BinDSA: Efficient, Precise Binary-Level Pointer Analysis with Context-Sensitive Heap Reconstruction

LIAN GAO, University of California at Riverside, USA
HENG YIN, University of California at Riverside, USA

Pointer analysis serves as a fundamental component in the realm of binary code reverse engineering. It can be leveraged to reconstruct a binary program's call graph and can be further applied to various security analyses. However, the absence of symbols and type information within binary code presents formidable challenges to effective pointer analysis. Existing works often apply approximations when performing pointer analysis on binary. Nevertheless, these methods tend to be inefficient and produce numerous false positive targets. In this paper, we propose BinDSA, a novel model tailored for binary pointer analysis. BinDSA prioritizes precision and efficiency over soundness. It is field- and context-sensitive, employing unification-based techniques and reconstructing a context-sensitive heap. It jointly recovers data structure and points-to relations so that precision can be further improved. In evaluation, we demonstrate that BinDSA is 5 times more efficient and notably more precise than the current state-of-the-art technique without significantly sacrificing soundness. We also apply BinDSA on CVE reachability analysis and vulnerability detection, demonstrating its effective application to security tasks.

CCS Concepts: • **Security and privacy → Software reverse engineering**; • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: Pointer Analysis, Data Structure Recovery.

ACM Reference Format:
Lian Gao and Heng Yin. 2025. BinDSA: Efficient, Precise Binary-Level Pointer Analysis with Context-Sensitive Heap Reconstruction. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA053 (July 2025), 22 pages. https://doi.org/10.1145/3728928

1 Introduction

Pointer analysis is essential in binary code reverse engineering. It is the foundation of reconstructing a complete call graph and value flow of a given binary program. Recovering points-to relations precisely and efficiently holds immense value for various downstream analyses such as binary code diffing [14, 16, 27], static taint analysis [11, 15], and bug detection [21, 22, 42].

However, pointer analysis on binary code is very challenging. When dealing with stripped binaries, source-level information such as variables and types is lost. Consequently, it becomes imperative to pre-identify the variable-like entities in order to facilitate subsequent static analysis of the binaries. Additionally, since types are missing, every register or memory location can potentially be a pointer, and the presence of type casts in binary code introduces additional complexities. Moreover, memory can be accessed indirectly through pointer arithmetics in binary code, further complicating the analysis process.

Authors' Contact Information: Lian Gao, University of California at Riverside, Riverside, USA, lgao027@ucr.edu; Heng Yin, University of California at Riverside, Riverside, USA, heng@cs.ucr.edu.

ACM 2994-970X/2025/7-ARTISSTA053
https://doi.org/10.1145/3728928

Existing works that focus on pointer analysis or indirect call target recovery indicate the challenge of simultaneously attaining precision, soundness, and efficiency. *Arity-based approaches* such as TypeArmor [41] and τCFI [30] treat all functions as potential targets and filter out false positive function targets by checking the compatibility of parameters (e.g., number of parameters) between the call sites and potential targets. They are sound and efficient but suffer low precision. *Deep-learning-based approaches* such as Callee [47] leverages deep neural networks to learn patterns concerning indirect calls to predict indirect call targets. Callee is promising in terms of precision and efficiency, but it is unsound. There also exists a *path-sampling-based approach* BDA [46] that leverages a path sampling algorithm to extract and combine the memory dependency among each path to infer the points-to relations. It is more precise but unsound since the algorithm cannot cover all paths.

Another line of work employs *program-analysis-based approaches* [7, 23, 24] that conduct sophisticated static analysis (listed in Table 1). Value-set Analysis [7] (VSA) identifies potential values stored at abstract locations (a-locs) to determine pointer targets. However, it is imprecise and inefficient due to imprecise tracking of pointer arithmetic. BPA [23] and BinPointer [24] address the limitations of VSA by employing a block memory model to avoid pointer arithmetic within the block. However, some of their design decisions compromise precision. BinPointer enhances BPA by tracking 0-based dereferences. However, its functionality remains constrained, as non-0-based dereferences revert to block memory modeling. In general, BPA and BinPointer are sound but not precise and efficient enough. More discussions can be found in §2.2.

Table 1. Comparison of Different Program-analysis-based Approaches

Algorithm	Sensitivity Context-	Sensitivity Field-	Model of Aggregation	Model of Memory	Precision	Soundness	Efficiency
VSA	✓	✗	Inclusion-based	A-loc	◔	●	◔
BPA	✗	✗	Inclusion-based	Block memory model	◐	●	◐
BinPointer	✓	✓	Inclusion-based	Block memory model	◐	●	◐
BinDSA (this paper)	✓	✓	Unification-based	A-loc w/ context-sensitive heap	◕	◕	●

Considering the difficulties in achieving precision, soundness, and efficiency simultaneously, we propose to prioritize precision and efficiency rather than soundness since complete soundness is not always required in security tasks. Precision is more crucial because it minimizes false positives and significantly reduces the workload of subsequent analyses. To this end, we propose a *soundy* [33] approach that sacrifices soundness only in limited situations (discussed in §7) to ensure high precision.

Specifically, we propose a program-analysis-based approach with several enhancements. First, we jointly recover data structure and points-to relations so that they can enhance each other. The recovery of data structures contributes to a field-sensitive pointer analysis, thereby increasing precision. Conversely, the points-to relations enable more complete type propagation during the recovery of data structures. Second, we conduct context-sensitive analysis and model the heap region to be context-sensitive (a.k.a., heap cloning) to further elevate precision. Moreover, we adopt a unification-based algorithm (a.k.a., Steensgaard's style [36]) to enhance efficiency. To mitigate the inherent imprecision associated with unification-based algorithms, we adopt a conservative strategy during the unification process. Specifically, we leverage the recovered type information as a filter to systematically eliminate conflicts. While this approach sacrifices soundness to some extent, it is necessary to ensure greater precision through conflict filtering.

To validate the feasibility of our approach, we implement a system prototype called BinDSA. Our evaluation results show that BinDSA achieves an average precision improvement of 4.9% and

14.9% over BinPointer in global and heap object access evaluations, and a 25% improvement over BPA in resolving indirect call targets, as reflected in profiling-based evaluation results. Additionally, BinDSA successfully recovers over 90% of memory accesses and indirect call targets. Furthermore, BinDSA demonstrates strong efficiency in both processing speed and memory usage. Specifically, it requires only 26 minutes and 29GB of memory to analyze `403.gcc` from the SPEC2006 benchmark suite, whereas BPA takes approximately seven hours on a machine with 350GB of RAM. To further illustrate its effectiveness, we conduct a case study on the CVE reachability problem, demonstrating BinDSA 's ability to identify unreachable vulnerable functions within software programs. Additionally, we successfully apply BinDSA to detect use-after-free and double-free vulnerabilities in the Juliet Test Suite [1].

Contributions. This paper makes the following contributions:

- We introduce BinDSA, a novel algorithm that combines field- and context-sensitive unification-based pointer analysis (with heap cloning) and the data structure type recovery to tackle the challenges of binary pointer analysis.
- We evaluate BinDSA on SPEC 2006 benchmarks and real-world programs and show that it outperforms existing state-of-the-art solutions in both precision and efficiency without sacrificing soundness significantly.
- We conduct case studies showcasing the effectiveness of BinDSA in CVE reachability analysis and vulnerability detection.

2 Background and Related Work

In this section, we first introduce the points-to analysis for source code and discuss the challenges of directly applying source code algorithms to binary code. Then we discuss the background and related work of points-to analysis at the binary level.

2.1 Points-to Analysis on Source Code

There is a vast body of literature on points-to analysis for source code. In general, several key dimensions significantly impact the performance of pointer analysis algorithms, including path-sensitivity [17, 38], flow-sensitivity [18, 19, 43, 44], field-sensitivity [8, 25, 26, 32], and (calling) context-sensitivity [20, 31, 39]. Additionally, the modeling of aggregates—encompassing Steensgaard's unification-based approach [36], Andersen's inclusion-based approach [6], and various hybrid approaches [13, 35]—plays a crucial role.

In general, increasing the sensitivity of a points-to analysis improves its precision but also significantly increases its computational cost. The choice between inclusion-based and unification-based modeling similarly affects this trade-off. Andersen's inclusion-based approach [6] maintains distinct points-to sets for different variables, preserving precision but leading to higher complexity, often cubic in the worst case. In contrast, Steensgaard's unification-based approach [36] merges points-to sets aggressively, making it highly scalable with nearly linear complexity at the cost of over-approximation.

In this context, Data Structure Analysis (DSA) [25] offers an attractive compromise. It is a unification-based points-to analysis algorithm that is field-sensitive and context-sensitive with full heap cloning through acyclic call paths. DSA has demonstrated a commendable balance between precision and efficiency. Evaluations have shown that DSA exhibits minimal precision loss compared to Andersen's algorithm, even when accounting for the loss of context sensitivity in strongly connected components (SCCs) [25].

