World Journal of Information Technology
ISSN: 2959-9911
DOI: https://doi.org/10.61784/wjit3024

BINARY PROGRAM DEPENDENCE ANALYSIS: TECHNIQUES, CHALLENGES, AND FUTURE DIRECTIONS

ChunFang Li[1,2], Yu Wen[1*], Dan Meng[2,3]
[1]State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China.
[2]Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China.
[3]School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China.
Corresponding Author: Yu Wen, Email: wenyu@iie.ac.cn

Abstract: Binary program dependence analysis is pivotal for security applications such as vulnerability detection and malware analysis, yet faces significant challenges due to path explosion, indirect branches, and over-approximation. This survey systematically examines state-of-the-art techniques, including value set analysis (VSA), path-sampling methods (BDA, DueForce), block memory models (BPA, BinPointer), and machine learning approaches (NeuDep), to address three core research questions: (1) how existing methods achieve scalability, (2) the compromises made in scalability and their impact on precision/soundness, and (3) alternative strategies to transcend these tradeoffs. We propose a three-dimensional analytical framework—methodological taxonomy, empirical evaluation, and forward-looking synthesis—to categorize 11 representative tools and evaluate their performance on the SPEC CINT 2000 benchmark. Key findings reveal that path-sampling methods like BDA balance soundness and efficiency but struggle with complex control flow, while machine learning-based NeuDep mitigates false positives through hybrid modeling. Dynamic analysis (DueForce) prioritizes precision but suffers from scalability limitations. Our contributions include a novel taxonomy exposing precision-soundness-scalability tradeoffs, a refined evaluation methodology integrating symbolic execution for accuracy validation, and pioneering pathways for next-generation analysis via sparse value-flow analysis. The results underscore the need for context-aware strategies to handle modern software complexity, offering actionable insights for advancing binary analysis in security hardening and vulnerability defense.
Keywords: Dependence analysis; Binary analysis; Static analysis; Path explosion; Abstract interpretation

1 INTRODUCTION

Binary program dependence analysis serves as a critical foundation for binary code analysis, encompassing two core components: data dependence analysis and control dependence analysis [1]. Data dependence analysis identifies define-use relationships between instruction operands [1,2], while control dependence analysis traces define-use chains involving status registers in conditional branches. In dynamic binary analysis, dependence analysis mitigates over-approximation-induced false positives by comprehensively modeling feasible execution paths (via control dependence) and reconstructing precise data flows (via data dependence) [3]. These derived paths and flows enable downstream security applications such as vulnerability discovery [4-8], malware analysis [1,3,9-12], and software hardening[13-16], forming the basis for robust program understanding and transformation.
According to statistics from the National Institute of Standards and Technology (NIST) [17], the most common newly added vulnerability types from 2019 to 2023 are as follows: memory safety vulnerabilities (38%), input validation vulnerabilities (22%), logic errors (18%), configuration and permission vulnerabilities (12%), and other types (10%). Among them, all memory safety vulnerabilities, along with some input validation and logic error vulnerabilities, can be identified through binary data flow-based analysis. About 65% of these vulnerabilities can be exploited on binary programs without source code. For example, the CVE-2021-3156 (sudo heap buffer overflow vulnerability) disclosed in 2021 affected millions of Linux devices worldwide due to unvalidated command-line argument length (CWE-119); the CVE-2022-23521 (Git integer overflow vulnerability) disclosed in 2023 allowed arbitrary read-write of heap memory (CWE-787) due to improper handling of path pattern counts (CWE-190), which could lead to remote code execution; and the CVE-2024-3171 (Google Chrome use-after-free vulnerability) disclosed in 2024 was caused by improper tracking of accessibility object lifecycles (CWE-416), leading to heap corruption. These vulnerabilities pose threats to operating systems, applications, and the entire network environment through different triggering paths, offering attackers opportunities for remote code execution, privilege escalation, or bypassing security mechanisms, further highlighting the importance of binary analysis in vulnerability defense.
Currently, dynamic analysis methods such as fuzzing [18] are widely applied in the field of binary vulnerability discovery. However, since execution paths largely depend on input data and environment variables, dynamic analysis is unable to achieve comprehensive coverage of all feasible paths [1,3,19]. It also cannot automate software hardening or provide sound hardening strategies based on its poor path coverage. In contrast, static analysis theoretically guarantees execution path coverage [20-23] and can reuse analysis results from different paths through abstract interpretation [24-26] to improve efficiency. Furthermore, abstract interpretation provides guarantees for the soundness of security hardening methods by describing program behaviors and vulnerability triggering paths[1,20,21,27], which also ensures that no new logical issues are introduced during the patching process. It should be noted that traditional binary static

© By the Author(s) 2025, under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0).

analysis, relying solely on executable file structure and individual instruction semantics, struggles to handle complex control structures like indirect branch instructions [20,28]. Therefore, existing research [1,3,29,30] often combines techniques such as symbolic execution [26,31,32] to construct more accurate control flow graphs (CFGs) and identify all dependencies.

Although static binary analysis based on dependence analysis has advantages over dynamic analysis in path coverage and soundness, with the increasing complexity of modern software and the rise of various code obfuscation techniques, traditional symbolic execution-based binary dependence analysis faces significant challenges in scalability, primarily due to path explosion caused by complex control flows [1]. In recent years, academia has proposed several methods to mitigate path explosion by balancing analysis precision, coverage, and efficiency to achieve more scalable binary program dependence analysis [1-3,33,34]. In summary, we focus on three research questions:

- **RQ1** - How do existing analysis methods achieve scalability?
- **RQ2** - What compromises have existing analysis methods made in pursuit of scalability, and how do these compromises affect analysis precision and soundness?
- **RQ3** - Are there methods from other related research areas that can replace these compromise strategies without sacrificing precision and soundness?

Table 1 The Sensitivity of Existing Tools for Binary Dependence Analysis

Category	Tool	Flow-Sensitive	Context-Sensitive	Path-Sensitive
Alias Analysis	Alto[35]	●	○	○
Dependence Analysis	Salto[28]	●	○	○
Dependence Analysis	VSA[20]	●	○	○
Dependence Analysis	CodeSurfer[27]	●	●	○
Dependence Analysis	BDA[1]	●	●	○
Force Execution	DueForce[3]	●	●	●
Points-to Analysis	BPA[29]	●	●	○
Points-to Analysis	BinPointer[30]	●	●	○
Alias Analysis	RENN[33]	●	○	○
Alias Analysis	DEEPVSA[34]	●	○	○
Dependence Analysis	NeuDep[2]	●	●	○

To answer the research questions above, we systematically investigate existing approaches through a three-dimensional analytical framework: methodological taxonomy, empirical evaluation, and forward-looking synthesis. First, we establish theoretical foundations by formalizing key concepts of binary dependence analysis and dissecting the interplay between different analysis sensitivities (Section 2). This framework enables us to systematically categorize 11 representative methods in Table 1 through our novel taxonomy (Section 4), revealing inherent tradeoffs between precision, soundness, and scalability (RQ1). Building on this comprehensive survey, we conduct focused empirical evaluation of three paradigmatic implementations - selected for their contrasting compromise strategies - using real-world programs from the SPEC CINT 2000 benchmark (Section 5). This targeted analysis quantitatively demonstrates how architectural differences impact practical effectiveness, particularly in handling path explosion and indirect branches (RQ2). Finally, by synthesizing insights from software engineering, formal methods, and machine learning domains (Section 7), we propose novel pathways to transcend traditional tradeoffs through sparse value-flow analysis and neural-symbolic integration (RQ3). Our multi-stage investigation progresses from fundamental principles to concrete implementations, culminating in a unified evaluation framework and actionable directions for next-generation analysis systems.

Overall, our main contributions are as follows:

- We develop a novel framework that systematically categorizes existing techniques, revealing fundamental tradeoffs between precision, soundness, and scalability across 11 state-of-the-art methods.
- Building upon the experiment setup from BDA, we provide a more systematic evaluation method to reveal the precision of different binary dependence analysis method.
- Finally, we discuss our observations on the existing sparse value-flow analysis and pioneer sparse value-flow analysis as a viable pathway for next-generation dependence analysis on binary programs.

2 PRELIMINARY

In this section, we aim to establish the theoretical foundation and overall understanding of binary dependence analysis. First, we systematically introduce the key concepts in binary analysis, including alias analysis, points-to analysis, data dependence analysis, and control dependence analysis. Following that, we introduce different sensitivities of analysis in

data dependence analysis and their implications for analysis methods. Then, we present the common process for various binary data dependence analysis methods, laying a solid foundation for the in-depth discussion in the following sections. Finally, we discuss the existing evaluation methods and criteria as the fundamental design of our innovative evaluation method.

2.1 Concepts of Binary Dependence Analysis

In the field of software analysis, dependence analysis was first used to identify data dependencies between statements in high-level languages (such as C/C++, FORTRAN) [36] to assist in parallel design and compiler optimization. To achieve instruction-level parallelism through instruction scheduling, Amme, et al. [28] combined binary alias analysis with high-level language data dependence analysis, thus introducing the earliest form of binary data dependence analysis. Since control dependence analysis can be considered as a combination of data dependence analysis results, its main function is to verify the feasibility of execution paths. Therefore, binary data dependence analysis can simultaneously complete control dependence analysis. We will first introduce the foundational concepts of binary alias analysis and points-to analysis, and then define the problem of binary dependence analysis based on existing methods.

2.1.1 Alias and points-to analysis on binary programs

In binary analysis, both alias analysis and points-to analysis focus on memory operands [29,30,35,37-39]. Alias analysis is used to determine whether different operands will point to the same memory location along a particular execution path [20,33-35,39], while points-to analysis identifies the memory address that a memory operand points to along different execution paths [29,30]. Since memory operands in binary programs are only identified by operand addresses composed of registers and values, as well as operand lengths, alias analysis or points-to analysis for the entire binary program treats all memory operands as dereferenced pointer variables. Furthermore, when relative symbolic expressions are allowed to represent memory address results in points-to analysis, binary alias analysis and points-to analysis become equivalent analytical processes [37].

2.1.2 Data dependence analysis on binary programs

In binary analysis, data dependence analysis is primarily used to identify the define-use relationships between operands in instructions [1,2,20,21]. Since the register values in the memory operand address expressions vary across different execution paths or instructions, when performing data dependence analysis on an instruction with a memory operand as the source operand, it is necessary to first determine the alias operands of this memory operand along certain execution paths through alias analysis. These alias operands may form define-use, use-use, use-define, or unreachable relationships with the analyzed memory operand [28]. Binary data dependence analysis filters out the instruction pairs that can form define-use relationships as the instruction pairs containing data dependencies.

Definition 1. In a binary program, for a pair of instructions I_1 and I_2, where O_1 is the destination operand of I_1 and O_2 is a source operand of I_2, both of which are memory operands, if there exists a feasible execution path π that passes through I_1 and I_2 such that O_1 and O_2 are alias operands, and there are no instructions I_3 on path π between I_1 and I_2 with a destination operand that is an alias of O_2, then it can be said that there is a data dependence from I_2 on I_1.

Based on the above definition, the operands involved in binary data dependence include both register operands and memory operands. Binary data dependence analysis that focuses solely on memory operands [1,3] is referred to as binary memory dependence analysis. Since alias relationships between register operands can be directly determined based on their names, data dependence analysis for register operands only requires verifying the feasibility of execution paths between instructions. Therefore, existing research on binary data dependence analysis methods primarily focuses on binary memory dependence analysis.

2.1.3 Control dependence analysis on binary programs

Like control dependence analysis in high-level languages, binary control dependence analysis constructs define-use chains along execution paths to determine the feasible branch targets of conditional or indirect branch statements on different execution paths [1,7,29,40-44]. This helps to limit the feasible search space for data dependencies or data flow in subsequent analyses and completes control flows involving indirect branches that pure static analysis cannot resolve. Due to the interdependence between data dependence and control dependence, existing analysis methods often require iteratively alternating between data dependence and control dependence analysis until the analysis results reach a fixed point.

Definition 2. In a binary program, for a pair of instructions I_1 and I_2, where I_2 is a conditional branch instruction or an indirect branch instruction, if there exists a feasible execution path π that passes through I_1 and I_2 such that there is a define-use chain between I_1 and I_2, then it can be said that there is a data dependence between I_1 and I_2.

2.2 Sensitivities of Analysis

In binary dependence analysis, different methods selectively maintain varying levels of sensitivity, such as flow sensitivity, context sensitivity, and path sensitivity, to balance precision, soundness, and analysis efficiency.

2.2.1 Flow sensitivity

Flow sensitivity requires that the analysis method determine the execution order of different program statements based on the control flow [20,28,45]. In binary dependence analysis, ensuring flow sensitivity helps distinguish true data dependencies (define-use relationships) from anti-dependencies (use-define relationships) [28]. Since the execution

order between instructions must be maintained and evaluated, flow-sensitive analysis methods incur more time and memory overhead compared to flow-insensitive methods. However, this overhead is necessary to implement a usable binary dependence analysis method [29].

