-wide code representations for binary diffing," in *Network and distributed system security symposium.*

[28] ULLAH, S. and H. OH (2021) "BinDiffNN: Learning Distributed Representation of Assembly for Robust Binary Diffing against Semantic Differences," *IEEE Transactions on Software Engineering*, **48**(9), pp. 3442–3466.

[29] Van Der Veen, V., E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida (2016) "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 934–953.

[30] Muntean, P., M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert (2018) "tauCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pp. 423–444.

[31] Lin, Y. and D. Gao (2021) "When function signature recovery meets compiler optimization," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 36–52.

[32] Kim, S. H., D. Zeng, C. Sun, and G. Tan (2022) "BinPointer: towards precise, sound, and scalable binary-level pointer analysis," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pp. 169–180.

[33] Leroy, X. and S. Blazy (2008) "Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations," *Journal of Autom. Reasoning*, **41**(1), pp. 1–31.

[34] Andersen, L. O. (1994) *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, University of Copenhagen.

[35] Steensgaard, B. (1996) "Points-to analysis in almost linear time," in *23rd ACM Symposium on Principles of Programming Languages (POPL)*, pp. 32–41.

[36] Lattner, C., A. Lanharth, and V. Adve (2007) "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Programming Language Design and Implementation (PLDI)*, pp. 278–289.

[37] Sui, Y. and J. Xue (2016) "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th international conference on compiler construction*, pp. 265–266.

[38] Bravenboer, M. and Y. Smaragdakis (2009) "Strictly declarative specification of sophisticated points-to analyses," in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 243–262.

[39] Kastrinis, G. and Y. Smaragdakis (2013) "Hybrid context-sensitivity for points-to analysis," *ACM SIGPLAN Notices*, **48**(6), pp. 423–434.

[40] Smaragdakis, Y., G. Kastrinis, and G. Balatsouras (2014) "Introspective analysis: context-sensitivity, across the board," in *Programming Language Design and Implementation (PLDI)*, pp. 485–495.

[41] Balatsouras, G. and Y. Smaragdakis (2016) "Structure-sensitive points-to analysis for C and C++," in *International Static Analysis Symposium*, pp. 84–104.

[42] Grech, N. and Y. Smaragdakis (2017) "P/Taint: unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, **1**(OOPSLA), pp. 1–28.

[43] Ceri, S., G. Gottlob, L. Tanca, et al. (1989) "What you always wanted to know about Datalog(and never dared to ask)," *IEEE transactions on knowledge and data engineering*, **1**(1), pp. 146–166.

[44] Szabó, T., S. Erdweg, and G. Bergmann (2021) "Incremental whole-program analysis in Datalog with lattices," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 1–15.

[45] Schilling, J. and T. Müller (2022) "VANDALIR: Vulnerability Analyses Based on Datalog and LLVM-IR," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, pp. 96–115.

[46] Zeng, D. and G. Tan (2018) "From Debugging-Information Based Binary-Level Type Inference to CFG Generation," in *8th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 366–376.

[47] Zhang, M. and R. Sekar (2013) "Control Flow Integrity for COTS Binaries," in *22nd Usenix Security Symposium*, pp. 337–352.

[48] Zhu, W., Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang (2023) "Callee: Recovering call graphs for binaries with transfer and contrastive learning," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 2357–2374.

[49] Brumley, D., I. Jager, T. Avgerinos, and E. J. Schwartz (2011) "BAP: A Binary Analysis Platform," in *Computer Aided Verification (CAV)*, pp. 463–469.

[50] Balakrishnan, G., R. Gruian, T. Reps, and T. Teitelbaum (2005) "CodeSurfer/x86—A platform for analyzing x86 executables," in *International Conference on Compiler Construction (CC)*, Springer, pp. 250–254.

[51] Hex-Rays (2008), "The IDA Pro disassembler and debugger," https://www.hex-rays.com/products/ida/.

[52] Flores-Montoya, A. and E. Schulte (2020) "Datalog disassembly," in *29th Usenix Security Symposium*.

[53] Agency, N. S. (2017), "Ghidra Reverse Engineering Tool," https://www.nsa.gov/resources/everyone/ghidra/.

[54] ARNBORG, S. (1974) "Optimal memory management in a system with garbage collection," *BIT*, pp. 375–81.

[55] EMMERIK, M. and T. WADDINGTON (2004) "Using a decompiler for real-world source recovery," in *11th Working Conference on Reverse Engineering*, IEEE, pp. 27–36.

[56] LIN, Z., X. ZHANG, and D. XU (2010) "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, pp. 1–1.

[57] LEE, J., T. AVGERINOS, and D. BRUMLEY (2011) "TIE: Principled reverse engineering of types in binary programs," .

[58] DEWEY, D. and J. T. GIFFIN (2012) "Static detection of C++ vtable escape vulnerabilities in binary code." in *NDSS*.

[59] ZHANG, C., C. SONG, K. Z. CHEN, Z. CHEN, and D. SONG (2015) "VTint: Protecting Virtual Function Tables' Integrity." in *NDSS*.