However, directly applying the DSA algorithm (or other source code algorithms) on lifted IRs remains infeasible. A comprehensive survey conducted by Liu et al. [28] highlighted that the

absence of type information and the presence of errors in lifted IRs pose significant impediments to conducting rigorous static analysis on lifted IRs. After a thorough investigation, we summarize the challenges of applying source code algorithms to binary code as follows:

C1. No variable information. Source code pointer analysis relies on variable information, which is absent in binary code. Incorrect identification of variables can result in either a loss of precision or a violation of soundness.

C2. Lack of type information.

C2.1. Lack of distinction between pointers and values. A source code pointer analysis typically focuses on pointers rather than values. In binary code, the absence of type information makes it challenging to distinguish between pointers and values. This ambiguity arises, for example, when an addition operation involves two registers, which can represent either the addition of two numerical values or a base pointer plus an offset. It is crucial to determine which register serves as the base pointer to clarify the points-to relationship.

C2.2. Absence of field information and the necessity for pointer arithmetic. High-level IRs generated from source code will retain data structure field information, whereas binary code requires pointer arithmetic to achieve field sensitivity. However, pointer arithmetic is challenging, considering the complex data structures such as different kinds of arrays and embedded data structures.

C3. Errors in lifted IRs. A correct pointer analysis relies on a series of accurate pre-analyses. We observe that insufficient dead code elimination or incorrectly identifying additional function parameters compared to source code will lead to incorrect placement of the `phi` node. These errors can accumulate in subsequent analyses and significantly impact the precision of the pointer analysis.

C4. Scalability. The analysis of lifted binary IRs requires tracking both pointers and potential pointers, leading to a more complex graph with a greater number of nodes compared to source code compiled IRs. This severely impacts the efficiency of analysis methods.

These challenges make pointer analysis on binary-lifted IRs extremely difficult. In §3.1, we will explain how BinDSA tackles these challenges.

2.2 Points-to Analysis on Binaries

Existing works on points-to analysis of binaries also face the challenges mentioned above. Achieving precision, soundness, and scalability simultaneously poses a significant challenge in this context.

Value-set Analysis [7] (VSA) records the possible values stored at each abstract location (*a-loc*s) to determine possible pointer targets. However, as noted in BPA [23] and BDA [46], the arithmetic calculations on the strided intervals involve calculating the greatest common divisor (GCD), which can result in a lot of false positive targets. This accumulated imprecision not only complicates points-to relations but also affects execution time, making VSA impractical for large programs.

BPA [23] and BinPointer [24] mitigate the limitations of VSA by adopting a block memory model, where a memory block is similar to a variable in the program. BPA treats an entire block as a single unit, meaning that memory reads and writes through pointers to the block at different offsets are not distinguished. This approach assumes that the pointer arithmetic will not cause a pointer reference another block. To maintain soundness and uphold this assumption, BPA must conservatively divide the blocks,

```
1 int BZ2_bzReadOpen(...) {
2     __ptr = (int *)malloc(0x13cc);
3     iVar2 = BZ2_bzDecompressInit(__ptr + 0x4e5, ...);
4 }
5 int BZ2_bzDecompressInit(int param_1, ...) {
6   *(param_1 + 0x24) = default_bzalloc;
7 }
```

Fig. 1. Example in 401.bzip2

which compromises precision. Additionally, BPA is context-insensitive, meaning that a pointer's points-to relations are merged into a single representation without distinguishing between different calling contexts, and blocks allocated at one allocation site are treated as one block. Moreover, BPA does not differentiate between accesses to different offsets within the same block. BinPointer improves upon BPA by adding the tracking of 0-based dereferences. However, it is still limited since non-0-based dereferences will fall back to the block memory modeling. For example, in a simplified `401.bzip2` pseudocode snippet (shown in Figure 1), BinPointer fails to track a further dereference of `param_1` in `BZ2_bzDecompressInit` at Line 6 because its base is altered to `0x4e5` due to the function call. Furthermore, since BPA and BinPointer are inclusion-based and is implemented in Datalog [34], their memory consumption can be huge.

BDA [46] employs a path sampling algorithm to extract and combine the memory dependency among each path to infer the points-to relations. It is more precise than VSA, BPA, and BinPointer, but it is inherently unsound since it does not cover all execution paths. Osprey [45] relies on the points-to relations extracted by BDA and other facts obtained from the binary to infer the data structures probabilistically. Unlike traditional approaches that rely solely on memory access patterns to identify structures, Osprey incorporates additional hints, such as object copying, points-to relations, and unified access patterns, to improve inference accuracy.

3 Overview

3.1 Addressing the Challenges

To tackle the challenges mentioned in §2.1, we develop BinDSA, a tool designed for binary pointer analysis. First, given the favorable trade-off of precision and scalability of DSA [25], we consider extending DSA to binary pointer analysis and building BinDSA upon it to address **C4**. Second, we jointly recover data structure type information and points-to relations, allowing them to mutually enhance each other. In this section, we utilize a motivating example to illustrate this idea.

On the one hand, the recovery of data structure type information can increase the precision of points-to analysis. First, the recovery of data structures alleviates challenge **C2.2**, enabling a field-sensitive pointer analysis. The example in Figure 2 illustrate code snippets of two functions `foo` and `bar`. The comments on the right show the assembly code for each line of the code. In function `bar`, pointer arithmetic reveals the structure of the object referenced by `a`, as indicated by accesses to offsets such as `rdi+0x40` and `rdi+0x50`. Notably, DSA constructs a Data Structure Graph (DS Graph), where memory objects are represented by DS Nodes, which explicitly model data structures. This approach aligns with our need to recover data structures from binaries. Inspired by DSA, BinDSA leverages the concept of DS Graph to achieve this goal.

```
1  void foo(void)
2  { ...
3    y = func1;              //lea rax,[func1];mov [rsp+0x58],rax
4    z.b = 0xfef3;           //mov [rsp+0x8], rcx
5    _ptr1 = malloc(100);    //call malloc
6    z.c = _ptr1;            //mov [rsp+0x40], rax
7    z.d = func2;            //lea rax,[func2];mov [rsp+0x50],rax
8    z.a = data1;            //lea rax,[data1];mov [rsp],rax
9    bar(&z);                //mov rdi, rsp; call bar
10 }
```

```
1  typedef struct {long a;...} stream;
2  int bar(stream* arg1)
3  { ...
4    x = arg1->c;            //mov rcx,[rdi+0x40]
5    if (x == 0x0) {
6      _ptr2 = malloc(100);  //call malloc
7      x = _ptr2;            //lea rcx,[rax]
8      arg1->d();            //mov rdx,[rdi+0x50];call rdx
9    }
10 }
```

Fig. 2. A Motivating Example

Second, the recovery of data structures helps clarify variable boundaries, effectively addressing challenge **C1**. Based on the pointer dereferences within function `bar`, we can infer that the minimum size of the object referenced by `a` is `0x50`. Consequently, the stack object `z` in function `foo` must be at least `0x50` in size. The variables `z.a`, `z.b`, `z.c`, and `z.d` are all within this range. However, discerning this in assembly is challenging as they are all accessed via `rsp+offset`. For instance, Ghidra [4] treats these memory accesses as separate variables, sacrificing analysis soundness. In contrast, BPA takes a conservative approach to stack partitioning. It adopts heuristics and only considers the stack top and stack locations that are stored either in a register or a memory location as block boundaries. As a result, BPA incorrectly merges variable `y` with `z`, reducing precision. This imprecision affects the resolution of indirect call targets in function `bar`, causing BPA to incorrectly identify both `func1` and `func2` as potential targets at line 8.

Third, when recovering type information, we place particular emphasis on distinguishing between pointers and values to tackle challenge **C2.1**. Our goal is not to solve the entire type recovery problem but rather to focus on aspects critical to points-to analysis. We also leverage recovered data structure fields and their associated types to filter out incompatible candidates during merging, effectively mitigating challenge **C3**. This filtering strategy is also applied when identifying potential callee functions, further enhancing the precision of the call graph.

On the other hand, the points-to relations provide additional information to aid in the recovery of data structures. During the unification process of the pointer analysis, objects that are pointed by the same pointer are gathered together (such as `_ptr1` and `_ptr2` in Figure 2), thereby enriching the available type information. This aligns with the points-to hint mentioned in Osprey [45]. Additionally, as previously discussed, BinDSA employs a filtering mechanism to eliminate incompatible types during the merging process, thereby ensuring precision.

3.2 System Overview

The system overview of BinDSA is shown in Figure 3. BinDSA accepts a binary file as input and proceeds to disassemble and lift it into Intermediate Representations (IRs). Then it performs local analysis on each of the functions to construct a local Data Structure Graph (DS Graph). This step is described in §4 and §5.2. Subsequently, a bottom-up analysis is performed, where the DS Graph of each callee function is copied and merged into its caller's DS Graph, producing Bottom-up Graphs (BU Graphs) that summarize the total impact of invoking the caller function. During this process, indirect function calls are resolved, allowing the construction of a complete call graph. The bottom-up analysis will be further explained in §5.3. Finally, a top-down analysis is performed to refine the points-to relations by propagating caller-side information down to its callees. The details of this process are discussed in §5.4.