2.2.2 Context sensitivity

Context sensitivity requires that the analysis method maintain the program's call stack, recording the call and return relationships between different functions [21,27,46-48]. During the traversal of program statements, context-sensitive analysis methods can determine the call chain that reaches the current function based on the call stack state and the push/pop records. The call chain will facilitate the context-sensitive methods to identify potential data interactions between functions. In binary dependence analysis, ensuring context sensitivity is essential for accurate inter-function analysis. When traversing a binary program along its control flow, context-sensitive analysis methods can not only determine the data or control dependencies between instructions of the current function and externally callees through parameter registers and specific call sites, but also identify data or control dependencies within the control flow graph of the current function from callees, based on their parameters, return values, and different call sites.

Obviously, maintaining the call stack and distinguishing between different call sites requires additional time and memory. As the complexity of the program's call graph increases and recursive structures are introduced, existing analysis methods are unable to fully achieve complete context sensitivity. Some early methods, such as the original VSA[20], abandoned context sensitivity. To balance precision and analysis efficiency, current methods typically use techniques to limit call stack depth to implement partially context-sensitive analysis [23,31,32].

2.2.3 Path sensitivity

Path sensitivity requires the analysis method to distinguish between data flow states from different execution paths and determine whether each path is feasible [1,49-51]. Consequently, state information from different branches cannot be directly merged at control flow join points. According to Definitions 1 and 2, accurate binary dependence analysis demands path sensitivity to ensure traceable data dependencies. However, maintaining path sensitivity often leads to a significant increase in computational complexity [20].

Firstly, analysis methods that do not allow path merging inevitably face the issue of path explosion [23,26,52]. Subsequently, determining path feasibility typically involves using an SMT solver to treat all branch conditions on a path as constraints and assess the model's feasibility [26,49-51,53]. As the number of conditional branches increases, the complexity of the path feasibility model grows, making the problem increasingly NP-hard and potentially unsolvable [23,26,43]. Compared to high-level language programs, binary programs usually require more complex instruction combinations to achieve the same functionality, resulting in longer execution paths and more path constraints. Therefore, existing binary dependence analysis methods [1,20,27,39] avoid considering path sensitivity to mitigate path explosion and constraint explosion.

2.3 Process of Binary Dependence Analysis

Although binary dependence analysis and its associated alias analysis and points-to analysis have been studied for over 30 years, the structural characteristics of binary programs and the features of assembly language have not fundamentally changed [1,3,20,28-30]. As a result, different binary dependence analysis methods still follow the same basic process framework. As shown in Figure 1, all binary dependence analysis methods consist of five parts, arranged from bottom to top: disassembly, inter-procedural control flow graph (iCFG) construction, abstract interpretation, partial alias analysis, and dependency construction. All these modules will support the indirect branch target inference from the indirect branch complement module that could direct other modules to complement their analysis processes.

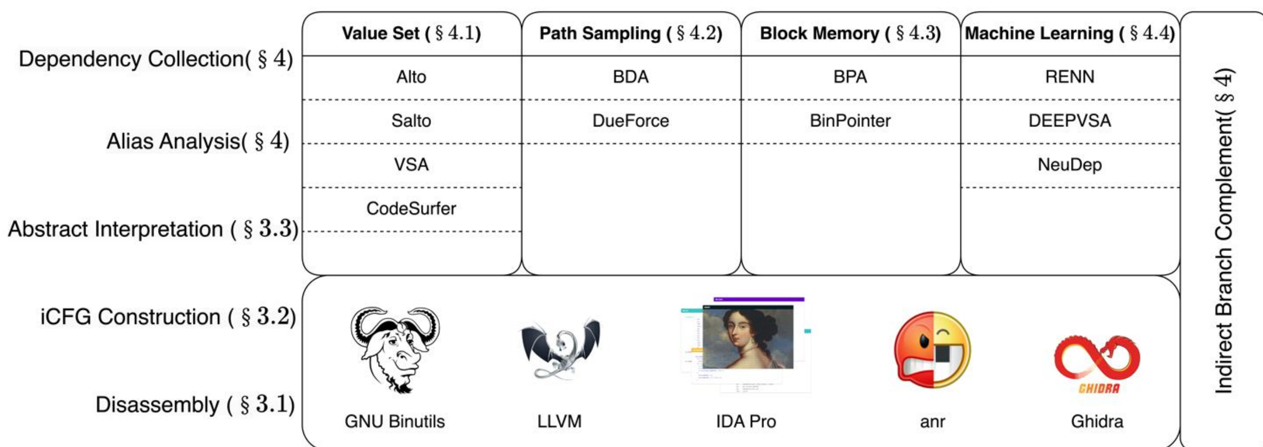

Figure 1 The Common Architecture of Existing Binary Dependence Analysis Methods

Binary dependence analysis identifies data dependencies by analyzing instruction sequences in assembly code extracted from executable files. This process relies on foundational binary analysis frameworks that implement three core components: (1) disassembly methods to recover assembly code [54-61], (2) function partitioning techniques to

delineate code boundaries [62-64], and (3) iCFG construction to model program execution paths [41,43,61,65]. The granularity of iCFG nodes varies across methods: nodes may represent individual instructions or basic blocks, with edges dynamically defined to reflect control flow transitions between these units [23,41].
To achieve architecture-agnostic analysis and simplify downstream tasks, most approaches build abstract interpretations or intermediate representations (IR) atop the iCFG [1,20,29]. These abstractions decouple analysis logic from low-level instruction semantics while enabling integration with diverse alias analysis techniques to resolve memory operands. Dependencies are then derived by tracking data flows along control paths, correlating aliases with their definition-use chains.
A critical challenge lies in resolving indirect branches (e.g., jumps via registers or computed addresses). Modern methods address this by inferring branch targets after accomplishing the necessary dependency collection, dynamically refining the iCFG to incorporate resolved control flow edges [1,29,30]. This process may occur incrementally during dependency collection or as a post-processing step, allowing flexibility to revisit and refine earlier analysis stages. By unifying disassembly, iCFG construction, abstract interpretation, and alias resolution, binary dependence analysis frameworks achieve both precision in dependency extraction and adaptability across instruction sets and optimization patterns.

2.4 Existing Evaluation Setup for Binary Dependence Analysis

Due to the complexity of programs used in the real world, there is still no feasible method to identify all data dependencies in a binary program. As a result, existing evaluation methods rely on multiple metrics to compare their results with selected benchmark methods under specific conditions. Early comparative methods focused on ablation experiments of core analysis techniques based on basic analysis frameworks, comparing the analysis results for the same program, including the number of discovered data dependencies and performance metrics such as analysis time and memory usage. Since Value Set Analysis (VSA) [20,21,27] dominates in binary dependence analysis, recent research methods compare their results with VSA or its variants [27,34], while also considering other existing methods to demonstrate general advantages.
The experimental methods used in the paper of BDA [1] have been widely adopted in subsequent binary dependence analysis research. Unlike early comparison methods, this evaluation approach primarily focuses on data dependencies caused by memory operands, i.e., comparing binary memory dependence analysis results. Specifically, the evaluation method first uses dynamic analysis [66] to obtain memory dependencies with high code coverage as reference dependencies. By comparing with the reference dependencies, it identifies missing dependencies (missing dependencies) and additional dependencies (extra dependencies) in the results of different binary dependence analysis methods when analyzing the corresponding binary program. To verify the correctness of the additional dependencies, the evaluation method also identifies the data types involved in each instruction by designing the specific pass for LLVM compilation test samples, which then identify the false positives (incorrect dependencies) from extra dependencies if the instructions involve the same data type. When comparing different analysis methods, this evaluation method mainly compares the number of missing dependencies and additional dependencies, while using false dependencies as auxiliary indicators to determine if the difference in the number of additional dependencies primarily stems from changes in the number of false dependencies.

3 FUNDAMENTAL STAGES

In this section, we focus on the foundational technologies supporting binary dependence analysis, specifically the three underlying components of binary dependence analysis methods. We first introduce the basic principles and methods of disassembly technology, then explore the construction of key program representations such as the inter-procedural control flow graph (iCFG). Following that, we present abstract domains and abstract interpretation techniques, highlighting their core role in reducing analysis complexity and improving accuracy. These basic stages not only provide the theoretical foundation for understanding subsequent advanced methods but also offer technical support for solving practical problems in real-world applications.

3.1 Disassembly

As a fundamental technology for various binary analysis tasks, disassembly has continuously evolved over the past decades. Consequently, binary dependence analysis methods from different periods have employed different disassembly tools. Here we focus on the disassembly technologies and tools used in current binary dependence analysis techniques, which can be categorized into two types: linear scanning by instruction addresses and recursive descent parsing based on static control flow of instructions.
Among the disassembly tools using the linear scanning method, the most widely used is objdump from GNU Binutils [55]. Objdump identifies the file type and segmentation information by parsing the executable file's header. It also determines the start and end addresses of the code section based on file metadata for instruction-by-instruction disassembly. In addition, objdump supports disassembly for a given range of instruction addresses and can decode specific data sections in the provided encoding format. Another linear scanning disassembler used in binary dependence analysis is Radare2 [67]. Compared to objdump, Radare2 better supports plugin extensions and user scripts, enabling multi-round heuristic scanning to improve disassembly coverage.

Currently, more disassembly tools use recursive descent parsing methods, which can independently disassemble binary code with overlapping instructions and complex control flow without relying on plugin scripts. These tools include free and open-source tools like Dyninst [41,42], Angr [23], and Ghidra [57], as well as commercial tools like IDA Pro [56] and Binary Ninja [68]. Among them, the Hex-Rays plugin for IDA Pro is the most widely used disassembly tool. IDA Pro and its available plugins generally use pattern matching and other methods to identify code block entries and cross-references from the code section or symbol table. Hex-Rays utilizes this information to restore the control flow and uses a multi-stage recursive descent method along the control flow to achieve efficient and complete disassembly. Similarly, other tools also rely on pattern matching to identify function entries, with the main difference being in the methods used to restore control flow, such as VSA, symbolic execution, and intermediate language abstract interpretation.

Although disassembly technology has matured after decades of development, its static analysis characteristics still present three main challenges when relying solely on disassemblers. First, the removal of symbols from binary files makes many disassembly strategies ineffective, leading to difficulties in identifying some code block entry addresses [41,69,70]. This can result in incomplete control flow recovery and the inability to identify certain overlapping instructions. Second, special control flows caused by indirect branches and functions without return values often lead to failures in control flow recovery when using recursive descent parsing methods, causing incorrect identification of code block entries [7,41,65,71]. Lastly, the complex instruction scheduling during compilation and the use of tail calls lead to many disassembly tools being unable to accurately delineate function boundaries [23,41,42,69]. This can even result in instructions within the same function not being in a contiguous address range, or different functions sharing code, which further causes control flow recovery to fail.

3.2 Construction of Inter-procedural Control Flow Graph

In binary analysis, the construction of the Control Flow Graph (CFG) is the foundation for various analysis tasks. Depending on the subsequent tasks, instructions or basic blocks are used as nodes in the CFG, with control flow information connecting different nodes. Each CFG represents the control flow within a function. By merging all CFGs and connecting the call points to the entry points of the callees and the exit points of the callees to the return points, the entire program's inter-procedural control flow graph (iCFG) can be obtained. Therefore, constructing the iCFG mainly involves solving three problems: function boundary identification, function exit recognition, and the construction of indirect branch control flow edges.

First, function boundary identification aims to determine the nodes within a function's CFG while also identifying the function's entry point [41,62,63,72]. Existing binary dependence analysis methods [1,2,20,29] typically use disassembly tools [54,56,73,74] to identify function boundaries directly. When analyzing symbol-stripped binary code, decompilation tools generally combine various strategies, including function prologue pattern matching, recognizing function entry points based on exception handling sections, tracing function entries through cross-references, and scanning instructions using heuristic rules. In recent years, studies on disassembly methods have introduced more probabilistic models and machine learning approaches, including traditional bidirectional RNN models [75], attention-based Transformer models [76,77], and probabilistic models based on Bayesian networks [63].

Although function boundary identification also helps identify function entry points, when constructing the inter-procedural edges for the iCFG, it is also necessary to identify the exit points of callees. The main challenge here is recognizing tail calls within callees and recursively collecting the tail call exit points to establish the actual exit-to-return edges [7,41]. The disassembly tools used in existing binary dependence analysis methods identify tail calls in different ways. These tools use stack frame analysis to identify stack frame recovery code ending with *jmp* instructions as tail calls, with differences lying in the method used to determine which code blocks require stack frame analysis. IDA Pro and Ghidra use control flow pattern matching to identify potential tail call locations, Radare2 conducts stack frame analysis only at the highest function addresses based on function boundary identification, and Binary Ninja, Angr, and Dyninst identify the start of stack frame recovery through intermediate languages, symbolic execution, and dynamic instrumentation, respectively. Once the *jmp* instruction for a tail call is identified, the callee can be determined based on the instruction's source operand, and further recursive search for exit points is conducted.