[60] CHRISTODORESCU, M., N. KIDD, and W.-H. GOH (2005) "String analysis for x86 binaries," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 88–95.

[61] PEI, K., J. GUAN, M. BROUGHTON, Z. CHEN, S. YAO, D. WILLIAMS-KING, V. UMMADISETTY, J. YANG, B. RAY, and S. JANA (2021) "StateFormer: Fine-grained type recovery from binaries using generative state modeling," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 690–702.

[62] MAIER, A., H. GASCON, C. WRESSNEGGER, and K. RIECK (2019) "TypeMiner: Recovering Types in Binary Programs Using Machine Learning," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, Springer, pp. 288–308.

[63] CHUA, Z. L., S. SHEN, P. SAXENA, and Z. LIANG (2017) "Neural Nets Can Learn Function Type Signatures From Binaries." in *USENIX Security Symposium*, pp. 99–116.

[64] HE, J., P. IVANOV, P. TSANKOV, V. RAYCHEV, and M. VECHEV (2018) "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1667–1680.

[65] HELLENDOORN, V. J., C. BIRD, E. T. BARR, and M. ALLAMANIS (2018) "Deep learning type inference," in *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 152–162.

[66] SMARAGDAKIS, Y. and G. BALATSOURAS (2015) "Pointer Analysis," *Foundations and Trends in Programming Languages*, **2**(1), pp. 1–69.

[67] CONWAY, C. L., D. DAMS, K. S. NAMJOSHI, and C. BARRETT (2008) "Pointer Analysis, Conditional Soundness, and Proving the Absence of Errors," in *15th International Symposium on Static Analysis*, pp. 62–77.

[68] TAN, G. and T. JAEGER (2017) "CFG Construction Soundness in Control-Flow Integrity," in *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, pp. 3–13.

[69] AYCOCK, J. and N. HORSPOOL (2000) "Simple generation of static single-assignment form," in *International Conference on Compiler Construction (CC)*, Springer, pp. 110–125.

[70] LU, Y., L. SHANG, X. XIE, and J. XUE (2013) "An Incremental Points-to Analysis with CFL-Reachability," in *International Conference on Compiler Construction (CC)*, pp. 61–81.

[71] SAHA, D. and C. RAMAKRISHNAN (2005) "Incremental and demand-driven points-to analysis using logic programming," in *International conference on Principles and practice of declarative programming*, pp. 117–128.

[72] LUK, C.-K., R. COHN, R. MUTH, H. PATIL, A. KLAUSER, G. LOWNEY, S. WALLACE, V. J. REDDI, and K. HAZELWOOD (2005) "Pin: building customized program analysis tools with dynamic instrumentation," in *Programming Language Design and Implementation (PLDI)*, pp. 190–200.

[73] JORDAN, H., B. SCHOLZ, and P. SUBOTIĆ (2016) "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*, Springer, pp. 422–430.

[74] SUI, Y. and J. XUE (2016) "On-demand strong update analysis via value-flow refinement," in *International Conference on Software engineering (ICSE)*, pp. 460–473.

[75] SUI, Y., D. YE, and J. XUE (2014) "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, **40**(2), pp. 107–122.

Vita

Sun Hyoung Kim

EDUCATION

- **Ph.D. candidate in Computer Science** December 2023
  - Pennsylvania State University, University Park, PA Advisor: Prof. Gang Tan
- **B.A. in Computer Science and Mathematics** December 2016
  - University at Buffalo, The State University of New York, Buffalo, NY

PROFESSIONAL EXPERIENCE

- **NVIDIA** Austin, TX
  Backend Compiler Engineer December 2023 ∼ Present
  - Adding supports for NVIDIA GPU compiler backend.
- **Pennsylvania State University** University Park, PA
  Graduate Research Assistant August 2017 ∼ May 2022
  - Conducted research on binary pointer analysis, and designed and implemented BPA, BinPointer, and BinType.
- **NVIDIA** Austin, TX
  Compiler Software Engineer Intern May 2021 ∼ August 2021
  - Implemented an instruction scheduling optimization pass for NVIDIA GPU compiler backend.

PUBLICATIONS

- **Sun Hyoung Kim**, Dongrui Zeng, Cong Sun, and Gang Tan. BinPointer: Towards Precise, Sound, and Scalable Binary-Level Pointer Analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (**CC**), 2022.
- **Sun Hyoung Kim**, Cong Sun, Dongrui Zeng and Gang Tan. Refining Indirect Call Targets at the Binary Level. In *Proceedings of the 28th Network and Distributed System Security Symposium* (**NDSS**), 2021.
- Jeffrey C Murphy, Bhargav Shivkumar, Amy Pritchard, Grant Iraci, Dhruv Kumar, **Sun Hyoung Kim**, and Lukasz Ziarek. A Survey of Real-time Capabilities in Functional Languages and Compilers. *Concurrency and Computation: Practice and Experience*, 2019.
- **Sun Hyoung Kim**, Dongrui Zeng, and Gang Tan. BinType: Type based Indirect Call Target Refinement on Binary Programs. Under Submission.