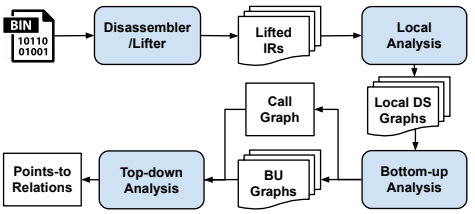

Fig. 3. Overview of BinDSA

3.3 Assumptions

The effectiveness of BinDSA relies on the following assumptions:

- All paths are considered feasible. BinDSA assumes the feasibility of all paths within the codebase, adopting a path-insensitive approach. Thereby, BinDSA ignores the conditions in conditional instructions during analysis.

- Binaries are correctly disassembled. This assumption involves the accurate identification of code and data, as well as all instructions within the binary. Without this assumption, the soundness of the analysis would be significantly compromised due to the loss of data flow.
- Proper handling of system and library calls is ensured. This assumption plays an important role in capturing potential data flow that arises from interactions with external resources and dependencies.

4 Data Structure Analysis

In this section, we present essential background and definitions related to Data Structure Analysis (DSA) to support a clearer understanding of the design of BinDSA.

4.1 Data Structure Graph

The Data Structure Graph (DS graph) G is constructed during the analysis process to represent the data structures and points-to relations. $G ::= \langle N, E, E_v, N_{call} \rangle$. Specifically,

- N is a set of DSNodes. Each DSNode $n \in N$ contains multiple fields f. A pair $\langle n, f \rangle$ is referred to as a Cell, where n is a DSNode and f is one of its fields in the data structure.
- E is a set of edges in the graph that represent the points-to relations. It is a function that maps the source cell to the dest cell, defined in Figure 4.
- E_v is a set of edges that originate from a register or a memory location var and point to the target field $\langle n, f \rangle$ referenced by var, provided var is a pointer-compatible type.
- N_{call} represents the CallSite set. $N_{call} \in N$. It contains multiple fields: r, $func$, a_1,...,a_k. r stands for the return value, $func$ is the invoked function, and a_1,...,a_k are the arguments. We denote the various fields of a CallSite CS using $CS[i]$, where i ranges from 0 to $k+1$.

Other related features are described as follows and defined in Figure 4.

- Type map T maps a node to its type.
- Global value GV records the global symbol set that a Cell $\langle n, f \rangle$ represents.
- Flags FL denotes the set of flags associated with n, e.g., the location of the DSNode: stack (denoted by S), the heap (denoted by H), or the global memory (denoted by G).

$$
\begin{array}{ll}
\langle Edge \rangle & E ::= \langle n_1, f_1 \rangle \rightarrow \langle n_2, f_2 \rangle \\
\langle VarToCell \rangle & E_v ::= var \rightarrow \langle n, f \rangle \\
\langle TypeMap \rangle & T ::= n \rightarrow \tau \\
\langle GlobalVal \rangle & GV ::= \langle n, f \rangle \rightarrow \{c\} \\
\langle Flags \rangle & FL ::= n \rightarrow S|H|G
\end{array}
$$

Fig. 4. Graph Definitions

4.2 Algorithm of DSA

Local Analysis. The local analysis phase builds a local DS Graph for each function without the information of its callees and callers. It creates empty nodes for pointer-compatible virtual registers and for globals by calling `makeNode` (describe in Table 2).

Then it performs a linear scan to process each instruction of the function to construct the graph. Figure 5 shows the local graph of function `bar`. A blue ellipse represents a variable, a rounded rectangle represents a DSNode with cells for its fields, and a sharp-cornered rectangle represents an N_{call} with cells for its fields. The number on the edge corresponds to the line of code in `bar` where the edge is created.

Table 2. Primitive operations in DSA

Name	Operation
makeNode	Creates a new, empty node n for the input register var, $E_v(var) := \langle n, 0 \rangle$.
mergeCells	Merges the first input Cell to the second Cell by merging the type information, flags, globals and outgoing edges of the two nodes, and moving the incoming edges to the resulting node. Then, the first Cell is deleted.

For example, to handle the `malloc` instruction at Line ⑥, `makeNode` is called and the heap flag H is added to the heap object. Load instruction $X = *Y$ is handled by calling mergeCells($E_v(X)$, $E(E_v(Y))$), merging the Cell of X to the Cell $E_v(Y)$ points to. `mergeCells` is an important operation in unification-based points-to algorithms and we describe it in Table 2. For example, the load instruction $x = arg1 \rightarrow c$ at Line ④ is handled in this way, where the outgoing edge of Cell c in $arg1$ has been merged with the Cell of $E_v(x)$. Function calls at Line ⑧ result in a new call node being added to the DS graph and $func$ field in call node (currently unknown and marked as *unk*) will merge with $arg1 \rightarrow d$.

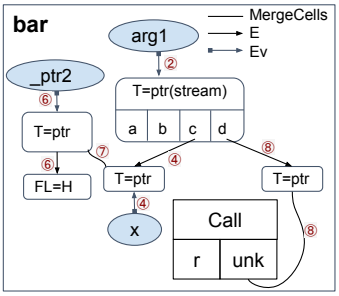

Fig. 5. Local Graph for `bar` in DSA

Bottom-up Analysis. The bottom-up algorithm resolves callees at each call site by cloning the callee's graph into the caller's graph and merging the actual and formal parameter DS nodes, along with the return nodes. Through cloning, the context-sensitivity and heap cloning are ensured.

To comprehensively explore the call graph in a bottom-up fashion, DSA runs a revised Tarjan's linear-time algorithm [40] to identify and traverse Strongly Connected Components (SCCs) in the call graph in postorder. It gives up the context sensitivity within the SCCs. Additionally, it incorporates several engineering design choices to make the context-sensitive algorithm scalable. Detailed algorithms can be found in the DSA [24] paper.

In the example, $arg1$ and Cells connected to it are copied and merged with actual argument Cells in function `foo`. Therefore, the indirect call at Line ⑧ in function `bar` can be resolved since $arg1 \rightarrow d$ is going to be merged with $\&z \rightarrow d$.

Top-down Analysis. The top-down phase, akin to the bottom-up analysis, propagates each caller's information to its callees, to recover any missed points-to relations. Note that the call graph has already been fully recovered after the bottom-up phase. Consequently, the top-down phase processes it by visiting the strongly connected components (SCCs) in reverse postorder.

5 BinDSA Algorithm

In this section, we describe the core algorithm of BinDSA, focusing on the unique features.

5.1 Definitions

Since BinDSA operates on lifted IRs (e.g., Ghidra IR) of binaries rather than source code, we redefine the relevant syntax accordingly. We also describe four unique definitions in this section.

$$
\begin{array}{ll}
\langle \text{AddressSpace} \rangle & aspace ::= ram | reg | temp | constant \\
\langle \text{Varnode} \rangle & var ::= \langle aspace, c, c \rangle \\
\langle \text{Const} \rangle & c ::= \{0, 1, 2, 3, \ldots\} \\
\langle \text{Types} \rangle & \tau ::= val | ptr(\tau) | \tau_{struct} | \tau_{func} | \top \\
\langle \text{StructTypes} \rangle & \tau_{struct} ::= \{c_1 : \tau_1, c_2 : \tau_2, \ldots\} \\
\langle \text{FuncTypes} \rangle & \tau_{func} ::= (\tau_1, \tau_2, \ldots) \rightarrow \tau_n
\end{array}
$$

Fig. 6. Varnode and Type Definitions

Varnode and Types. Figure 6 shows the definitions of Varnode in Ghidra IR and the types we defined in our algorithm. Varnode var is the high-level IR variable, which is a generalization of either a register or a memory location. It is formally represented as a triple consisting of an address space, an offset within the space, and a size, which can be denoted as $var[0]$, $var[1]$, and $var[2]$, respectively. The address space, denoted as $aspace$ (i.e., $var[0]$), includes the RAM space ram which represents the memory accessible through the main data bus, the general purpose registers space reg, virtually limitless temporary register space known as $temp$, and a designated area for constants, denoted as $constant$. Note that when the

aspace is *constant*, the offset in the triple (i.e., $var[1]$) represents the concrete constant value the Varnode represents.

Our algorithm encompasses various types, including basic values (such as int, double, and char) denoted as val, pointers of type $ptr(\tau)$ where τ represents another type, data struct types (including array) denoted as τ_{struct}, function types denoted as τ_{func}, and $\top$ that represents a variable being "any" type. Note that even though Ghidra recovered the type information, we do not depend on it. Instead, we opt to infer our own type information during the analysis.