Finally, the target address of indirect branches often cannot be determined through simple static analysis, so additional methods are designed to resolve the target of indirect branches to complete the construction of the full CFG and iCFG. The most used method involves backward slicing to trace the values of the source operands in indirect branch instructions [11,23,25,26,44,78], followed by over-approximation techniques to directly treat the obtained values as the unconditional target of indirect branch instructions, thus completing the construction of the CFG and iCFG. Some analysis methods also tired to replace backward slicing with forward slicing [16,79,80], limiting the number of explored paths to achieve infinite length slicing along a single execution path. In binary dependence analysis, control dependence analysis can more accurately identify indirect branch targets, so all analysis methods use their own control dependence analysis results, obtained through the define-use chains, to resolve the targets of indirect branches. With this enhanced control flow, a more accurate iCFG can be constructed, leading to more precise dependence analysis.

3.3 Abstract Interpretation

Abstract interpretation is a formal static program analysis technique [24], where its core function is to use mathematical models to reliably approximate program behavior, thereby achieving efficient program analysis methods. Compared to the intermediate representations (IR) used in disassembly tools [23,55], abstract interpretation reduces the amount of information to be processed by ignoring operations unrelated to the analysis task. On the other hand, it allows analysis methods to conservatively estimate uncertain values to ensure path coverage. In binary dependence analysis, abstract interpretation is generally used as the foundation for the analysis, providing an abstract expression of the disassembled binary program before the analysis, aligning with the design of subsequent dependence analysis methods.

In program analysis, the abstract interpretation method constructs an abstract domain based on lattice theory and the concrete domain of various components in the program. This process ensures that the concrete expression of each variable and other components in the concrete domain can be mapped to a unique abstract expression in the abstract domain via a specific Galois connection. On top of the abstract domain, abstract interpretation also requires the construction of transfer functions to simulate operations in program execution, including arithmetic and memory operations. According to lattice theory, traditional abstract interpretation methods also require performing a fixed-point calculation, which iteratively applies transfer functions until the concrete domain ranges of each abstract symbol converge to a definite interval. Fixed-point calculation ensures the stability of the program state obtained from the analysis, thereby guaranteeing path coverage in abstract interpretation.

During the computation of the program state's fixed point, the use of interval sets for abstract expression allows for merging abstract expressions of the same operand at control flow join points. By merging different execution path states, this approach avoids path explosion and improves analysis efficiency. When merging abstract expression sets for an operand, abstract interpretation uses over-approximation methods to ensure that the merged abstract expression covers all execution paths that pass through the merged program points. By combining fixed-point calculations with over-approximation designs used during branch merging, the soundness of the analysis results using abstract interpretation can be ensured.

In binary dependence analysis, given the complexity of analyzing the control flow of actual programs, existing analysis methods generally first partition the entire program, then abstractly interpret different components of the program as needed, and replace the fixed-point calculation in abstract interpretation with their own unique analysis strategies to achieve efficient binary dependence analysis. Existing binary analysis methods for partitioning the abstract interpretation domain are generally categorized into three types: memory region partitioning, represented by VSA [20]; execution path partitioning, represented by BDA [1]; and variable memory block partitioning, represented by BPA [29]. Among these, memory region partitioning methods have a relatively coarse granularity, and under conservative analysis strategies, they can lead to many false positives. In contrast, execution path partitioning, although able to reduce the granularity of abstract interpretation partitioning to lower false positives, can somewhat damage the soundness of abstract interpretation due to the path sampling methods used in BDA. Moreover, different execution paths often have many overlapping segments, which results in redundant computations. Another abstract interpretation method transforms program statements into Static Single Assignment (SSA) form [81], which partitions memory blocks for individual variables. This analysis method achieves the finest granularity of abstract interpretation and memory model partitioning, but it also requires more transfer function applications, leading to higher computational complexity when using similar analysis methods.

4 EXISTING BINARY DEPENDENCE ANALYSIS METHODS

In this section, we will provide a comprehensive review and summary of the current mainstream binary dependence analysis methods. By comparing strategies based on value set analysis, path sampling, variable block memory models, and machine learning, we will explore the advantages and disadvantages of each method in terms of soundness, precision, and scalability. Through a categorized discussion of these methods, we aim to reveal the suitable application scenarios and potential limitations of different strategies, providing valuable insights for researchers in selecting and improving their technical approaches.

4.1 Value Set -Based Binary Data Dependence Analysis

The use of value set-based analysis methods to simplify the analysis process in binary analysis first appeared in the alias analysis of executable code based on Alto [35]. This method combines abstract states I and modular k-residue sets M as address descriptors $\langle I, M \rangle$ for each register at specific program points. By performing forward data flow propagation and widening operations at control flow branch join points, the method transforms the address descriptors and determines whether the abstract states and modular k-residue sets in different descriptors are equivalent or have overlapping regions, which helps identify alias operands at different program points. After completing alias analysis, control flow information, implemented using Alto, can be used to filter out data dependencies from alias instructions, thus enabling data dependence analysis.

Amme, et al. [28] introduced the first value set-based binary dependence analysis in their data dependence analysis of assembly code. This method uses symbolic value sets to achieve efficient symbolic value propagation and loop processing strategies during symbolic execution. Compared to Debray et al.'s abstract expression for addresses, the symbolic value set propagation in this method includes both abstract address sets and abstract symbolic value sets. On this basis, Amme et al. first proposed the use of non-symbolic value (NSV) registers in loop analysis. These NSV

registers retain constant values within loops, which reduces unnecessary symbolic execution in loops and lowers the computational overhead of abstract interpretation during loop analysis.

Subsequently, Balakrishnan and Reps [20] proposed the Value Set Analysis (VSA), which became the dominant method in binary alias analysis and subsequent binary data dependence analysis in a decade. The core idea of VSA is to compute the set expressions of operand addresses and values along the control flow using abstract interpretation. Memory operands are represented as abstract locations (a-loc), and whether they alias is determined by checking for intersections between different a-loc value sets. As the control flow graph is traversed, VSA tracks operand lifecycles via memory states and determines the reachability of data flow between operands and the potential for affine relationships. Based on whether aliases can form a define-use relationship, VSA determines whether data dependencies exist between instructions.

The key to VSA's efficient, sound, and relatively precise design lies in three aspects:

1. **Abstract memory model**: VSA divides the memory space of a program into global, heap, and function-specific stack regions, and defines continuous memory blocks belonging to the same operand as a specific a-loc within the abstract memory model.
2. **Precise value set representation**: Based on the address expression forms of memory operands in the instruction set, VSA represents the a-loc value set as a collection of congruent integers within a restricted interval (RIC). During program traversal, as the memory model is updated by instructions, VSA performs widening on the a-loc value set at control flow join points and keeps it in the RIC form, while restricting the widening range based on affine relationships between operands in the execution flow.
3. **Dynamic control flow completion**: During control flow traversal, VSA uses specific memory model content to infer indirect branch target addresses, dynamically completing missing control flow edges.

Compared to earlier methods that also used set expressions for operand addresses and values, VSA achieves more efficient alias operand matching through simpler value set representations and memory space partitioning. Additionally, maintaining affine relationships between operand value sets allows VSA to achieve higher analysis accuracy. VSA also considers indirect branch instructions in binary programs and uses a dynamic analysis-like method to complete the control flow graph during traversal, leading to higher coverage and soundness.

Although VSA has made significant progress in accuracy and efficiency, it assumes that the base addresses for local function addresses are the stack frame register rbp or stack pointer register rsp, and the address offsets must be negative. In optimized code and malicious code analysis, this assumption leads VSA to often mistakenly treat memory operands not using rbp or rsp as global operands. VSA refers to these memory operands as indirect operands. Since the address expressions of indirect operands are not numerical and VSA is context-insensitive, VSA generalizes the a-loc of indirect operands in callees to $\top$, representing an unconstrained set, and merges these a-locs into the global memory model. This leads VSA to believe that indirect operands might be located anywhere in the global memory model, causing significant false positives in alias analysis and data dependence analysis.

To address the low precision caused by over-approximation and context insensitivity, Reps and Balakrishnan [27] introduced the GMOD merging mechanism and context-sensitive mechanisms into VSA. The GMOD information is used to track the set of operands modified within a function. During cross-function analysis, modified VSA only merges the a-locs of indirect operands in the GMOD information of the callee into the global memory model, reducing over-approximation of the global memory model state and improving cross-function analysis performance. When merging a-locs, the modified VSA uses an aggregate structure (ASI) to represent a-locs of compound data types such as arrays and structures. Combined with a VSA-ASI iterative mechanism, it repeatedly identifies elements within data structures and progressively optimizes the a-locs of these compound data types. By refining the a-loc elements contained in complex data structures, the modified VSA further improves the tracking precision of indirect memory operations. Finally, the method abstracts the context information at function call sites as a finite-length call string and implements a context-sensitive VSA using a worklist algorithm combined with the GMOD merging mechanism. Compared to the original context-insensitive VSA, this new design increased the traceable usage and definition of indirect operands in the experimental program samples from 29% and 33% to 81% and 90%, respectively.

4.2 Path-Sampling-Based Binary Data Dependence Analysis

Although context sensitivity has greatly improved the analysis accuracy of VSA, the root cause of the significant alias analysis false positives in VSA lies in the over-approximation caused by the widening of a-loc value sets during control flow join point merging, which ensures their RIC form. Specifically, when merging the a-locs of non-static address operands in the global region across functions, VSA directly converts these a-loc value sets to $\top$, resulting in severe false positives in alias analysis and data dependence analysis. To fundamentally address this issue of over-approximation caused by the widening operation of RIC-form value set merging, Zhang, et al. [1] proposed a path-sampling-based binary data dependence analysis method called BDA. Compared to previous methods, BDA focuses more on balancing analysis efficiency and precision while ensuring path coverage without significantly compromising analysis soundness. Specifically, BDA achieves precise, efficient, and fundamentally sound binary data dependence analysis through three core mechanisms:

1. **Unbiased Whole Program Path Sampling**: To ensure the diversity of sampled execution paths and maximize code coverage, BDA adopts a probability model-based path sampling method. This method determines the sampling probability of a branch based on the number of execution paths involved after each conditional branch,

and dynamically adjusts the sampling probability of branches during the path sampling process based on already sampled paths. This dynamically adjusted path sampling probability model ensures that all execution paths have an equal probability of being sampled at any stage, thus avoiding sampling bias caused by varying path lengths. To guarantee adequate path sampling and sampling efficiency, BDA also designs a probability lower bound model to estimate the number of paths to be sampled, based on the need for code coverage and the probability of a single path covering specific data dependencies.

2. **Per-Path Abstract Interpretation**: To avoid the over-approximation caused by widening a-loc value sets, BDA performs abstract interpretation on each sampled path individually. Since no path merging occurs, this abstract interpretation method is also a symbolic execution approach that uses a-loc value sets to express operands and assigns concrete values to taint sources through sampling. As a result, BDA's path-by-path abstract interpretation ensures context sensitivity and, during symbolic execution, resolves indirect branches, providing a sound intermediate representation of a single path for subsequent data dependence analysis.
3. **Posterior Data Dependence Analysis**: After abstract interpretation on all sampled paths, BDA aggregates the abstract interpretations of all paths to perform flow-sensitive and context-sensitive data dependence analysis. By aggregating the abstract interpretations of individual paths, BDA can merge paths traditionally, thus covering un-sampled paths by merging the a-loc value sets of the same operand from different sampled paths, enhancing the soundness of the data dependence analysis results. When performing data dependence analysis based on the merged abstract interpretations, BDA uses a worklist algorithm to traverse all memory operations along the cross-function control flow graph. By comparing whether the a-loc value sets for the same operand across all sampled paths are consistent, BDA can determine if the operand's a-loc is strongly updated. Data flow on strongly updated operands will also be strongly terminated, significantly reducing false positives in data dependence analysis.

Building on BDA's path sampling approach, He, et al. [3] replaced the abstract interpretation method with fuzzing in their DueForce method, designing a path sampling approach that ensures coverage of any two basic blocks, thereby achieving more efficient data dependence analysis and inferring program behavior based on data dependencies. By traversing basic blocks along the cross-function control flow graph, DueForce can more efficiently sample execution paths for potential data dependencies between different basic blocks, allowing it to identify more data dependencies with fewer sampled paths than BDA. Since DueForce uses fuzzing to collect data dependencies, it achieves path-sensitive analysis, resulting in fewer false positives than BDA. However, due to fuzzing, DueForce cannot achieve full coverage of all execution paths between any two basic blocks as abstract interpretation does. Therefore, even with relaxed time and memory usage limits, DueForce still fails to recognize some data dependencies when analyzing real-world programs.