IR. Figure 7 presents the major syntax of Ghidra high-level IR. Specifically, the STORE operation is responsible for storing the value represented by var_2 into the memory location pointed to by var_1. The CBRANCH operation evaluates var_1 as a condition and redirects the program flow to the location indicated by var_2. However, since our analysis is path-insensitive, this operation is not explicitly handled in BinDSA. The CALL operation initiates a function call to the address specified by var_1, accepting zero or more parameters and yielding zero or one return value. The LOAD operation retrieves data from the memory location specified by var. Additionally, the MULTIEQUAL operation serves as a phi-node in Single Static Assignment (SSA) form [12], combining expressions $var_1, var_2, \ldots$ originating from different paths.

$$
\begin{aligned}
program &::= stmt* \\
stmt &::= \texttt{STORE}\ var_1, var_2 \mid \texttt{BRANCH}\ var \\
&\quad \mid \texttt{RETURN}\ var \mid var := expr \\
&\quad \mid \texttt{CBRANCH}\ var_1, var_2 \\
&\quad \mid var\ \texttt{CALL}\ var_1, \ldots \\
expr &::= var \mid \diamond_U var \mid \diamond_B\ var_1, var_2 \\
&\quad \mid \diamond_{cmp}\ var_1, var_2 \mid \texttt{LOAD}\ var \\
&\quad \mid \texttt{MULTIEQUAL}\ var_1, var_2, \ldots \\
\diamond_U &::= \texttt{INT\_NEGATE} \mid \texttt{BOOL\_NEGATE} \mid \ldots \\
\diamond_B &::= \texttt{INT\_ADD} \mid \texttt{INT\_SUB} \mid \texttt{INT\_MULT} \mid \texttt{INT\_DIV} \mid \ldots \\
\diamond_{cmp} &::= \texttt{INT\_NOTEQUAL} \mid \texttt{INT\_EQUAL} \mid \ldots
\end{aligned}
$$

Fig. 7. The major Ghidra IR Syntax

Extra DS Graph Features. Additionally, we define the following unique features associated with the DS Graph to assist the analysis of binary programs, with corresponding definitions provided in Figure 8.

- *ConstVal* records the constant value a Cell $\langle n, f \rangle$ represents.
- *Stride* records the inferred stride of a DSNode n if it is an array.
- *VarRelation* captures the arithmetic relationship of two Varnodes var and var_{base}. Specifically, if $E_v(var)$ and $E_v(var_{base})$ belong to the same DSNode and the distance of their field offsets is c (which means $E_v(var)[1] - E_v(var_{base})[1] = c$).
- *StackObjs* represents the stack objects recorded for each function. $\langle n, f \rangle$ is the Cell on the stack at offset c.

5.2 Local Analysis

The local analysis in BinDSA has two main goals: constructing the DS graph for each function (as outlined in §4.1) and recovering type information. We start by introducing the AnalyzeStmt function (Algorithm 1), focusing on the differences between BinDSA and DSA in statement handling, followed by our type recovery strategy and an illustrative example. Note that a key distinction of our approach from the original DSA algorithm is the added requirement to handle pointer arithmetic and type recovery.

$$
\begin{aligned}
&\langle ConstVal \rangle && CV ::= \langle n, f \rangle \rightarrow \{c\} \\
&\langle Stride \rangle && ST ::= n \rightarrow \{c\} \\
&\langle VarRelation \rangle && VR ::= var \rightarrow \langle var_{base}, c \rangle \\
&\langle StackObjs \rangle && SO ::= c \rightarrow \langle n, f \rangle
\end{aligned}
$$

Fig. 8. Unique Definitions

Pointer arithmetics. In BinDSA, the key approach to handling pointer arithmetic is to track the *VarRelation* between two Varnodes during arithmetic operations (e.g., INT_ADD and INT_SUB) and compute the corresponding Cells for these Varnodes when required, as illustrated in Figure 9.

The INT_ADD and INT_SUB statements can represent the operation between two values or a pointer and a value. Additionally, the INT_ADD statement can represent adding a value and a pointer. Therefore, in ptrArith function, in the case where both var_1 and var_2 are values and their concrete values are known, we set the concrete value stored in var accordingly. When var_1 or var_2 is a pointer, the pointer arithmetic can only be computed if the other variable can be resolved to a constant. In this case, we will set the VarRelations VR between var_1 and var_2 with the field offset distance specified by the constant value. Note that other arithmetic operations such as INT_MULT are only handled when the concrete values for both var_1 and var_2 are known.

The last three lines in ptrArith are for identifying stack objects. We check whether var_1 represents the stack top (e.g., RSP in x86_64 programs). If so, the Cell pointed to by the Cell of var is on the stack at offset $var_2[1]$, and we store it in SO. This information is used in §5.3 to determine stack object size.

VarRelations VR is used to compute the Cells through the getCell function. Specifically, if the existing E_v contains the edge starting from var, it directly outputs the $E_v(var)$. If a Varnode relationship exists between the Varnode var and another Varnode var_{base}, it implies the correct Cell for var based on its association with var_{base}. Otherwise, the makeNode function is invoked to generate an empty node for var. Soundness is sacrificed since the relation between the new node and existing nodes is lost.

Algorithm 1 Analyze Statement

procedure AnalyzeStmt(stmt s)
  switch s **do**
    case var := INT_ADD var_1, var_2 (and other arithmetic):
      ptrArith(var, var_1, var_2, op)
    case var := MULTIEQUAL var_1, var_2, ..., var_n:
      inferStride(getCell(var), getCell(var_1), getCell(var_2)), if $n = 2$
      if ST(getCell(var)[0]) = $\perp$ **then**
        mergeCell(getCell(var), getCell(var_i)), $i \in 1, \ldots, n$
    default : same as DSA
end procedure

ptrArith(Varnode var, Varnode var_1, Varnode var_2, $\diamond_B$ op)
  if CV(getCell(var_1)) and CV(getCell(var_2)) exist
    CV(getCell(var)) := $var_1[2]$ op $var_2[2]$
  elif op = INT_ADD or op = INT_SUB
    if CV(getCell(var_2)) exist and T(getCell(var_1)) $\neq val$
      $const$:= CV(getCell(var_2))
      VR(var) := $\langle var_1, 0\ op\ const \rangle$
    if CV(getCell(var_1)) exist and T(getCell(var_2)) $\neq val$
      $const$:= CV(getCell(var_1))
      VR(var) := $\langle var_2, const \rangle$
    if var_1 is $stackTop$ and $var_2[0] = constant$
      SO($var_2[1]$) := E(getCell(var))
      FL(E(getCell(var))) := S

getCell(Varnode var)
  if var in E_v: return $E_v(var)$
  if var in VR:
    $\langle var_{base}, offset \rangle$:= VR(var)
    $\langle n_{base}, f_{base} \rangle$:= getCell(var_{base})
    $E_v(var) := \langle n_{base}, f_{base} + offset \rangle$
    return $E_v(var)$
  return makeNode(var)

inferStride(Cell $\langle n, f \rangle$, Cell $\langle n_1, f_1 \rangle$, Cell $\langle n_2, f_2 \rangle$)
  if $\langle n, f \rangle$ is loop variant
    if T(n_1) = T(n_1) = val
      ST(n) := abs(CV($\langle n_1, f_1 \rangle$) − CV($\langle n_2, f_2 \rangle$))
      CV($\langle n, f \rangle$)) := min(CV($\langle n_1, f_1 \rangle$), CV($\langle n_2, f_2 \rangle$))
    else if $n_1 = n_2$
      ST(n) := abs($f_1 - f_2$))
      copy and destroy the larger Cell

Fig. 9. Handling Pointer Arithmetics

Arrays. The pointer arithmetic is more complex in arrays. We introduce inferStride in BinDSA to assist in the handling of arrays by inferring the size of each element in the array. This function is only invoked in MULTIEQUAL statement, as arrays are typically accessed in loops, and the MULTIEQUAL statement is used to handle loop variants. It first determines whether the involved Cell is a loop variant by using heuristics, specifically checking if it is located inside a loop body and performing self-increment or self-decrement operations. Note that embedded arrays are not handled in our approach. Therefore, we also exclude any DSNode that has directly accessed fields

other than f_i, $i \in 1, ..., n$ here. A loop variant can be a value or a pointer. In the former case, the possible stride is inferred using the constant values every input cell stores, and CV(n) is the minimum among these constant values. In the latter case, the possible stride is inferred using the field values. Next, we copy the Cell with a larger field to the smaller one and destroy the larger Cell. After inferring the stride s, note that all accesses to field f are transformed to accessing f mod s. Additionally, adding a value that has stride s to a pointer results in the pointer also having a stride of s. We omit these details in the algorithm.

Other statements. Most of the statements are handled in the same way as DSA. And in `MULTIEQUAL` statement, we first attempt to infer the stride. If no stride can be inferred, each input Varnode Cell is merged with the output Varnode Cell. Besides, we do not perform collapse that is defined in DSA since it significantly sacrifices precision.

Type recovery. Data structure fields can be recovered through DS Graph, so here we focus on how BinDSA distinguishes between pointers and values:

- In `makeNode`, we check if a Varnode is a constant with a value within the program's valid address range; if true, the node is categorized as a pointer.
- In `AnalyzeStmt`, pointer type Varnodes are determined when handling pointer arithmetic for INT_ADD and INT_SUB statements. Varnodes involved in other arithmetic operations (e.g., INT_MULT) are identified as values. Besides, dereferenced Varnodes are marked as pointers.