4.3 Block Memory Model-Based Binary Data Dependence Analysis

Kim, et al. [29] proposed BPA, which leverages the concept of block memory models in compiler technology research and converts operands into SSA form based on the variable block memory they belong to, thereby implementing a fine-grained abstract interpretation. Based on the block memory model design, BPA performs pointer analysis for indirect branch source operands through Datalog-based value propagation logic, thereby completing indirect calls in the control flow graph. Specifically, BPA implements flow-sensitive and context-sensitive pointer analysis for binary programs in four steps:

1. **Input Processing and Program Expression**: BPA uses existing disassembly tools to convert the binary program into RTL language and partitions the entire program into multiple functions using both direct and indirect methods to prepare for subsequent cross-function analysis. During the traversal of binary instructions, BPA also records the instruction addresses as part of the function entry addresses. The functions using these addresses are referred to as "address-taken functions."
2. **Block Memory Model Generation**: To partition a function's memory model into appropriately sized memory blocks, BPA divides global and stack memory areas in each function based on the memory locations involved in memory access instructions and operand sizes. After partitioning the global and stack memory areas, BPA uses a heuristic method to check whether operands in different memory blocks belong to the same composite C language variable (such as arrays and structures) and merges the memory blocks that belong to the same composite variable. For the heap region, BPA partitions the memory model based on heap memory allocation points, without needing to check composite structure variables like in global and stack memory areas.
3. **Datalog-Based Value Tracking**: After generating the block memory model, BPA converts the broadly supported RTL instructions for binary instructions into SSA form, producing a memory block access intermediate representation (MBA-IR). This enables Datalog-based value tracking that focuses on register and memory block memory access operations. The value tracking propagates values along control flow for registers and memory blocks and handles operations such as register assignments, memory loads, and pointer dereferencing, merging equivalent relationships between global memory blocks and blocks.
4. **Incremental Fixed-Point Calculation**: To address the control flow graph's incompleteness caused by indirect calls, indirect jumps, and function return instructions, BPA designs an incremental control flow graph update and value tracking mechanism based on iterative fixed-point computation. By performing context-sensitive pointer analysis for specific operands, BPA infers the exact target addresses for indirect branches and function returns.

Incremental updates are then made to the control flow graph, SSA-form MBA-IR, and value tracking. BPA continues iterating until no new indirect branches or function return targets are produced.

With its block memory model-based simplification of abstraction and MBA-IR design, BPA only tracks the base address of the corresponding memory block during value tracking. For memory blocks containing simple variables, the base address is the address of the variable itself; for composite structure variables, the base address represents the address of the variable as well as its first element, allowing the tracking of multiple operand elements of a composite structure variable. When verifying whether two memory blocks have aliasing operands, BPA only requires the a-loc value sets of the two blocks to intersect. Furthermore, by applying SSA-form IR expressions, BPA can merge the value set expressions of the same memory block at the ϕ instruction in a flow-insensitive manner, which is equivalent to performing flow-sensitive analysis with non-SSA-form IR expressions. Additionally, the automation and parallel computing capabilities of the Datalog engine further improve BPA's computational performance.

While BPA improves analysis efficiency by treating all elements of a composite type of variable as the same operand during value tracking, this over-approximation design inevitably leads to significant false positives, such as confusing data dependencies and alias relationships between different elements of the same array or structure. To address the severe false positive problem in BPA, Kim, et al. [30] introduced BinPointer, a memory block offset-sensitive method, based on BPA. This method uses the same input processing, intermediate expressions, and block memory model generation as BPA. During value tracking, BinPointer combines value set analysis with Datalog-based value propagation methods. Through 0-base abstract interpretation, it performs precise data dependency analysis and alias analysis for the first elements of composite type variables. Furthermore, BinPointer can achieve more fine-grained partitioning for global and stack memory regions by utilizing the binary program's symbol table, thus improving analysis precision and recall rate.

4.4 Machine Learning-Based Binary Data Dependence Analysis

The earliest attempt to implement binary alias analysis using machine learning was RENN [33]. By applying RNNs to classify the memory regions involved in instructions, RENN can identify more non-alias instruction pairs in less time compared to traditional reverse execution tools like POMP [39] and REPT [82]. Specifically, RENN uses a bidirectional conditional GRU model as its core network structure that takes the bytecode in the instructions as input. Based on the implicit dependencies between consecutive instructions, RENN can determine the memory regions involved in each instruction, including global, stack, heap, and other regions. Instructions accessing different memory regions are not alias pairs, so RENN labels such instructions as non-alias pairs.

Since RENN can efficiently identify more non-alias instruction pairs, it allows POMP and REPT to avoid verifying these non-alias instruction pairs, thus achieving more efficient binary alias analysis. Therefore, RENN uses the bidirectional conditional GRU model to first filter potential alias pairs and then connects to POMP for verifying the remaining potential alias pairs. Based on the alias analysis results and the reachability between instructions, data dependence analysis can then be performed on the entire binary program.

A similar study to RENN is DEEPVSA [34], which also uses RNN to identify non-alias instruction pairs and then uses VSA to analyze other potential alias pairs. Compared to RENN, the core network structure of DEEPVSA is a hierarchical bidirectional LSTM (Bi-LSTM) network. First, the lower-level Bi-LSTM integrates the bytecode of the same instruction, and then the upper-level Bi-LSTM processes the encodings from different instructions. By combining information from previous and subsequent instructions, it recognizes the memory regions accessed by the instructions. Like RENN, DEEPVSA categorizes each instruction's memory region into global, stack, heap, and other regions, and marks instructions accessing different memory regions as non-alias pairs.

When DEEPVSA connects to VSA for binary alias analysis, it limits VSA's inference of operand a-locs based on the recognized memory regions for each instruction. For example, for a memory operand in an instruction whose memory region is the stack, DEEPVSA marks the global and heap regions of that operand's a-loc as $\perp$ (unreachable) at any program point. As a result, DEEPVSA helps VSA achieve more efficient binary alias analysis by narrowing the feasible range of value sets. After completing alias analysis, VSA can further integrate control flow information to perform data dependence analysis on the binary program.

While both RENN and DEEPVSA assist traditional analysis methods in achieving more efficient and accurate binary alias analysis and binary data dependence analysis, they both only perform coarse-grained classification of instruction memory regions, indirectly enabling the identification of non-alias instruction pairs, rather than directly determining alias relationships and data dependencies between different instructions. To achieve fine-grained instruction memory region classification and directly determine data dependencies between instructions, Pei, et al. [2] proposed NeuDep, a Transformer-based method that implements static data dependence analysis through staged self-supervised pretraining and supervised fine-tuning.

During the self-supervised pretraining phase, NeuDep uses dynamic testing methods to obtain some real execution path records, employing a masking mechanism to train the Transformer structure for binary program representation. NeuDep uses different MLP networks as prediction heads, to predict execution paths based on instructions, to synthesize instructions based on execution paths, and to perform bidirectional reasoning using partial execution paths and partial instructions. During this process, NeuDep employs a curriculum learning strategy, gradually transitioning from short instruction sequences and low masking rates to longer instruction sequences and higher masking rates. In the supervised fine-tuning phase, NeuDep fixes the pretraining parameters of the Transformer structure, replaces the MLP network

with a prediction head for data dependency relations between instructions, and trains using the data dependencies obtained from dynamic testing as the ground truth. Finally, NeuDep uses this fine-tuned structure for binary data dependence analysis.

5 EVALUATION OF BINARY DEPENDENCE ANALYSIS METHODS

Although the four types of analysis methods mentioned above can theoretically achieve data dependence analysis for binary programs and thus perform binary control dependence analysis, the experimental setups vary due to the different focuses of each method. For example, early VSA methods only counted the data dependencies in ablation experiments, while RENN and DEEPVSA focus solely on the coarse-grained memory partitions of instructions, allowing them to roughly determine non-alias pairs. In recent years, research on binary memory dependence analysis (such as BDA, NeuDep, and DueForce) has proposed a relatively rigorous comparative experimental setup. This setup uses memory dependencies obtained through dynamic testing with high code coverage as a reference to verify the soundness of different analysis methods. It then compares the data types of instructions with data dependencies, recorded during the compilation process, to validate the precision of different analysis methods.
To objectively assess the performance and practicality of various binary dependence analysis methods, we propose a new systematic evaluation design for binary data dependence analysis based on the experimental setups of existing memory analysis methods. First, we introduce the test dataset and then detail the selected metrics and evaluation methods. Through quantitative analysis of the test results across multiple dimensions, we provide data support for the development of future improvement solutions and establishes a sound standard framework for evaluating research outcomes in the future.

5.1 Dataset

As we develop our evaluation method based on the experiment setup from BDA and DueForce, we employ the same evaluation dataset, SPEC CINT 2000 benchmark [83], in our evaluation method. Since BDA and DueForce achieve binary data dependence analysis with previous path sampling based on the iCFG, they cannot deal with some optimized control flow structure, such as tail call. To enable the evaluation on BDA and DueForce, we compile the testing cases with O0 optimization option to avoid control flow adjustment. Nevertheless, our evaluation method still supports the evaluation on optimized testing cases though they will not present in this paper. When using this software suite to validate binary dependence analysis methods, the test samples are first sorted based on their size, and testing is conducted in ascending order of sample size, continuing until a sample's dependence analysis cannot be completed within an acceptable runtime.

5.2 Performance Metric

The main objectives of binary data dependence analysis include two aspects: precision and soundness. Precision aims to measure the proportion of true data dependencies among the data dependencies identified by the analysis method, while soundness concerns the extent to which the analysis method ensures the completeness of the collected data dependencies. However, the complexity of the control flow in binary programs makes execution paths unenumerable, leading to the theoretical inability to validate data dependencies on all execution paths. Therefore, existing analysis methods typically use indirect analysis metrics and comparisons with other methods to highlight their superior analytical capabilities.
Among the evaluation methods used by existing analysis methods, the experimental setups adopted by BDA and DueForce are relatively more rigorous, providing a more comprehensive measurement of the effectiveness of the analysis methods. For instance, early analysis methods such as VSA, RENN, and DEEPVSA focus on only one aspect of data dependence analysis or alias analysis, such as the number of identified data dependencies or non-alias instruction pairs. These methods only reflect the soundness or precision of the analysis method. Although BDA can evaluate analysis methods from both precision and soundness perspectives, its method for identifying data dependence false positives based on LLVM-IR data types is still not sufficiently accurate. This is primarily due to the generality of the LLVM-IR design, where a single LLVM-IR data type can correspond to multiple different high-level language types, leading to potential failure in detecting some false positives using this evaluation method.
To achieve a more accurate evaluation of the effectiveness of binary data dependence analysis methods, we modify the experimental setups used in methods like BDA and proposes a new evaluation method that better measures the precision of different analysis methods.

5.2.1 Soundness testing

To verify the reliability of different analysis methods, we adopt the same experimental setup as BDA, DueForce, and NeuDep. The experimental setup requires using dynamic analysis instrumentation to obtain real data dependencies as reference dependencies while ensuring code coverage. The analytical reliability is evaluated by comparing the number of reference dependencies identified by different methods for the same test samples. Although dynamically acquired data dependencies inevitably cannot cover all execution paths and data dependencies, their high code coverage guarantees the diversity of execution paths [1]. Additionally, during the dynamic testing phase, the number of instructions traversed by Intel Pin [66] recorded in this paper exceeds 10,000 times the number of instructions in each test sample. Consequently, there is substantial overlap of instructions or basic blocks across different sampled paths,

further ensuring the diversity of sampled paths. In the experiments, the reliability differences between analysis methods are determined based on the proportion of reference dependencies failed to be identified by each method.

5.2.2 Accuracy testing

To achieve a better accuracy testing, we design a hierarchical data dependency validation method based on the idea of RENN [33] and DEEPVSA [34] assisting traditional analysis methods, which includes an indirect validation phase and a direct validation phase. The indirect validation phase uses existing lightweight methods for indirect data dependency false positive validation, mainly including the method used by RENN based on instruction memory regions and the method used by BDA based on operand data types. This phase can only accurately identify some false positives among the non-reference dependencies generated by binary dependence analysis methods but cannot validate whether other non-reference dependencies are false positives. Therefore, in the direct validation phase, we use a symbolic execution method based on angr to validate the remaining non-reference dependencies and directly determines whether these dependencies are real data dependencies based on the symbolic execution results.

Since memory regions and data types can be quickly obtained during the compilation of the test samples, using this information to validate data dependency false positives introduces minimal computational overhead. As a result, the indirect validation phase is a lightweight phase in the accuracy testing. In contrast, the direct validation phase, which uses symbolic execution to check each pair of data dependencies, incurs more computational overhead. To improve the computational efficiency of the direct validation phase, we slice the program based on the inter-procedural control flow graph and the basic blocks of the instruction pairs before performing symbolic execution validation on any data dependency instruction pairs. angr uses directed symbolic execution to explore execution paths along the slice.