Type Compatibility. `isCompatible` in Figure 10 is newly introduced in BinDSA which checks the type compatibility of two Cells. In `mergeCells`, we only merge the two Cells when they are compatible to ensure precision. If the Cells are not compatible, the merge process is halted. For instance, when merging $getCell(var)$ and $getCell(var_i)$ in a `MULTIEQUAL` operation, if var and var_i are incompatible, the points-to relations of var_i will not be propagated to var. While this approach may result in a loss of soundness, it helps maintain precision in the majority of cases, as it could prevent errors from propagating further in subsequent analysis.

isCompatible($\langle n_1, f_1 \rangle$, $\langle n_2, f_2 \rangle$)
  if (T(n_2)=val and T(n_1)=ptr) or (T(n_1)=val and T(n_2)=ptr): return False
  ∀ Cell $\langle n_1, f_j \rangle$ in n_1:
    if $\langle n_2, f_j + f_2 - f_1 \rangle$ exists:
      t_1 := T(E(n_1, f_j)[0]), t_2 := T(E(n_2, $f_j + f_2 - f_1$)[0])
      if (t_2=val and t_1=ptr) or (t_1=val and t_2=ptr):
        return False
  return True

Fig. 10. Type compatibility check

`isCompatible` is based on the principle that value types should be incompatible with pointer types. So if one of n_1 and n_2 is a pointer while the other is a value, the function directly returns `False`. Subsequently, if both of them are pointers, we continue to check the type compatibility of the outgoing edges of each Cell. We refrain from further verifying the type compatibility of outgoing edges at the next level, considering efficiency and the presence of `void *` type.

Example. Figure 11 shows the local DS Graph for the function `foo` and `bar` defined in Figure 2. Note that the stack objects related to `z` in function `foo` are still identified as separate variables during the local analysis phase. In `bar`, it can be deduced that `arg1` is of pointer type, pointing to a data structure of type $\{0 :, 0x40 : ptr, 0x50 : ptr\}$. Specifically, the type of data at offset 0 is unknown, while at offset $0x40$, it is

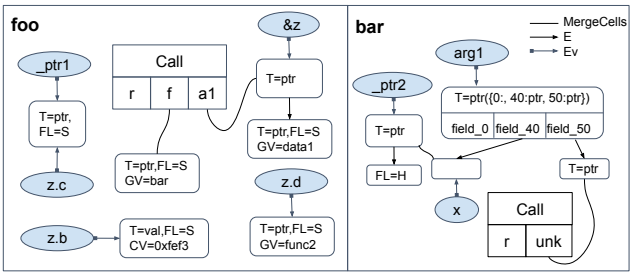

Fig. 11. Local DS Graph for `foo` and `bar`

resolveCallee(Graph G_{callee}, Graph G_{caller}, CallSite CS)
  if not isCalleeCompatible(CS, G_{callee}) : return
  G_{copied} := make a copy of graph G_{callee}
  Add nodes and edges of G_{copied} to G_{caller}
  ∀ DSNode $n \in G_{copied}$:
    ∀ global $g \in$ GV(n):
      merge n with the node containing g in G_{caller}
  mergeCells(CS[0], return Cell of G_{callee})
  for $1 \le i \le$ min(NumArgs(G_{callee}), NumArgs(CS)):
    $\langle n_a, f_a \rangle$:= $CS[i+1]$
    $\langle n_f, f_f \rangle$:= ith arg Cell of G_{callee}
    if $\langle n_a, f_a \rangle$ represents $stackTop + offset$:
      joinStkObj(SO of G_{caller}, $offset$, sizeOf(n_f))
    mergeCells($\langle n_a, f_a \rangle$, $\langle n_f, f_f \rangle$)

isCalleeCompatible(CallSite CS, Graph G_{callee})
  if arity not compatible:
    return False
  if the return Cell $\langle n, f \rangle$ of G_{callee} exists and CS[0] exists and not isCompatible($\langle n, f \rangle$, CS[0]):
    return False
  for $1 \le i \le$ min(NumArgs(G_{callee}), NumArgs(CS)):
    if not isCompatible(ith arg Cell of G_{callee}, $CS[i+1]$):
      return False
  return True
joinStkObj(StackObjs SO, Const $offset$, Const $size$)
  Cell $\langle n, f \rangle$:= $SO(offset)$
  for $offset < i < offset + size$:
    if $\langle n_i, f_i \rangle$:= $SO(i)$ exists:
      mergeCells($\langle n_i, f_i \rangle$, $\langle n, f - offset + i \rangle$)

Fig. 12. Operations used in Bottom-Up Analysis Algorithm

of type *ptr*, as it mirrors the type of the DSNode referenced by *_ptr2*. At offset $0x50$, the type is also *ptr*, as it serves as a function pointer in the `CALL` statement.

5.3 Bottom-Up Analysis

Our bottom-up analysis is similar to that of DSA. To achieve context sensitivity, the algorithm copies the callee's BU graph into the caller's BU graph and then merges the copied formal arguments/return Cells with the corresponding actual arguments/return Cells, as shown in black text in `resolveCallee` in Figure 12. Nevertheless, there are two distinct logical steps in our algorithm compared to DSA, as colored red in Figure 12.

First, we will check the compatibility between the G_{callee} and CS before merging the callee into the caller. The `isCalleeCompatible` operation in Figure 12 shows how the compatibility is checked. We first consider the arity compatibility similar to TypeArmor [41]: indirect call sites with a maximum of "max" arguments cannot target functions requiring more than "max" arguments. Additionally, call sites expecting a return value cannot jump to functions that do not provide one. Then we check the type compatibility between formal and actual arguments/returns by calling `isCompatible` defined in Figure 10.

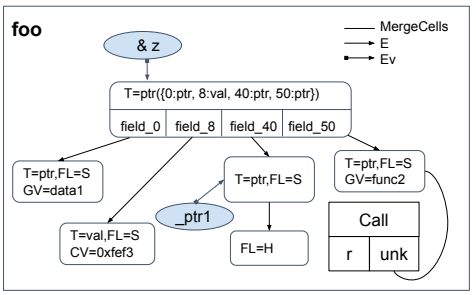

Fig. 13. BU Graph for `foo`

Second, we will combine the objects on the stack according to the size information obtained from callees. Specifically, if the ith argument Cell is a stack pointer at $stackTop+offset$, `joinStkObj` will be called with SO of G_{caller}, $offset$, and sizeOf(n_f) as arguments. sizeOf(n_f) is the size information inferred according to the Cells' field offsets in n_f. If the size is larger than the existing stack object at $offset$, we need to combine the next stack object at offset i with the stack object at offset $offset$, until i is larger than $offset$+sizeOf(n_f). Global variables are handled similarly. We rely on the global variables recovered according to heuristics first and then join the adjacent variables as necessary. The specific operation is omitted here.

Example. Figure 13 shows the BU Graph for function `foo` during the bottom-up analysis. The function `bar` has been cloned into `foo` and the formal parameters have been merged with the actual

parameters. In this process, the variables related to `z` are combined based on the size information recovered in `bar`. Local variables in `bar` are not cloned, but the heap object has been cloned into `foo` (since it is related to `arg1`) and then merged with the heap object allocated in `foo` when merging `a1` with `arg1`. The unresolved indirect call site is also cloned to `foo`. The target of this call site is resolved to `func2` when the field at offset `0x50` of `arg1` and the field at offset `0x50` of `a1` are merged. Once resolved, the algorithm will proceed to merge `func2` into `foo`. We adopt this algorithm from DSA [25].

5.4 Top-Down Analysis

In this phase, we adhere to the same algorithm introduced in DSA, which merges each function's graph into that of its callees. This process eliminates incomplete information due to incoming arguments. Since the call graph has already been recovered during the bottom-up phase, this process can be performed directly. Same as DSA, it is performed by visiting SCCs of the call graph in reverse postorder. In the same example, the DSNode related to `&z` in `foo` will be copied into `bar`.

6 Evaluation

In this section, we start by describing the experimental setup. Then we assess the performance of BinDSA by measuring its precision and recall in recovering memory access targets. Next, we evaluate the application of BinDSA in indirect-call target recovery by examining the average indirect-call target (AICT) metric and the profiling-based precision and recall. Additionally, we evaluate the efficiency of BinDSA according to the execution time and memory consumption. Lastly, we demonstrate two case studies of BinDSA: one focused on CVE reachability analysis and the other on vulnerability detection.

6.1 Experimental Setup

The experiments are conducted on a server equipped with a Ryzen 3900X CPU running at 3.80 GHz with 12 cores, accompanied by 64 GB of memory and a 500 GB SSD. It is installed with GCC-9.2, Clang-13.0, and Java-11.0. Our analysis algorithms are implemented utilizing the APIs of Ghidra version 11.0.1.

6.1.1 Dataset. We use the same dataset as the prior studies [23, 24, 47] and construct the SPEC2006's C benchmarks that contain indirect calls, along with five security-critical applications: thttpd-2.29, memcached-1.5.4, lighttpd-1.4.48, exim-4.89, and nginx-1.10. These binaries are compiled by GCC-9.2 with optimization level -O2. For testing SVF [37], we compile the source code using clang version 13.0 and then provide the resulting intermediate representations (IRs) to SVF for processing.