During the directed symbolic execution process, angr prioritizes traversing the path segments with the highest proportion of unvisited edges and selects the shortest path segments. Since existing analysis methods are not path-sensitive, the symbolic execution validation method used in we follow the may-analysis standard in binary alias analysis. Specifically, if any feasible path between a pair of data dependency instructions makes the data dependency hold, it is considered a real data dependency. Additionally, a timeout mechanism is set for composite program slice coverage in the direct validation phase. After the number of feasible paths traversed exceeds one and the edge coverage in the slice exceeds 80%, if no execution path is found that makes the data dependency hold, the dependency is considered a false positive. On the other hand, if the data dependency validation is not completed within the specified time threshold, the data dependency is marked as unknown.

In summary, the evaluation method used in we categorize the data dependencies discovered by a binary data dependence analysis method into four types during effectiveness validation: discovered reference dependencies, validated real dependencies, validated false positive dependencies, and unknown dependencies. To reflect the accuracy of the analysis method, we will list the proportion of validated false positive dependencies and unknown dependencies in all discovered data dependencies. Meanwhile, the paper will also provide the number of missing reference dependencies in the experimental data and their proportion in all reference dependencies of the corresponding test samples, enabling comparison of the soundness between different analysis methods.

5.3 Evaluation Setup

Based on the binary data dependence analysis evaluation method designed above, we conduct soundness and accuracy testing on existing analysis methods using the SPEC CINT 2000 benchmark. Since some existing analysis methods do not have open-source code or accessible tools, and some methods require integration with other tools, we only examine three available tools: BDA, NeuDep, and DueForce. Since BDA's analysis results depend on the path sampling time settings, we delve into the path sampling time settings in the BDA paper. First, we adjust the compilation settings to generate SPEC CINT 2000 test samples that contain the same number of reference dependencies as in the BDA paper. Then, it compares the instruction count for each sample with the corresponding path sampling time to obtain a ratio of sampling time to instruction count, ranging from 0.01 to 0.06. Considering that the open-source code implementation for DueForce also aims to improve the soundness of the analysis results, we use the highest ratio of sampling time to instruction count from these methods, i.e., it maximizes the path sampling time within a reasonable range, thereby prioritizing the soundness of BDA's analysis results as well.

5.4 Evaluation Results & Analysis

5.4.1 Soundness testing

The soundness testing results from Table 2 reveal substantial differences among the three approaches, underscoring their unique mechanisms and limitations. BDA demonstrates exceptional correctness across many programs (e.g., 99.26% in *164.gzip* and 98.48% in *300.twolf*) due to its path sampling and abstract interpretation, which aggregate insights from diverse execution paths. However, its performance varies dramatically: while some programs achieve near-perfect precision, others (e.g., *253.perlbmk*) exhibit drastically lower accuracy (17.63%), suggesting challenges in handling highly branchy or path-sensitive code. The high number of detected dependencies (*#Found*) in BDA hints at potential overestimation or false positives, particularly in programs with complex control flow.

DueForce, relying on single-path execution per basic block path scheme, shows significantly lower accuracy (e.g., 60.60% in *175.vpr* and 75.89% in *256.bzip2*) and inconsistent results. Its simplistic selection of paths often misses

critical execution variations, leading to under-detection and higher error rates. The inability to analyze *176.gcc* within reasonable time frames further highlights scalability limitations, making it impractical for large or complex binaries.

NeuDep balances accuracy and efficiency, achieving moderate-to-high correctness (71.03 – 89.01%) across all programs. Its hybrid model—using a transformer for path tracking and an MLP for dependency prediction—avoids BDA's overestimation and DueForce's incomplete coverage. Nevertheless, performance fluctuations (e.g., 78.00% in *254.gap* vs. 83.00% in *255.vortex*) suggest sensitivity to code structure and training biases. While effective in many cases, its reliance on pre-trained networks may not generalize well to novel binary patterns.

In summary, BDA excels in capturing path-specific dependencies but risks overestimation, DueForce lacks practicality for large programs due to limited coverage, and NeuDep offers a promising trade-off through learned modeling. These results emphasize the inherent challenges of achieving robust memory dependence analysis across heterogeneous workloads, particularly balancing precision, scalability, and generalization.

Table 2 Soundness Testing on Memory Dependence Analysis

Program	#Refer	BDA			DueForce			NeuDep		
		#Found	#Correct	#Missing	#Found	#Correct	#Missing	#Found	#Correct	#Missing
164.gzip	3,648	2,034,326	3,621 (99.26%)	27 (0.74%)	3,347	2,506 (68.70%)	1,142 (31.30%)	3,962	2,591 (71.03%)	1,057 (28.97%)
175.vpr	13,962	1,016,459	13,274 (95.07%)	688 (4.93%)	12,211	8,461 (60.60%)	5,501 (39.40%)	15,013	12,427 (89.01%)	1,535 (10.99%)
176.gcc*	324,884	574,925,021	252,960 (77.86%)	71,924 (22.14%)	-	-	-	446,135	240,415 (74.00%)	84,469 (26.00%)
181.mcf	2,053	45,059	2,050 (99.85%)	3 (0.15%)	2,056	1,548 (75.40%)	505 (24.60%)	2,195	1,561 (76.04%)	492 (23.96%)
186.crafty	31,631	909,355	17,057 (53.92%)	14,574 (46.08%)	14,858	12,136 (38.37%)	19,495 (61.63%)	36,722	23,724 (75.00%)	7,907 (25.00%)
197.parser	16,575	43,566,563	16,549 (99.84%)	26 (0.16%)	10,714	7,471 (45.07%)	9,104 (54.93%)	19,914	13,758 (83.00%)	2,817 (17.00%)
253.perlbmk	61,939	3,656,833	10,918 (17.63%)	51,021 (82.37%)	10,714	7,471 (20.17%)	9,104 (79.83%)	70,036	49,552 (80.00%)	12,387 (20.00%)
254.gap	42,276	403,229	7,106 (16.81%)	35,170 (83.19%)	2,448	1,338 (3.16%)	40,938 (96.84%)	47,962	32,976 (78.00%)	9,300 (22.00%)
255.vortex	42,523	4,106,937	33,113 (77.87%)	9,410 (22.13%)	38,839	15,823 (37.21%)	26,700 (62.79%)	49,301	34,869 (82.00%)	7,654 (18.00%)
256.bzip2	4,306	29,779	4,306 (100.00%)	0 (0.00%)	3,848	3,268 (75.89%)	1,038 (24.11%)	5,017	3,402 (79.01%)	904 (20.99%)
300.twolf	17,876	17,115,546	17,604 (98.48%)	272 (1.52%)	26,747	11,572 (64.73%)	6,304 (35.27%)	19,173	13,050 (73.00%)	4,826 (27.00%)

Note: *DueForce fails to accomplish the analysis on 176.gcc in 24 hours, which we ignore its analysis result.

Table 3 Accuracy Testing on Memory Dependence Analysis

Program	#Refer	BDA			DueForce			NeuDep		
		#Extra	#FP	#Unknown	#Extra	#FP	#Unknown	#Extra	#FP	#Unknown
164.gzip	3,648	2,030,705	1,402,029 (69.04%)	504,652 (24.85%)	841	7 (0.83%)	16 (1.90%)	1,371	1,179 (86.00%)	24 (1.75%)
175.vpr	13,962	1,003,185	712,720 (71.05%)	1,821 (0.18%)	3,750	207 (5.52%)	248 (6.61%)	2,586	2,586 (100.00%)	0 (0.00%)
176.gcc*	324,884	574M	484M (84.30%)	27M (4.77%)	-	-	-	205,720	199,548 (97.00%)	2,935 (1.43%)
181.mcf	2,053	43,009	37,786 (87.86%)	281 (0.65%)	508	89 (17.52%)	8 (1.57%)	634	557 (87.85%)	32 (5.05%)
186.crafty	31,631	892,298	516,539 (57.89%)	139,479 (15.63%)	2,722	72 (2.65%)	204 (7.49%)	12,998	11,438 (88.00%)	1,191 (9.16%)
197.parser	16,575	43,550,014	24,978,455 (57.36%)	16,885,457 (38.77%)	3,243	864 (26.64%)	118 (3.64%)	6,156	5,355 (86.99%)	561 (9.11%)
253.perlbmk	61,939	3,645,915	3,498,716 (95.96%)	52,519 (1.44%)	11,071	983 (8.88%)	1,143 (10.32%)	20,484	20,074 (98.00%)	365 (1.78%)

254.gap	42,276	396,123	315,764 (79.71%)	43,486 (10.98%)	1,110	19 (1.71%)	98 (8.83%)	14,986	14,836 (99.00%)	20 (0.13%)
255.vortex	42,523	4,073,824	1,943,481 (47.71%)	2,034,891 (49.95%)	38,839	15,823 (40.74%)	2,301 (5.92%)	14,432	12,988 (89.99%)	765 (5.30%)
256.bzip2	4,306	25,473	14,041 (55.12%)	764 (3.00%)	841	7 (0.83%)	16 (1.90%)	1,615	1,501 (92.94%)	75 (4.64%)
300.twolf	17,876	17,097,942	15,080,358 (88.20%)	1,546,820 (9.05%)	15,175	1,733 (11.42%)	268 (1.77%)	6,123	6,000 (97.99%)	77 (1.26%)

Note: *DueForce fails to accomplish the analysis on 176.gcc in 24 hours, which we ignore its analysis result.

5.4.2 Accuracy testing

The accuracy testing results demonstrated in Table 3 reveal critical insights into the precision and reliability of memory dependence analysis across three approaches. BDA demonstrates significant variability in its performance, often producing high false positives (*#FP*) while struggling to resolve ambiguities. For instance, it reports 84.30% false positives in *176.gcc* (27 million out of 324 million dependencies) but achieves near-perfect resolution in *164.gzip* (only 24.85% unknown). This inconsistency suggests that BDA's path sampling strategy may overgeneralize dependencies in complex programs like *176.gcc*, yet effectively captures clear patterns in simpler ones like *164.gzip*.

DueForce, by contrast, exhibits extremely low false positives (e.g., 0.83% in *164.gzip*) but struggles with uncertainty, leaving a large fraction of dependencies unresolved (*#Unknown*). For example, in *175.vpr*, it resolves all dependencies with perfect accuracy but fails to address any in *254.gap* due to its conservative single-path execution model. This trade-off between precision and completeness limits its practicality, as unresolved dependencies (e.g., 8.83% in *254.gap*) may represent critical misses.

NeuDep achieves a balanced accuracy profile, resolving most dependencies definitively (e.g., 100% resolution in *175.vpr* and *254.gap*) while maintaining low false positives (≤9.16% in most cases). Its hybrid model leverages the transformer to track path segments and the MLP to predict dependencies, minimizing both overestimation and under-detection. Notably, it handles ambiguous cases more effectively than BDA (e.g., 1.75% *#Unknown* in *164.gzip* vs. 24.85% for BDA) but still faces challenges in highly dynamic code like *186.crafty* (1.19% *#Unknown*).

In summary, BDA prioritizes comprehensive coverage at the cost of precision, DueForce balances low false positives with incomplete resolution, and NeuDep strikes an optimal middle ground through learned modeling. These results highlight the inherent trade-offs between accuracy, completeness, and scalability in memory dependence analysis, underscoring the need for context-aware strategies tailored to program characteristics.

5.4.3 Analysis overheads

The computational overhead results shown in Table 4 reveal significant disparities in efficiency, rooted in the three approaches' fundamental mechanisms. BDA incurs the highest runtime costs due to its exhaustive enumeration of execution paths combined with static abstract interpretation. By sampling and analyzing vast numbers of paths (even in the hundreds of thousands for large programs like *176.gcc*), it ensures comprehensive coverage but becomes impractical for real-world use. This static analysis nature further amplifies its resource demands, as it must process every path individually without dynamic optimizations.

Table 4 Analysis Computational Overhead on Memory Dependence Analysis

Program	#LOC	BDA(s)	DueForce(s)	NeuDep(s)**
164.gzip	10,977	5,724	258	0.84
175.vpr	48,545	24,480	1,358	2.69
176.gcc	548,231	36,216	-*	42.54
181.mcf	4,779	2,988	46	0.21
186.crafty	42,084	26,604	86,408	3.36
197.parser	36,758	5,724	1,624	3.12
253.perlbmk	133,755	40,860	49,115	12.48
254.gap	133,246	20,412	6,713	7.55
255.vortex	150,589	42,300	67,359	12.59
256.bzip2	10,389	8,352	258	0.34
300.twolf	90,639	42,048	2,973	4.30

Note: *DueForce fails to accomplish the analysis on 176.gcc in 24 hours, which we ignore its analysis result.
**The analysis computational overhead of NeuDep refers to its inference time

DueForce, leveraging dynamic analysis, demonstrates moderate efficiency in small programs (e.g., completing *181.mcf* in minutes). Its single-path execution per basic block path scheme avoids redundant computations, making it faster than BDA for simple code. However, this approach struggles with large or highly branchy programs (e.g., *176.gcc*, which remains unresolved after 24 hours), as dynamic path exploration still risks path explosion in complex control flows. The trade-off between speed and completeness limits its applicability to medium-sized codebases.