6.1.2 Baselines. We compare our work with BinPointer [24] regarding pointer analysis results, and compare our indirect call recovery results with several state-of-the-art binary level solutions including BPA [23], BinPointer [24], and Callee [47]. We directly adopt their results from their papers since BPA and BinPointer are not open-sourced, while Callee is a deep-learning-based approach and a doc2vec model is missing in its repository. We also include a source code points-to-analysis framework SVF [37]. Specifically, we execute the sparse flow-sensitive pointer analysis algorithm [9, 10] in SVF.

6.2 Precision and Soundness

We first analyze the precision and soundness of the pointer analysis performed by BinDSA. In this experiment, we mainly compare with BinPointer [24] since it is much more precise than BPA and VSA according to their evaluation. Besides, we evaluate the same programs in SPEC2006 as BinPointer. Given the difficulty in obtaining ground truth for points-to relations, we adopt a similar

evaluation approach as previous studies [24], leveraging profiling outcomes collected by Intel's Pin [29]. Moreover, we compare the collected runtime memory traces of benchmark programs with the memory access targets recovered by BinDSA to compute the precision and recall metrics.

In order to compare them effectively, we follow the same approach as BinPointer to dynamically convert the memory access in runtime memory traces to an abstract location in the heap, stack, and global regions. Same as BinPointer, for an instruction i that accesses memory address a, we convert it to the form (i, b, o). Specifically, i is the memory access instruction, b is a block in heap, stack, or global, and o is the offset. Blocks in the heap are differentiated by allocation site. During the profiling, we record the allocation site, allocated address, and the size of the block to convert a into b and o. Blocks in the stack are represented by their stack frame (named according to the function name) and their offsets on the stack frame. We first map the accessed memory address a to the corresponding stack frame and offset by maintaining a call stack. Then the accessed offsets on the stack frame can be converted to b and o according to block partitions of the stack frame generated by BinDSA. Moreover, blocks in the global region are represented by the global memory addresses. Similar to stack objects, we leverage the partition generated by BinDSA to decide the b and o of an accessed address a. More implementation details of the Pin tool plugin can be found in the BinPointer [24] paper.

Table 3. Profiling-based Precision of BinPointer and BinDSA

Program	Profiling-based Precision (%)					
	Stack		Global		Heap	
	BinP	BinD	BinP	BinD	BinP	BinD
mcf	**100.0**	**100.0**	85.7	**88.9**	n/a	n/a
lbm	99.5	**100.0**	**100.0**	**100.0**	n/a	n/a
lib-quantum	**100.0**	**100.0**	**100.0**	**100.0**	6.9	**10.1**
bzip2	93.2	**96.3**	51.7	**85.3**	21.8	**69.5**
sjeng	**97.5**	92.7	55.6	**61.0**	n/a	n/a
milc	**99.4**	99.0	88.9	**94.5**	**23.7**	22.6
hmmer	**99.9**	99.7	76.4	**90.1**	11.5	**26.1**
h264ref	97.3	**98.1**	**65.5**	42.3	40.8	**50.8**
Average	**98.3**	98.2	77.9	**82.8**	20.9	**35.8**

We then obtain the set of (i, b, o) triplets from the DSGraph generated after the top-down phase to compare with the profiling results. Specifically, during the local analysis phase, we record all the instructions that access a cell and the blocks that a DSNode represents. This information is propagated during the merging. After the top-down phase, we go over the Cells that are accessed by an instruction to print out the block information and the offset o can be calculated according to the field offset of a Cell.

Then we calculate the precision and recall using the same metric as BinPointer. Note that our analysis is performed on high-level IR, thus some of the memory access instructions (especially the accesses to the stack) have been optimized out. Therefore, we only consider the instructions that are available on both the profiling results and the results of BinDSA. Additionally, as also mentioned in BinPointer, the profiling-based memory accesses are a subset of the ground truth. Therefore, in this experiment, the precision is underestimated, and the recall is overestimated.

Table 4. Profiling-based Recall of BinDSA

Program	Recall (%)		
	Stack	Global	Heap
mcf	97.1	100.0	n/a
lbm	100.0	98.5	n/a
lib-quantum	97.4	100.0	100.0
bzip2	95.8	100.0	98.0
sjeng	96.6	96.3	n/a
milc	97.7	99.8	68.9
hmmer	99.6	84.2	96.7
h264ref	98.3	83.4	98.0
Average	97.8	95.3	92.3

Table 3 presents the profiling-based precision of BinPointer and BinDSA. n/a in the table indicates there are no memory references for that memory block type. As shown, the precision of BinDSA is 4.9% and 14.9% higher than BinPointer for global and heap accesses. This improvement can be attributed, in part, to the expanded field-tracking capabilities inherent to BinDSA compared to BinPointer. BinPointer only tracks 0-based dereferences so it falls back to block memory modeling and loses the field sensitivity for non-0-based dereferences (e.g., the example in Figure 1). One additional reason behind the efficacy of BinDSA lies in its implementation of heap cloning. For example, the precision of BinPointer is lower than

Table 5. AICT Evaluation Results

Programs	BinDSA			SVF	BPA		BinPointer	Callee	
	#iCallsites	Solved	AICT	AICT	#iCallsites	AICT	AICT	#iCallsites	AICT
401.bzip2	20	20	1.0	1.0	20	2.0	-	22	1.4
458.sjeng	1	1	7.0	7.0	1	7.0	-	3	7.0
433.milc	4	4	1.0	2.0	4	2.0	-	6	2.0
482.sphinx3	0	0	-	-	7	0.7	-	10	5.6
456.hmmer	10	10	3.3	2.6	10	2.8	-	12	7.2
464.h264ref	352	352	2.04	3.1	352	26.4	22.8	354	20.9
445.gobmk	44	44	86.1	503.3	44	1297.2	-	46	672.4
400.perlbench	140	114	20.8	89.7	110	363.7	-	117	354.0
403.gcc	362	345	111.2	34.5	450	427.8	-	44	338.0
nginx	210	143	8.7	176.2	331	525.1	465.2	220	383.0
lighttpd	77	48	17.6	3.5	109	33.9	-	56	31.7
exim	82	82	5.0	5.3	106	30.6	-	78	22.4
memcached	69	69	1.2	1.0	72	1.4	-	50	11.3

BinDSA in `hmmer` on heap object since there is a wrapper function for `malloc` in the program, resulting in numerous heap objects sharing the same allocation site. It is noteworthy that our evaluation metric for heap access precision does not distinguish between heap objects originating from distinct calling contexts. However, the observed enhancements in precision facilitated by BinDSA can be attributed to its heap cloning approach.

Table 4 presents the profiling-based recall of BinDSA. BinDSA reaches a recall rate of 97.8%, 95.3%, and 92.3% for stack, global, and heap accesses on average. Note that we omit the recall of BinPointer in this table as it maintains soundness. Several instances of unsoundness arise due to the failure to track pointer arithmetic or the inaccurate identification of block boundaries, leading to the loss of points-to relationships. Nevertheless, a holistic evaluation of both precision and recall reveals that BinDSA significantly improves precision while maintaining a satisfactory level of soundness.

6.3 Indirect-Call Targets Recovery

6.3.1 AICT Comparison. We utilize the traditional metric AICT (average indirect-call targets) for measuring the precision of our recovered call graph and comparing it with the state-of-the-art solutions. The results are shown in Table 5. Since we directly adopt the numbers of existing approaches from their paper, the programs we built are different and the total number of indirect call sites is slightly different. So we listed both the number of indirect call sites (denoted as *iCallsites*) and the AICT results in the table. Note that the #iCallsites are the same between BinDSA and SVF. It is also the same between BPA and BinPointer since they come from the same authors. The 482.sphinx3 program we built has no indirect call site, therefore we are unable to provide results for it. BinPointer only reports their results for 464.h264ref and nginx, thus data of other programs is unavailable. We also listed the number of solved indirect call sites of BinDSA which reflect the recall. The AICT we present for BinDSA is based only on the solved indirect call sites.

As shown, BinDSA has a significantly lower AICT compared to the existing approaches in most of the programs. For instance, it has a lower AICT than BPA on 401.bzip2 since it is field-sensitive. It is significantly more precise than BPA and BinPointer on 433.milc, 464.h264ref, and 445.gobmk because it is context-sensitive. Our heap cloning strategy is found effective on nginx because nginx encapsulates heap allocation into specific functions and heaps are allocated by calling these functions, which means heap objects cannot be differentiated solely by the allocation site. Moreover,

our algorithm successfully filters out the incompatible type mergings in nginx that are caused by errors in the lifted IR. Besides, our strategy of filtering out incompatible callees also contributes to the lower AICT of our results.