NeuDep achieves the best performance by focusing solely on inference during evaluation, bypassing exhaustive path enumeration. Its hybrid model—pre-trained transformer for path tracking and MLP for dependency prediction—enables efficient generalization without the need for exhaustive static or dynamic analysis. This design allows it to handle small programs in sub-second intervals (e.g., *181.mcf*) and larger binaries (up to 12.59 seconds for *255.vortex*). Although its performance correlates with program size, its reliance on learned patterns minimizes overhead compared to BDA and DueForce.

In summary, BDA prioritizes accuracy through exhaustive path analysis but sacrifices scalability; DueForce balances speed and simplicity for small programs but falters in complex scenarios due to path explosion; NeuDep delivers the best trade-off by leveraging inference-based learning, offering efficient analysis across diverse program sizes and structures.

6 CHALLENGES IN BINARY DEPENDENCE ANALYSIS RESEARCH

Despite significant progress in binary dependence analysis in recent years, there are still many challenges in addressing practical issues such as complex control flow, compiler optimizations, and code obfuscation. In this section, we will delve into these challenges, including path explosion, indirect branch analysis, and over-approximation in abstract interpretation. By systematically examining the current challenges, we aim to highlight the critical technical bottlenecks that need to be overcome in binary dependence analysis.

6.1 Challenges of Path Explosion in Binary Dependence Analysis

Based on the comparison of the performance of existing analysis methods discussed earlier, the computational time overhead of DueForce, which is based on enforced execution, increases rapidly as the scale of the test sample grows. Particularly, when the number of instructions in the sample exceeds 150,000, the analysis time for DueForce grows even more sharply, exhibiting an exponential growth trend relative to the number of instructions. The basic principle of DueForce requires traversing different execution paths to ensure code coverage, and the dynamic testing method used in fuzzing incurs minimal time cost for executing a single path. Therefore, the exponential increase in computational time overhead reflects the exponential growth of execution paths in dynamic analysis. In binary dependence analysis, this phenomenon, where the number of execution paths grows exponentially with the number of program instructions, is known as path explosion.

The root cause of the path explosion problem lies in conditional branches within the program. Continuous conditional branch operations turn the program's control flow structure into a tree rooted at the program entry point. In simple acyclic control flow structures, assuming no indirect branch instructions exist, the control flow graph will resemble a binary tree. As the binary tree structure with n nodes gradually approaches a full binary tree, the number of execution paths will tend to 2^n, exhibiting an exponential growth trend relative to the number of basic blocks. If the number of instructions in each basic block is finite, this can be seen as the number of execution paths in a binary program growing exponentially relative to its instruction count.

On the other hand, the loop structures and function calls commonly found in real-world programs further exacerbate path explosion by enabling the reuse of certain program statements. For the same function, non-recursive function calls effectively insert the path segments contained in that function at each call point. Every time a function with multiple path segments is inserted, the number of execution paths at the call point doubles based on the number of path segments in the callee. The impact on path growth is even more pronounced in circular structures like loops and recursion. Since these circular structures often cannot be fully determined through static analysis in terms of their iteration counts, even without considering the computational time overhead, conventional static analysis methods cannot ensure the soundness of the analysis.

6.2 Challenges from Indirect Branches in Binary Dependence Analysis

In binary programs, jump instructions and function call instructions that use non-immediate operands are referred to as indirect jumps and indirect calls, respectively, collectively known as indirect branches. Among the analysis methods selected for comparison, BDA and DueForce are the most affected by indirect branches. The path sampling in BDA and the basic block successor state closure computation in DueForce both rely on the pre-constructed iCFG. Therefore, binary programs containing tail calls result in severe iCFG omissions, leading to a significant number of missed memory dependence analyses.

Since the branch targets of indirect branches cannot be directly determined from the semantics of the indirect branch instruction itself, existing static analysis methods generally infer the targets by using techniques such as jump tables and program slicing after constructing other parts of the control flow graph. Symbolic execution methods, like dynamic

analysis methods, directly identify the value range of operands in indirect branch instructions through value propagation. Indirect branch instructions can have different branch targets depending on preceding execution path segments, and existing analysis methods, due to the lack of path sensitivity, allow data flow that reaches the indirect branch instruction along any execution path to potentially flow to all indirect branch targets, leading to significant false positives in data dependence analysis.

Additionally, not all indirect branch targets can be inferred through the aforementioned techniques, as some operands of indirect branch instructions may depend on runtime register values, memory contents, or external inputs. This uncertainty causes several issues: First, static analysis tools struggle to exhaustively enumerate all potential jump targets, which may result in the omission of critical data flow paths, leading to incomplete dependence analysis. Second, the target of an indirect branch may be driven by complex calculations or external data, making the association between data sources and jump behaviors unclear, thus breaking the continuity of data dependencies. Furthermore, malicious code often utilizes indirect branches to obfuscate control flow (e.g., through self-modifying code or polymorphic jumps), further interfering with analysis tools' ability to reconstruct true data dependencies. Even with the assistance of dynamic analysis or symbolic execution, limitations such as path explosion, environmental dependencies, or insufficient condition coverage may still arise. Therefore, the existence of indirect branches makes it especially difficult to accurately construct inter-procedural and inter-module data dependencies, directly impacting the soundness of tasks such as vulnerability discovery, code optimization, and security verification.

6.3 Challenges of Over-approximation from Abstract Interpretation

Facing the above two challenges, an effective solution in static analysis methods is to use abstract interpretation. This method uses sets to represent the state of each operand or variable before and after a program point (i.e., abstract domains) and treats each program point as an abstract domain transformation function. When analyzing a program, abstract interpretation requires calculating the fixed point of the input and output state sets of each program point as a prerequisite for further analysis of the program. To improve analysis efficiency, the abstract interpretation methods used in existing approaches require that all input states be consolidated as much as possible before analyzing a program point, so that the analysis of the program point can be completed in one transformation. Specifically, in the analysis of a binary program, abstract interpretation in methods like VSA, BPA, and BinPointer typically involves three steps for traversing any instruction:

1. **Input State Check**: Check whether all predecessor instructions have been analyzed. Only after obtaining the output states of all predecessor instructions can the current instruction behavior be analyzed.
2. **Input State Normalization**: Merge the output state sets of all predecessor instructions and widen this union into a form suitable for the transformation function, such as the RIC used in VSA.
3. **State Transformation**: Use the widened state set as input, complete the state transformation within a limited time, and obtain the output state.

This path-insensitive design results in the computational complexity of abstract interpretation for analyzing acyclic control flow graphs being close to the number of nodes in the control flow graph. Since the state transformation does not distinguish between states from different execution paths and, before transformation, the input state set union is widened, the output state set after transformation will necessarily be a superset of the actual output state set for that program point. Therefore, abstract interpretation can ensure the soundness of its analysis results but cannot guarantee accuracy. When analyzing programs with loops or recursive functions, directly using abstract interpretation in programs where the number of loop iterations or recursion is indeterminable will lead to the inference of loop-related operand states as $\top$, resulting in significant false positives. According to the classification of sensitivity in different analysis methods, using abstract interpretation alone for binary dependence analysis can only ensure flow sensitivity.

Similarly, abstract interpretation can also be used to address indirect branch analysis to some extent. Compared to other static analysis methods and symbolic execution, abstract interpretation can infer the values of indirect branch operands that depend on runtime register values, memory contents, or external inputs based on path constraints. Specifically, abstract interpretation will verify the value range of the non-numeric part of the indirect branch instruction operand's value expression according to the path feasibility model constructed by path conditions. Based on this range, the indirect branch instruction operand is converted into a set of numeric values, representing the set of branch targets for the indirect branch. However, the path-insensitive nature of abstract interpretation leads to potential loss of key path constraints during path merging, which in turn causes over-approximation when inferring indirect branch targets, resulting in false positives for indirect branch targets. This leads to a significant number of false positives in subsequent data dependence analysis along erroneous indirect branch control flow edges, and analysis operations performed along these non-existent execution paths are redundant operations that waste computational resources.

7 FUTURE RESEARCH DIRECTION IN BINARY DEPENDENCE ANALYSIS

Based on the analysis of the aforementioned challenges, we will explore potential future research directions in the field of binary dependence analysis. According to the analysis of the current challenges and their underlying causes, the key to simultaneously improving both accuracy and efficiency in binary dependence analysis while ensuring soundness is to explore feasible methods that can guarantee path sensitivity. In this section, we will first introduce the design of data dependence analysis methods in high-level languages, followed by a discussion on applying analysis strategies from

other fields to binary dependence analysis. It will further explore research directions for achieving path-sensitive binary dependence analysis, providing innovative ideas and potential technical pathways for future research.

7.1 Sparse Value-flow analysis

Compared to assembly language or binary code, high-level languages like C/C++ offer a richer set of semantic expressions, such as built-in functions, syntactic sugar, and variable naming conventions. This rich semantics can simplify the control flow of code by using relatively complex statement functions, and it can explicitly express the scope of variables through variable naming or operator overloading. As a result, existing research on source code data dependence analysis has integrated sparse designs to achieve efficient path-sensitive analysis methods, which are known as Sparse Value-flow analysis (SVFA) in the field of source code analysis [84].

7.1.1 Semi-sparse value-flow analysis

The earliest path-sensitive sparse analysis method implemented on C language was IPSSA [46], which aimed to achieve a pointer alias analysis method that ensures both path sensitivity and context sensitivity while being scalable. IPSSA uses an extended SSA form for precise modeling of global and local variables. It also employs different expression methods to implement path and context-sensitive alias tracking only for certain hot spot local variables, laying the foundation for sparse analysis frameworks. Later, Hackett and Aiken [85] transformed arithmetic constraints into SAT constraints (Boolean satisfiability problems) in alias analysis, using SAT solvers to simplify the feasibility determination of function-level path segments. Similarly, Cherem, et al. [86] converted path constraints into SAT constraints when using value-flow analysis to detect memory leaks, eliminating false positives in the data dependency graph caused by using path-insensitive methods to construct the detection paths.

These early analyses, which differentiated between variables of varying importance and used SSA-based variable renaming, implemented path-sensitive sparse value-flow analysis for only a small subset of variables. Since they could only perform sparse analysis on certain variables in the program, these early methods are collectively referred to as semi-sparse value-flow analysis. Building upon these early semi-sparse value-flow analysis methods, Hardekopf and Lin [87] first introduced a semi-sparse analysis method that extended the range of sparse analysis. This method performs sparse analysis only on non-pointer dereferencing variables (top-level variables) and applies traditional abstract interpretation and fixed-point computation to pointer dereferencing variables. Since most variables in C language programs are top-level variables, this method effectively broadened the scope of sparse analysis within a program.

7.1.2 Full-sparse value-flow analysis

Building on this foundation, Hardekopf and Lin [88] extended pointer alias analysis to a fully sparse design, SFS, using a staged approach. This method first performs a flow- and context-insensitive but sound Andersen analysis, which further guides the sparse analysis of indirect reference variables. Around the same time, Yu, et al. [47] proposed LevPA, a method similar to SFS, aimed at achieving fully sparse pointer alias analysis while ensuring flow sensitivity and context sensitivity. LevPA links pointers to dereferenced variables based on their reference and dereference chains. Based on the connections, LevPA constructs an extended SSA form expression for all variables based on this linkage. Finally, it combines the pointer reference chain, dereference chain, and the SSA form of variables to achieve fully sparse analysis. Because both SFS and LevPA implement fully sparse analysis, they can perform pointer alias analysis with guaranteed flow sensitivity and context sensitivity on programs of up to millions of lines of code.

To further improve the accuracy of fully sparse analysis methods, Sui, et al. [50] proposed the first path-sensitive fully sparse pointer analysis method, SPAS, based on LevPA. By combining the SSA transformation method used in LevPA with the path representation method using Binary Decision Diagrams (BDD) [89], SPAS integrates path feasibility conditions containing context information with pointer operations. During pointer alias analysis, SPAS propagates the path feasibility conditions along the execution paths and promptly eliminates infeasible paths to avoid redundant operations. Based on SPAS, Sui and Xue [84] further proposed the LLVM-based SVF analysis framework. SVF combines pointer alias analysis with Value Flow Graph (VFG) construction, using a path-sensitive sparse fixed-point computation method to generate an accurate VFG for a program, which is then used to guide various subsequent software analysis tasks [90,91].

7.1.3 Modular Approaches for Path-Sensitive Sparse Value-flow analysis

Although SPAS and SVF implement path-sensitive fully sparse analysis, the use of SAT solvers to validate path feasibility still results in significant computational overhead, limiting their ability to handle code with more than 100,000 lines. To address this issue, Shi, et al. [51] proposed a path-sensitive SVFA method, Pinpoint, based on a holistic design. This method solves the "pointer trap" problem commonly encountered in the layered design of SVF, where high-precision pointer analysis is difficult to scale, while low-precision pointer analysis negatively impacts result accuracy. Pinpoint isolates intra- and inter-procedural analysis processes, enabling efficient on-demand inter-procedural path-sensitive analysis while maintaining value-flow analysis isolation. Additionally, through function summaries, reuse, and path condition concatenation, Pinpoint further improves its scalability, enabling it to analyze programs up to a million lines of code.