Regarding recall, BinDSA has recovered all the indirect call sites within all of the programs except for 400.perlbench, 403.gcc, nginx, and lighttpd. It is because the precision-focused design choices in BinDSA can sometimes compromise soundness. Many of the unsound cases are caused by failing to track the pointer arithmetic, which results in the loss of points-to relationships. BinDSA resolves fewer indirect call sites on nginx due to its conservative merging strategy and certain data structures in nginx incorporate a `void *` field designed to point to various types of data structures.

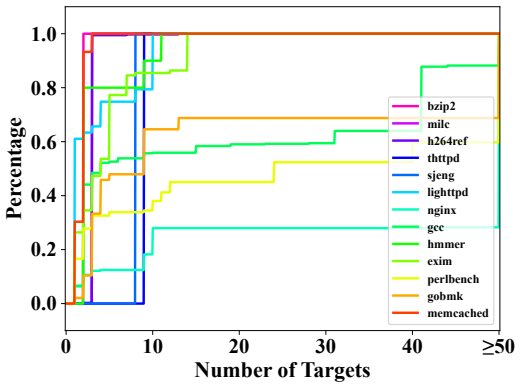

Fig. 14. Number of Targets Discovered by SVF

We also present the CDF distribution of indirect call sites based on the number of targets discovered by SVF in Figure 14, to examine the prevalence of complex indirect calls in the dataset. As shown, approximately 30% to 70% of indirect call sites in nginx, gcc, perlbench, and gobmk exhibit greater complexity, potentially invoking over 50 targets. After further analysis, we find that the recall of BinDSA in handling complex indirect call sites with over 50 potential targets is 100% for gobmk, 33% for nginx, 54% for gcc, and 87% for perlbench. A common scenario involves function pointers being stored in an array and invoked within a loop. BinDSA is equipped to handle this case effectively. However, there are cases in nginx where a single `void *` pointer references different data structures, each containing function pointers that may be invoked, resembling virtual calls in C++. These cases cannot be handled properly by BinDSA when the type-filtering strategy prevents merging, leading to a loss of recall.

6.3.2 Profiling-based Precision and Recall. The profiling-based evaluation involves gathering the real targets of indirect calls across a range of test cases. These identified targets are then considered as a reference to estimate the precision and recall of BinDSA. We adopt the same evaluation approach and metrics as BPA [23] when conducting this experiment. Specifically, we utilize the existing reference dataset in the SPEC CPU2006 benchmarks as the test cases and leverage the Pin tool to record the indirect targets at the runtime. Since these targets are a subset of the ground truth, the precision calculated based on it is lower than the real precision and the recall calculated is higher than the real recall, as also mentioned in BPA [23]. Note that we omit the 482.sphinx3 program since the program we built has no indirect call site.

Table 6. Profiling-based Precision and Recall

Programs	BinDSA		BPA	
	Precision	Recall	Precision	Recall
401.bzip2	1.00	1.00	0.30	1.00
458.sjeng	0.86	1.00	0.86	1.00
433.milc	1.00	1.00	1.00	1.00
456.hmmer	0.91	1.00	0.91	1.00
464.h264ref	0.65	1.00	0.03	1.00
445.gobmk	0.56	0.95	0.25	1.00
400.perlbench	0.53	0.86	0.35	1.00
403.gcc	0.46	0.67	0.33	1.00
Average	0.75	0.93	0.50	1.00

As shown in Table 6, BinDSA achieves equal or higher precision than BPA across all cases. Besides, BinDSA reaches a 0.93 recall on average. It resolves all indirect call sites according to

Table 7. Execution Time and Memory Consumption

Program	BinDSA		BPA		BinPointer	
	Execution Time	Memory	Execution Time	Memory	Execution Time	Memory
401.bzip2	20s	0.2G	8s	-	-	-
458.sjeng	46s	0.2G	131s	-	-	-
433.milc	50s	0.3G	33s	-	-	-
482.sphinx3	-	-	36s	-	-	-
456.hmmer	62s	0.4G	79s	0.6G	510s	1.8G
464.h264ref	193s	0.6G	379s	3.6G	23020s	8.8G
445.gobmk	785s	1.5G	1933s	28G	-	-
400.perlbench	3289s	4.8G	4006s	57G	-	-
403.gcc	1865s	24.9G	27619s	352G	-	-
nginx	140s	2.9G	1723s	23G	7692s	41G
lighttpd	90s	0.9G	-	-	-	-
exim	844s	3.6G	-	-	-	-
memcached	30s	0.7G	-	-	-	-

Table 5 but only reaches 0.95 recall on 445.gobmk in this experiment since it missed several targets at a call site. The average F1 score of BinDSA is 0.83, which is 24% higher than BPA's F1 score of 0.67.

6.4 Runtime and Memory Efficiency

The execution time and memory consumption results are presented in Table 7. We list all the reported numbers from the BPA and BinPointer paper. As shown, BinDSA exhibits notably reduced execution time and memory usage compared to BPA and BinPointer, especially for larger programs such as 403.gcc. Comparing the programs with reported results available both from BinDSA and BPA, BinDSA's average execution time is 5 times smaller than BPA, and memory consumption is 13 times smaller than BPA. This demonstrates that the unification-based aggregation strategy and the type filtering technique adopted in BinDSA have significantly improved the runtime and memory efficiency of points-to analysis.

6.5 Case Study

6.5.1 CVE Reachability. We further conduct a case study on CVE reachability, showcasing the practical application of BinDSA. CVE reachability analysis allows security researchers to assess the potential impact of identified vulnerabilities within software systems so that they can prioritize their patching efforts effectively. It's worth noting that we prefer conducting this analysis on binaries, even when the library is open-source. This preference arises due to inherent limitations of source code level analysis, such as potential modifications by vendors and the presence of various configuration options influencing the produced binary. Therefore, relying solely on source code analysis may not always be reliable.

Our analysis is on the Windows version of Zoom [5] (v5.9.7.3931), scrutinizing three vulnerabilities previously identified in the work of SigmaDiff [16], namely CVE-2020-13790, CVE-2021-38291, and CVE-2020-22037. Specifically, we first locate the vulnerable function of the CVE. Since the vulnerabilities reside in third-party libraries, the vulnerable functions are located within either `turbojpeg.dll` or `avcodec-58.dll`. Next, we run BinDSA on the two programs to generate the call graph, identifying API functions that will eventually call the vulnerable function. Then we check the reachability of these API functions in the other executables of Zoom in a similar manner. We continue this process iteratively until a main executable of Zoom is reached.

```
1  longlong alloc_small(longlong *param_1, ...) {
2      ...
3      puVar4 = (undefined8 *)jpeg_get_small(param_1,...); // calls malloc
4      ...
5      puVar8 = puVar4 + 0x18;
6      ...
7      return puVar8;
8  }
9  void * jinit_read_ppm(longlong param_1) {
10     ppcVar1 = (***(param_1 + 8))(param_1,1,0x58);  // calls alloc_small
11     *ppcVar1 = start_input_ppm;           // assign vulnerable function
12     ppcVar1[2] = finish_input_ppm;
13     return ppcVar1;
14 }
15 void * tjLoadImage(...) {
16     ...
17     local_58 = (code **)jinit_read_ppm((longlong)local_68);
18     ...
19     (**local_58)(...);     // calls vulnerable function
20     ...
21 }
```

Fig. 15. Pseudocode of related functions

Our analysis finds that among the three vulnerabilities related to the Windows version of Zoom, only CVE-2020-22037 is reachable, while CVE-2020-13790 and CVE-2021-38291 are found to be not reachable. The call path leading from the API function to the vulnerable function of CVE-2020-22037 is straightforward, without any indirect calls involved. Additionally, for CVE-2021-38291, situated within an API function, it is easily discernible that it remains uncalled by the main executables. Thus, the primary focus is on illustrating the call path of CVE-2020-13790 in Zoom, as depicted in Figure 16. This vulnerability resides within the `start_input_ppm` function in `turbojpeg.dll`. It is *indirectly* invoked by the API function `tjLoadImage`, as denoted by the dashed arrow. However, neither of the Zoom programs invokes this API function. Hence, this vulnerability is deemed unreachable.

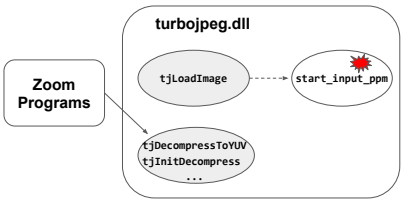

Fig. 16. Call paths of CVE-2020-13790 in Zoom

However, pinpointing the precise resolution of this indirect call within `tjLoadImage` is challenging. Figure 15 provides the pseudocode outlining the functions related to resolving this indirect call. `alloc_small` is a function that allocates small memory objects for the library and it is responsible for managing memory resources efficiently. We extract and simplify the logic of memory allocation in this function. At line 3, it calls `jpeg_get_small`, which internally calls the `malloc` function to allocate a heap object and returns the pointer to it. In the end, `alloc_small` returns the pointer to the allocated heap object plus an offset. The `jinit_read_ppm` function calls the `alloc_small` function indirectly (at line 10) and the function pointer of `start_input_ppm` is assigned to a field of the allocated heap object (at line 11). This pointer is invoked at line 19 in `tjLoadImage` which eventually calls the vulnerable function. Since `jpeg_get_small` encapsulates the `malloc` function, differentiating heap objects solely by allocation site is not precise enough and BinDSA enables heap cloning to handle this. Besides, the dereferences based on `puVar8` are non-0-based dereferences, thus BinPointer would lose its field sensitivity in this example.

6.5.2 Vulnerability Detection. Next, we perform a case study to evaluate the effectiveness of BinDSA in detecting two common CWE vulnerabilities: use-after-free (CWE-416) and double-free (CWE-415). For this purpose, we utilize the Juliet Test Suite [1] and benchmark our results against two popular binary-level vulnerability scanners: BinAbsInspector [2] and CWE Checker [3].

Table 8. Vulnerability Detection

Program	BinDSA		BAI		CC	
	FPR	FNR	FPR	FNR	FPR	FNR
CWE415	0.02	0.23	0.02	0.02	0.01	0.57
CWE416	0	0	0	0	0	0.73

BinAbsInspector implements a variant of value-set analysis on Ghidra IR to scan vulnerabilities in binaries and it leverages Z3 solver to solve path constraints.

CWE Checker, on the other hand, utilizes simple heuristics and abstract interpretation-based data-flow analysis to identify potential vulnerabilities. In BinDSA, we enhance the bottom-up analysis stage by merging callees of a function in the order of their corresponding callsites, to improve accuracy while preserving the efficiency of the analysis. This enables us to effectively identify whether a pointer is subject to double-free or use-after-free issues.

Despite the absence of path sensitivity in our approach, the results (shown in Table 8) demonstrate that BinDSA performs on par with BinAbsInspector (denoted as BAI in Table 8) in terms of false positive rate (FPR) and false negative rate (FNR). Moreover, BinDSA has significantly lower FNR compared to CWE Checker (denoted as CC in Table 8), as the data-flow analysis in CWE Checker may lose the points-to relationship during analysis. Notably, one of the key advantages of our approach is its efficiency. BinDSA is approximately 2.5 times faster than BinAbsInspector, offering a substantial improvement in analysis speed.

7 Discussion and Limitations

Loss of Soundness. The unsoundness inherent in our approach can be attributed primarily to three key factors. First, our methodology is limited in its ability to track pointer arithmetic operations. In our current framework, we only consider pointer arithmetic involving constant numerical values and handle simple cases of array access. Consequently, BinDSA overlooks scenarios where arrays are manipulated using variable or undetermined indices. Unlike BinPointer, which can revert to the block memory model when faced with failed pointer arithmetic, our approach is unification-based. If we fall back to the block model, the algorithm will merge all untracked fields together and inevitably compromise precision to a significant extent. Therefore, a trade-off must be made.

Second, the conflict-type filtering strategy we take could compromise the soundness. While this strategy effectively filters out erroneous merges resulting from mistakes in the lifting process and accurately excludes incorrect indirect callees, it also runs the risk of overlooking correct merges. For instance, situations may arise where two objects of incompatible types are both referenced by a `void *` pointer. In such cases, the filtering strategy might erroneously discard these merges, potentially leading to incomplete analysis outcomes.

Third, the current variable boundary is estimated by first identifying the base pointer and then inferring the data structure size based on memory access patterns from the base pointer. However, the size can be underestimated, causing a single variable to be split into multiple variables. Additionally, due to optimization, the base pointer is not always the start of a variable. These issues compromise soundness. We will systematically consider and solve these problems in future work.

Loss of Precision. The choice of the DSA algorithm [25], which employs a unification-based approach to points-to analysis, could sacrifice precision due to its inherent flow-insensitivity and the lack of context-sensitivity within Strongly Connected Components (SCCs). This limitation may lead to over-approximation and merging unrelated pointer variables, thereby reducing the granularity of the analysis and potentially impacting its effectiveness in scenarios requiring fine-grained pointer tracking.

For example, the effectiveness of BinDSA is limited in analyzing C++ virtual tables and multiple inheritance structures. Even though we added support for C++ libraries in order to work with the Juliet Test Suite in the case study, these programs are much simpler than real-world C++ programs. In C++, it is common for a single pointer to reference objects of two different classes, which can result in incorrect merging of their corresponding virtual tables during analysis. As a potential direction for future work, we plan to investigate hybrid approaches that combine the benefits of Steensgaard's algorithm with Andersen's, aiming to balance efficiency with precision while maximizing the recovery of type information.

8 Conclusion

In this paper, we propose BinDSA, a novel model designed for binary pointer analysis. BinDSA proposes to emphasize precision and efficiency, enabling the reverse engineering of a substantial number of programs with enhanced accuracy and speed. It jointly recovers data structure and pointers-to relations so that they can enhance each other. Its construction involves three distinct stages: local analysis, bottom-up analysis, and top-down analysis. These stages are aimed at establishing interprocedural points-to relations while concurrently utilizing the retrieved data structure types information to enhance the precision of the points-to analysis. Extensive experimental evaluations have been taken to compare BinDSA with state-of-the-art solutions such as BPA and BinPointer. The results demonstrate that BinDSA outperforms these models significantly in terms of both precision and efficiency. These findings underscore the effectiveness and practical utility of BinDSA in facilitating binary pointer analysis tasks.

9 DATA AVAILABILITY

Our code and data will be available at https://github.com/bitsecurerlab/BinDSA.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful feedback and constructive suggestions. This work was supported by NSF under grants No. 1719175 and No. 2133487.

References

[1] 2017. Juliet C/C++ 1.3. https://samate.nist.gov/SARD/test-suites/112/. (2017).
[2] 2024. BinAbsInspector. https://github.com/KeenSecurityLab/BinAbsInspector/. (2024).
[3] 2024. cwe_checker. https://github.com/fkie-cad/cwe_checker/. (2024).
[4] 2024. Ghidra. https://ghidra-sre.org/. (2024).
[5] 2024. Zoom. https://zoom.us/. (2024).
[6] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. (1994).
[7] Gogul Balakrishnan and Thomas Reps. 2010. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 1–84.
[8] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*. Springer, 84–104.
[9] Mohamad Barbar and Yulei Sui. 2021. Compacting points-to sets through object clustering. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.
[10] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object versioning for flow-sensitive pointer analysis. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 222–235.
[11] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. {SelectiveTaint}: Efficient Data Flow Tracking With Static Binary Rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*. 1665–1682.
[12] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.
[13] Manuvir Das. 2000. Unification-based pointer analysis with directional assignments. *Acm Sigplan Notices* 35, 5 (2000), 35–46.
[14] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*.
[15] Zenan Feng, Zhenyu Wang, Weiyu Dong, and Rui Chang. 2018. Bintaint: a static taint analysis method for binary vulnerability mining. In *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB)*. IEEE, 1–8.
[16] Lian Gao, Yu Qu, Sheng Yu, Yue Duan, and Heng Yin. 2024. SigmaDiff: Semantics-Aware Deep Graph Matching for Pseudocode Diffing. In *Network and Distributed System Security Symposium, February 2024 (NDSS'24)*.
[17] Tobias Gutzmann, Jonas Lundberg, and Welf Lowe. 2007. Towards path-sensitive points-to analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE, 59–68.

[18] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. *ACM SIGPLAN Notices* 44, 1 (2009), 226–238.

[19] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 289–298.

[20] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. *ACM SIGPLAN Notices* 36, 5 (2001), 254–263.

[21] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically detecting error handling bugs using error specifications. In *25th USENIX Security Symposium (USENIX Security 16)*. 345–362.

[22] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 472–482.

[23] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level.. In *NDSS*.

[24] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. Binpointer: towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 169–180.

[25] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices* 42, 6 (2007), 278–289.

[26] Yuxiang Lei and Yulei Sui. 2019. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings 26*. Springer, 27–47.

[27] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 667–678.

[28] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1100–1119.

[29] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[30] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. cfi: Type-assisted control flow integrity for x86-64 binaries. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 423–444.

[31] Erik M Nystrom, Hong-Seok Kim, and Wen-Mei W Hwu. 2004. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings 11*. Springer, 165–180.

[32] David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 4–es.

[33] Desirable Perhaps. 2015. Soundness is not even necessary for most modern analysis applications, however, as many. *Commun. ACM* 58, 2 (2015).

[34] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. 196–206.

[35] Marc Shapiro and Susan Horwitz. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–14.

[36] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.

[37] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[38] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *Programming Languages and Systems: 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings 9*. Springer, 155–171.

[39] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. 2014. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Software: Practice and Experience* 44, 12 (2014), 1485–1510.

[40] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[41] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.

[42] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 661–678.

[43] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-based selective flow-sensitive pointer analysis. In *International Static Analysis Symposium*. Springer, 319–336.

[44] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 218–229.

[45] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 813–832.

[46] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–31.

[47] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. 2023. Callee: Recovering call graphs for binaries with transfer and contrastive learning. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2357–2374.

Received 2025-02-26; accepted 2025-03-31