Building on Pinpoint, Shi, et al. [53] introduced the SVFA method Catapult, designed to enhance scalability. This method focuses on the value flow properties it defines and reduces redundant constraint solving operations by leveraging the synergy between different value flow properties. Catapult also optimizes path feasibility verification, pruning redundant graph traversals during the value-flow analysis process. Like Pinpoint, the modular analysis

approach of Catapult further enhances scalability. Moreover, by improving the reusability of path feasibility validation results obtained using SMT solvers [92], Shi, et al. [93] proposed Fusion, which integrates sparse analysis with SMT solving. This method takes advantage of the modular nature of program dependence graphs to avoid the explicit generation and caching of path conditions, significantly reducing computational complexity through optimized solving processes and further improving the scalability of path-sensitive SVFA.
The recently proposed SVFA method, Falcon [94], builds upon Fusion and reintroduces the on-demand path feasibility validation design. Through a two-phase analysis design, Falcon avoids redundant path feasibility checks. First, during the VFG construction phase, Fusion performs semi-path-sensitive analysis, identifying simple infeasible path conditions through a linear-time semi-decision process, merging redundant paths to reduce computational complexity. In the query application phase, Falcon uses Fusion's method to perform path feasibility validation on demand based on the query target, thus avoiding redundant calculations and further enhancing the scalability of the analysis method.

7.2 Exploring Path-Sensitive Binary Dependence Analysis Methods

Based on the analysis of the progression of SVFA methods from low scalability semi-sparse analysis to high scalability fully sparse analysis, we find that the analysis strategies applied in high-level language analysis can, to some extent, be applied to binary data dependence analysis. Therefore, binary data dependence analysis has the potential to achieve both scalability and path sensitivity in its analysis.

7.2.1 Hierarchical SSA format

In the process of achieving fully sparse analysis, the hierarchical SSA formed variable representation method based on pointer reference and dereference relationships is a solution chosen by both SFS and LevPA. In binary data dependence analysis, hierarchical SSA form expressions similarly have the potential to achieve path-sensitive representations for operands. In this context, for register operands, only their values need to be converted to a hierarchical SSA form, while for memory operands, their address representations also need to be converted.
Suppose there is a path-sensitive binary data dependence analysis method that can analyze along different execution paths and validate path feasibility, while also converting all operands along the execution path to SSA form. Considering that all memory operands in binary programs can be viewed as pointer variables and dereferenced pointers, the hierarchical SSA form preserves the characteristic of maintaining all levels of pointer reference and dereference relationships, which means the resulting operand representations are also path-sensitive. Based on the strong update characteristic of SSA form expressions, this path-sensitive binary data dependence analysis can directly identify data dependencies, like dynamic analysis methods, by matching operand address expressions.

7.2.2 Modular analysis and reuse

Due to the modular nature of the structure and logic of various programs, modular analysis has naturally become a commonly used optimization approach in the field of software analysis. In the staged path-sensitive SVFA, the modular approach has been widely applied and has proven effective in optimizing path-sensitive SVFA. Similarly, binary programs are composed of various modules, including functions and basic blocks. Although each function and basic block in a binary program may be accessed by multiple execution paths, their internal influence on external data flow follows the same pattern, and the interaction with external operands at different call points is also consistent. Therefore, binary data dependence analysis can also leverage modular analysis and reuse the analysis results to achieve efficient analysis.
In the process of implementing path-sensitive binary data dependence analysis, modular analysis and reuse significantly simplify the analysis process and enhance scalability. On one hand, since external inputs to functions and basic blocks can vary depending on the call points, there's no need to consider the external execution paths when determining internal data dependencies. This allows for a more efficient validation process by reducing the number of feasible paths that need to be checked. On the other hand, operands within functions or basic blocks can be mapped along their internal data dependency relationships to entry and exit points. By constructing mappings between entry and exit operands that include path feasibility constraints, these mappings can be reused to efficiently implement path-sensitive analysis across functions and basic blocks, minimizing redundant calculations and optimizing computational resources.

7.2.3 Information share between analysis processes

Based on the research of Catapult and Fusion, many different value-flow analysis processes share the same execution paths. Therefore, redundant analysis operations can be eliminated by sharing the feasibility validation results of execution paths. Additionally, Catapult further shares the results of value-flow analysis between different analysis processes. With the optimized analysis plan scheduling and the ability to reuse modular analysis results, Catapult can accelerate the subsequent analysis processes for data dependencies or control dependencies of those analysis results based on the attributes of previous analysis outcomes.
Similarly, in path-sensitive binary data dependence analysis, the execution paths involved in the data dependencies of operands within the same instruction or basic block often intersect. Clearly, these intersecting path segments share the same path feasibility. Therefore, by directly sharing the feasibility validation results for these path segments across different analysis processes and reasonably combining them with the unique analysis results for each path segment, better analysis scalability can be achieved, avoiding redundant validation of path feasibility for the same path segments.

7.2.4 Optimization of path feasibility checking

In addition to the analysis method design strategies mentioned above, the optimization of the process for path feasibility validation using SMT solvers [95] is also crucial for path-sensitive binary data dependence analysis. In binary programs,

the path feasibility constraints consist of a set of conditional branch constraints along an execution path, and it is generally assumed that the size of this set increases with the length of the execution path. However, due to the modular nature of programs, not all constraints in the feasible constraint set of an execution path are directly or indirectly related to each other. Given the solving approach used by SMT solvers to validate model feasibility, dividing a large set of constraints into smaller, related sets and validating their feasibility separately using the SMT solver will result in lower computational overhead than validating all constraints simultaneously.

8 CONCLUSION

We systematically review the research progress, technical challenges, and future directions of binary program dependence analysis methods. First, the paper explains the core concepts of data and control dependence analysis and their importance in fields such as vulnerability discovery and software hardening, highlighting the advantages of static analysis in path coverage and soundness. By comparing the limitations of dynamic analysis (such as fuzzing), it emphasizes the necessity of combining static analysis with techniques like symbolic execution and abstract interpretation in handling complex control flow.

The paper provides a detailed review of mainstream analysis methods: VSA enhances efficiency through abstract memory models and dynamic control flow supplementation, but faces false positives with indirect operands; path sampling-based BDA and DueForce reduce over-approximation with unbiased path sampling and posterior analysis, though insufficient path diversity may affect coverage; variable block memory model-based BPA and BinPointer improve precision through fine-grained partitioning, but false positives remain; deep learning-based methods (such as RENN and NeuDep) assist traditional analysis by learning instruction features, but they rely on dynamic data and cannot directly verify dependence relationships.

Furthermore, we propose a systematic evaluation framework using GNU Coreutils and SPEC CINT 2000 as test sets, comparing existing methods in terms of soundness, accuracy, and performance. The experiments show that BDA achieves a good balance between soundness and efficiency, while deep learning methods exhibit higher false negative rates. The paper also identifies current challenges, including path explosion, the complexity of indirect branch analysis, and the over-approximation issue in abstract interpretation.

Finally, the paper envisions future research directions, proposing the adoption of SVFA from high-level languages to achieve path-sensitive binary dependence analysis, and exploring strategies such as modular analysis and optimization of path feasibility validation to balance scalability and accuracy.

COMPETING INTERESTS

The authors have no relevant financial or non-financial interests to disclose.

REFERENCES

[1] Zhang, Z, You, W, Tao, G, et al. 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, 2019; 3(OOPSLA): 1-31. DOI: 10.1145/3360563.

[2] Pei, K, She, D, Wang, M, et al. NeuDep: neural binary memory dependence analysis. in ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022. ACM. DOI: 10.1145/3540250.3549147.

[3] He, D, Xie, D, Wang, Y, et al. Define-Use Guided Path Exploration for Better Forced Execution. in ISSTA '24: 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024. ACM. DOI: 10.1145/3650212.3652128.

[4] Gui, B, Song, W, Huang, J. UAFSan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. in ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021. ACM. DOI: 10.1145/3460319.3464835.

[5] Cheng, K, Zheng, Y, Liu, T, et al. Detecting Vulnerabilities in Linux-Based Embedded Firmware with SSE-Based On-Demand Alias Analysis. in ISSTA '23: 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023. ACM. DOI: 10.1145/3597926.3598062.

[6] Zhang, M, Sekar, R. Control Flow Integrity for COTS Binaries. USENIX Association. 2013.

[7] Van Der Veen, V, Goktas, E, Contag, M, et al. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. in 2016 IEEE Symposium on Security and Privacy (SP). 2016. IEEE. DOI: 10.1109/SP.2016.60.

[8] Gu, Y, Zhao, Q, Zhang, Y, et al. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. in CODASPY '17: Seventh ACM Conference on Data and Application Security and Privacy. 2017. ACM. DOI: 10.1145/3029806.3029830.

[9] Yan, J, Yan, G, Jin, D. Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network. in 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2019. IEEE. DOI: 10.1109/DSN.2019.00020.

[10] Yin, H, Song, D, Egele, M, et al. Panorama: capturing system-wide information flow for malware detection and analysis. in CCS07: 14th ACM Conference on Computer and Communications Security 2007. 2007. ACM. DOI: 10.1145/1315245.1315261.

[11] Cha, S K, Avgerinos, T, Rebert, A, et al. Unleashing Mayhem on Binary Code. in 2012 IEEE Symposium on Security and Privacy (SP) Conference dates subject to change. 2012. IEEE. DOI: 10.1109/SP.2012.31.
[12] Cozzi, E, Graziano, M. Fratantonio, Y, et al. Understanding Linux Malware. in 2018 IEEE Symposium on Security and Privacy (SP). 2018. IEEE. DOI: 10.1109/SP.2018.00054.
[13] Wu, W, Chen, Y, Xing, X, et al. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. USENIX Association. 2019.
[14] Spensky, C, Machiry, A, Burow, N, et al. Glitching Demystified: Analyzing Control-flow-based Glitching Attacks and Defenses. in 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2021. IEEE. DOI: 10.1109/DSN48987.2021.00051.
[15] Duta, V, Giuffrida, C, Bos, H, et al. PIBE: practical kernel control-flow hardening with profile-guided indirect branch elimination. in ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021. ACM. DOI: 10.1145/3445814.3446740.
[16] Chen, Y, Zhang, D, Wang, R, et al. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. in 2017 IEEE Symposium on Security and Privacy (SP). 2017. IEEE. DOI: 10.1109/SP.2017.30.
[17] MITRE. CWE Top 25 Most Dangerous Software Weaknesses. 2024. Retrieved from: https://cwe.mitre.org/top25/.
[18] Schloegel, M, Bars, N, Schiller, N, et al. SoK: Prudent Evaluation Practices for Fuzzing. in 2024 IEEE Symposium on Security and Privacy (SP). 2024. IEEE. DOI: 10.1109/SP54263.2024.00137.
[19] Kim, T E, Choi, J, Heo, K, et al. DAFL: Directed grey-box fuzzing guided by data dependency. USENIX Association. 2023.
[20] Balakrishnan, G, Reps, T. Analyzing Memory Accesses in x86 Executables, in Compiler Construction, E. Duesterwald, Editor. Springer Berlin Heidelberg: Berlin, Heidelberg. 2004, 5-23.
[21] Balakrishnan, G, Reps, T. WYSINWYX: What you see is not what you eXecute. ACM Transactions on Programming Languages and Systems, 2010, 32(6): 1-84. DOI: 10.1145/1749608.1749612.
[22] Song, D, Brumley, D, Yin, H, et al. BitBlaze: A New Approach to Computer Security via Binary Analysis, in Information Systems Security, R. Sekar and A.K. Pujari, Editors. Springer Berlin Heidelberg: Berlin, Heidelberg. 2008, 1-25.
[23] Shoshitaishvili, Y, Wang, R, Salls, C, et al. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. in 2016 IEEE Symposium on Security and Privacy (SP). 2016. IEEE. DOI: 10.1109/SP.2016.17.
[24] Cousot, P, Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. in the 4th ACM SIGACT-SIGPLAN symposium. 1977. ACM Press. DOI: 10.1145/512950.512973.
[25] Park, J, Lee, H, Ryu, S. A Survey of Parametric Static Analysis. ACM Computing Surveys, 2022, 54(7): 1-37. DOI: 10.1145/3464457.
[26] Baldoni, R, Coppa, E, D'elia, D C, et al. A Survey of Symbolic Execution Techniques. ACM Computing Surveys, 2019, 51(3): 1-39. DOI: 10.1145/3182657.
[27] Reps, T, Balakrishnan, G. Improved Memory-Access Analysis for x86 Executables, in Compiler Construction, L. Hendren, Editor. Springer Berlin Heidelberg: Berlin, Heidelberg. 2008, 16-35.
[28] Amme, W, Braun, P, Zehendner, E, et al. Data dependence analysis of assembly code. in 1998 International Conference on Parallel Architectures and Compilation Techniques. 1998. IEEE Comput. Soc. DOI: 10.1109/PACT.1998.727270.
[29] Kim, S H, Sun, C, Zeng, D, et al. Refining Indirect Call Targets at the Binary Level. in Network and Distributed System Security Symposium. 2021. Internet Society. DOI: 10.14722/ndss.2021.24386.
[30] Kim, S H, Zeng, D, Sun, C, et al. BinPointer: towards precise, sound, and scalable binary-level pointer analysis. in CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction. 2022. ACM. DOI; 10.1145/3497776.3517776.
[31] Chipounov, V, Kuznetsov, V, Candea, G. S2E: a platform for in-vivo multi-path analysis of software systems. ACM SIGARCH Computer Architecture News, 2011, 39(1): 265-278. DOI: 10.1145/1961295.1950396.
[32] Cadar, C, Dunbar, D, Engler, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. USENIX Association. 2008.
[33] Mu, D, Guo, W, Cuevas, A, et al. RENN: Efficient Reverse Execution with Neural-Network-Assisted Alias Analysis. in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2019. IEEE. DOI: 10.1109/ASE.2019.00090.
[34] Guo, W, Mu, D, Xing, X, et al. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. USENIX Association. 2019.
[35] Debray, S, Muth, R, Weippert, M. Alias analysis of executable code. in the 25th ACM SIGPLAN-SIGACT symposium. 1998. ACM Press. DOI: 10.1145/268946.268948.
[36] Aho, A V, Lam, M S, Sethi, R, et al. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc. 2006.
[37] Landi, W, Ryder, B G. Pointer-induced aliasing: a problem taxonomy. in the 18th ACM SIGPLAN-SIGACT symposium. 1991. ACM Press. DOI: 10.1145/99583.99599.
[38] Deutsch, A. Interprocedural may-alias analysis for pointers: beyond k-limiting. in PLDI94: ACM SIGPLAN Conference on Programming Language Design and Implementation. 1994. ACM. DOI: 10.1145/178243.178263.
[39] Xu, J, Mu, D, Xing, X, et al. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts.

USENIX Association. 2017.

[40] Zhu, W, Feng, Z, Zhang, Z, et al. Callee: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning. in 2023 IEEE Symposium on Security and Privacy (SP). 2023. IEEE. DOI: 10.1109/SP46215.2023.10179482.

[41] Meng, X, Miller, B P. Binary code is not easy. in ISSTA '16: International Symposium on Software Testing and Analysis. 2016. ACM. DOI: 10.1145/2931037.2931047.

[42] Meng, X, Anderson, J M, Mellor-Crummey, J, et al. Parallel binary code analysis. in PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2021. ACM. DOI: 10.1145/3437801.3441604.

[43] Xu, L, Sun, F, Su, Z. Constructing Precise Control Flow Graphs from Binaries. 2012.

[44] Nguyen, H, Priyadarshan, S, Sekar, R. Scalable, Sound, and Accurate Jump Table Analysis. in ISSTA '24: 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024. ACM. DOI: 10.1145/3650212.3680301.

[45] Reps, T, Horwitz, S, Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. in the 22nd ACM SIGPLAN-SIGACT symposium. 1995. ACM Press. DOI: 10.1145/199448.199462.

[46] Livshits, V B, Lam, M S. Tracking pointers with path and context sensitivity for bug detection in C programs. in 2003. Association for Computing Machinery. DOI: 10.1145/940071.940114.

[47] Yu, H, Xue, J, Huo, W, et al. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. in CGO '10: 8th Annual IEEE/ ACM International Symposium on Code Generation and Optimization. 2010. ACM. DOI: 10.1145/1772954.1772985.

[48] Van Der Veen, V, Andriesse, D, Göktaş, E, et al. Practical Context-Sensitive CFI. in CCS'15: The 22nd ACM Conference on Computer and Communications Security. 2015. ACM. DOI: 10.1145/2810103.2813673.

[49] Dillig, I, Dillig, T, Aiken, A. Sound, complete and scalable path-sensitive analysis. in PLDI '08: ACM SIGPLAN Conference on Programming Language Design and Implementation. 2008. ACM. 10.1145/1375581.1375615

[50] Sui, Y., Ye, S, Xue, J, et al. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA, in Programming Languages and Systems, H. Yang, Editor. Springer Berlin Heidelberg: Berlin, Heidelberg. 2011, 155-171.

[51] Shi, Q, Xiao, X, Wu, R, et al. Pinpoint: fast and precise sparse value flow analysis for million lines of code. in PLDI '18: ACM SIGPLAN Conference on Programming Language Design and Implementation. 2018. ACM. DOI: 10.1145/3192366.3192418.

[52] Li, T, Bai, J J, Sui, Y, et al. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. in ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022. ACM. DOI: 10.1145/3503222.3507770.

[53] Shi, Q, Wu, R, Fan, G, et al. Conquering the extensional scalability problem for value-flow analysis frameworks. in ICSE '20: 42nd International Conference on Software Engineering. 2020. ACM. DOI: 10.1145/3377811.3380346.

[54] Balakrishnan, G, Gruian, R, Reps, T, et al. CodeSurfer/x86—A platform for analyzing x86 executables, in Proceedings of the 14th international conference on Compiler Construction. Springer-Verlag: Edinburgh, UK. 2005, 250-254.

[55] Pesch, R H, Osier, J M. The GNU binary utilities. Free Software Foundation, 1993.

[56] Ferguson, J, Kaminsky, D. Reverse engineering code with IDA Pro. Syngress. 2008.

[57] Balakrishnan, G, Gruian, R, Reps, T, et al. CodeSurfer/x86—A platform for analyzing x86 executables, in Proceedings of the 14th international conference on Compiler Construction. Springer-Verlag: Edinburgh, UK. 2005, 250-254.

[58] Wang, S, Wang, P, Wu, D. Reassembleable Disassembling. USENIX Association. 2015.

[59] Bauman, E, Lin, Z, Hamlen, K W. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. in Network and Distributed System Security Symposium. 2018. Internet Society. DOI: 10.14722/ndss.2018.23300

[60] Miller, K, Kwon, Y, Sun, Y, et al. Probabilistic Disassembly. in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019. IEEE. DOI: 10.1109/ICSE.2019.00121.

[61] Wang, R, Shoshitaishvili, Y, Bianchi, A, et al. Ramblr: Making Reassembly Great Again. in Network and Distributed System Security Symposium. 2017. Internet Society. DOI: 10.14722/ndss.2017.23225.

[62] Alves-Foss, J, Song, J. Function boundary detection in stripped binaries. in ACSAC '19: 2019 Annual Computer Security Applications Conference. 2019. ACM. DOI: 10.1145/3359789.3359825.

[63] Kim, S, Kim, H, Cha, S K. FunProbe: Probing Functions from Binary Code through Probabilistic Analysis. in ESEC/FSE '23: 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023. ACM. DOI: 10.1145/3611643.3616366.

[64] Andriesse, D, Slowinska, A, Bos, H. Compiler-Agnostic Function Detection in Binaries. in 2017 IEEE European Symposium on Security and Privacy (EuroS&P). 2017. IEEE. DOI: 10.1109/EuroSP.2017.11.

[65] Di Federico, A, Payer, M, Agosta, G. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. in CC '17: Compiler Construction. 2017. ACM. DOI: 10.1145/3033019.3033028.

[66] Luk, C K, Cohn, R, Muth, R, et al. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 2005, 40(6): 190-200.

[67] Eom, H, Kim, D, Lim, S, et al. R2I: A Relative Readability Metric for Decompiled Code. Proceedings of the ACM on Software Engineering, 2024, 1(FSE): 383-405.

[68] Borzacchiello, L, Coppa, E, Demetrescu, C. SENinja: A symbolic execution plugin for Binary Ninja. SoftwareX, 2022, 20, 101219. DOI: https://doi.org/10.1016/j.softx.2022.101219.

[69] Pang, C, Yu, R, Chen, Y, et al. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. in 2021 IEEE Symposium on Security and Privacy (SP). 2021. IEEE. DOI: 10.1109/SP40001.2021.00012.

[70] Priyadarshan, S, Nguyen, H, Sekar, R. Accurate Disassembly of Complex Binaries Without Use of Compiler Metadata. in ASPLOS '23: 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4. 2023. ACM. DOI: 10.1145/3623278.3624766.

[71] Afianian, A, Niksefat, S, Sadeghiyan, B. et al. Malware Dynamic Analysis Evasion Techniques: A Survey. ACM Computing Surveys, 2020, 52(6): 1-28. DOI: 10.1145/3365001.

[72] Andriesse, D, Slowinska, A, Bos, H.Compiler-agnostic function detection in binaries. 2017. DOI; 10.1109/EuroSP.2017.11.

[73] Morrisett, G, Tan, G, Tassarotti, J, et al. RockSalt: better, faster, stronger SFI for the x86. SIGPLAN Not., 2012, 47(6): 395-404. DOI: 10.1145/2345156.2254111.

[74] Morrisett, G, Tan, G, Tassarotti, J, et al. RockSalt: better, faster, stronger SFI for the x86, in Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery: Beijing, China. 2012, 395-404.

[75] Pei, K, Guan, J, Williams-King, D, et al. XDA: Accurate, Robust Disassembly with Transfer Learning. in Network and Distributed System Security Symposium. 2021. Internet Society. 10.14722/ndss.2021.23112

[76] Yu, S., Y. Qu, X. Hu, et al. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. in 2022. USENIX Association.

[77] David, Y., U. Alon, and E. Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. Proceedings of the ACM on Programming Languages, 2020, 4(OOPSLA): 1-28. DOI: 10.1145/3428293.

[78] Chen, S, Lin, Z, Zhang, Y. SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting. USENIX Association. 2021.

[79] Ming, J, Xu, D, Jiang, Y, et al. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. USENIX Association. 2017.

[80] Ghaffarinia, M, Hamlen, K W. Binary Control-Flow Trimming. in CCS '19: 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019. ACM. DOI: 10.1145/3319535.3345665.

[81] Lemerre, M. SSA Translation Is an Abstract Interpretation. Proceedings of the ACM on Programming Languages, 2023, 7(POPL): 1895-1924. DOI: 10.1145/3571258.

[82] Cui, W, Ge, X, Kasikci, B, et al. REPT: Reverse debugging of failures in deployed software. USENIX Association. 2018.

[83] Corporation, S P E. SPEC CINT2000 (Integer Component of SPEC CPU2000). 2006.

[84] Sui, Y, Xue, J. SVF: interprocedural static value-flow analysis in LLVM. in CGO '16: 14th Annual IEEE/ACM International Symposium on Code Generation and Optimization. 2016. ACM. DOI: 10.1145/2892208.2892235.

[85] Hackett, B, Aiken, A. How is aliasing used in systems software? in 2006. Association for Computing Machinery. DOI: 10.1145/1181775.1181785.

[86] Cherem, S, Princehouse, L, Rugina, R. Practical memory leak detection using guarded value-flow analysis. in PLDI '07: ACM SIGPLAN Conference on Programming Language Design and Implementation. 2007. ACM. DOI: 10.1145/1250734.1250789.

[87] Hardekopf, B, Lin, C. Semi-sparse flow-sensitive pointer analysis. 2009. Association for Computing Machinery. DOI: 10.1145/1480881.1480911.

[88] Hardekopf, B, Lin, C. Flow-sensitive pointer analysis for millions of lines of code. in 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2011. IEEE. DOI: 10.1109/CGO.2011.5764696.

[89] Akers. Binary Decision Diagrams. IEEE Transactions on Computers, 1978, C-27(6): 509-516. DOI: 10.1109/TC.1978.1675141.

[90] Sui, Y, Ye, D, Xue, J. Static memory leak detection using full-sparse value-flow analysis. in ISSTA '12: International Symposium on Software Testing and Analysis. 2012. ACM. DOI: 10.1145/2338965.2336784.

[91] Sui, Y, Ye, D, Xue, J. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. IEEE Trans. Softw. Eng., 2014, 40(2): 107-122. DOI: 10.1109/tse.2014.2302311.

[92] de Moura, L, Bjrner, N. Z3: An Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg. 2008, 4963, 337-340. DOI: https://doi.org/10.1007/978-3-540-78800-3_24.

[93] Shi, Q, Yao, P, Wu, R, et al. Path-sensitive sparse analysis without path conditions. in PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021. ACM. DOI: 10.1145/3453483.3454086.

[94] Yao, P, Zhou, J, Xiao, X, et al. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. Proceedings of the ACM on Programming Languages, 2024, 8(PLDI): 567-592. DOI: 10.1145/3656400.

[95] Yao, P, Shi, Q, Huang, H, et al. Fast bit-vector satisfiability. in ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020. ACM. DOI: 10.1145/3395363.3397378.