Hermes: Making Path-Sensitive Pointer Analysis Scalable for Sparse Value-Flow Analysis

YUXUAN HE, Xiamen University, China
RUILIN JIANG, Xiamen University, China
HE ZHANG, Xiamen University, China
QINGKAI SHI, Nanjing University, China
HUAXUN HUANG, Xiamen University, China
RONGXIN WU*, Xiamen University, China

Sparse Value-Flow Analysis (SVFA) is essential for detecting software bugs such as null pointer dereference and memory leak. However, SVFA heavily relies on path-sensitive pointer analysis, which faces significant scalability challenges when analyzing industrial-scale projects, notably the summary-explosion problem. To address this issue, we propose Hermes, which symbolizes memory side effects and constructs an incomplete Sparse Value-Flow Graph (SVFG) called Lazy Symbolic Expression Graph (LSEG). Leveraging this structure, Hermes builds inter-procedural value flows relevant to bug detection only when necessary, significantly reducing the overhead of pointer analysis and streamlining the bug-search paths. Evaluations on large-scale real-world projects demonstrate that, compared to the state-of-the-art, Hermes achieves average speedups of at least 9.84× and 4.79× for pointer analysis and bug search, respectively, without sacrificing the effectiveness of bug detection.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; *Software verification and validation*; • **Theory of computation** → *Program analysis.*

Additional Key Words and Phrases: Pointer Analysis, Sparse Value-Flow Analysis, Bug Search

ACM Reference Format:
Yuxuan He, Ruilin Jiang, He Zhang, Qingkai Shi, Huaxun Huang, and Rongxin Wu. 2026. Hermes: Making Path-Sensitive Pointer Analysis Scalable for Sparse Value-Flow Analysis. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 103 (April 2026), 29 pages. https://doi.org/10.1145/3798211

1 Introduction

Sparse value-flow analysis (SVFA) is fundamental to many recent techniques to detect bugs statically, such as null pointer dereference and memory leak [7, 11, 37, 44–46, 56, 57, 59, 62, 64, 66, 68, 69], which can be modeled as the source-sink problem [44]. Compared with symbolic execution [5, 10, 61] and data-flow analysis [2, 3, 25, 33, 38, 39], SVFA has demonstrated its high precision and scalability

*Corresponding author.

Authors' Contact Information: Yuxuan He, Xiamen University, School of Informatics, Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen, China, yuxuanhe@stu.xmu.edu.cn; Ruilin Jiang, Xiamen University, School of Informatics, Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen, China, jiangruilin@stu.xmu.edu.cn; He Zhang, Xiamen University, School of Informatics, Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen, China, zhanghexmucs@stu.xmu.edu.cn; Qingkai Shi, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China, qingkaishi@nju.edu.cn; Huaxun Huang, Xiamen University, School of Informatics, Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen, China, huanghuaxun@xmu.edu.cn; Rongxin Wu, Xiamen University, School of Informatics, Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen, China, wurongxin@xmu.edu.cn.

ACM 2475-1421/2026/4-ART103
https://doi.org/10.1145/3798211

in analyzing large-scale projects. These benefits stem from modeling data dependencies in a Sparse Value-Flow Graph (SVFG), which avoids value propagation along irrelevant control-flow paths. Despite significant advancements, pointer analysis, the prerequisite for discovering implicit data dependencies and thus building an SVFG, remains the scalability bottleneck when applying SVFA at an industrial scale [45, 69]. Our experiments revealed that Falcon [69], though considered state-of-the-art, failed to complete pointer analysis for MariaDB, a project with millions of lines of code, even after running for over 12 hours. This cost stems not only from the intrinsic complexity of pointer analysis but also from the need for path sensitivity, which is essential for practical SVFA-based bug detection [64]. When path-sensitive pointer analysis is enforced, it exacerbates the scalability challenges for SVFA significantly.

1.1 Existing Studies

Prior efforts to improve SVFA performance by mitigating pointer analysis overhead fall into two categories. The first category, known as the layered approach [4, 7, 13, 15, 24, 32, 56, 63, 68], employs a less precise but more scalable pointer analysis as a pre-analysis to approximate value flows. These imprecise value flows are subsequently refined through a more precise pointer analysis. While this strategy is generally effective, achieving path sensitivity requires the pre-analysis to distinguish value flows associated with distinct memory objects [68]. This requirement introduces numerous spurious value flows, causing the aliasing-path-explosion problem: a single value may point to multiple memory locations, and a single memory location may contain multiple values. Handling these disjunctions—enumerating all possible memory locations a pointer may reference, determining the values that can be loaded from these locations, and solving the corresponding constraints of the enumerations—presents a significant scalability challenge.

The second category, known as the fused approach [6, 11, 14, 43–47, 62, 64, 69], constructs value flows on the fly, propagating points-to information sparsely. By introducing a symbolization design, the fused approach merges values passed inter-procedurally during pointer analysis, enabling on-demand tracking of inter-procedural value flows and guard constraints during subsequent bug search. This design avoids exhaustively enumerating all inter-procedural value flows and solving their constraints, thereby mitigating scalability challenges.

Although the fused approach outperforms the layered approach by mitigating the aliasing-path-explosion problem, it still suffers from the summary-explosion problem in pointer analysis. This problem arises from how the fused approach handles the side effects of callees—namely, the modifications made by callees to memory passed from callers. It uses the memory layouts pointed to by modifications as side-effect summaries and applies them by copying the memory layouts at each callsite. This design inevitably leads to the summary-explosion problem. Specifically, when analyzing a caller function, memory layouts copied from different callees at multiple callsites may overlap and become conflated. This conflation results in increasingly complex memory layouts being referenced by pointers within the caller, which in turn makes the analysis of load instructions significantly more complicated. Additionally, the side effects associated with the caller point to inflated memory layouts. As the bottom-up pointer analysis progresses, this inflation accumulates and propagates, eventually compounding at an exponential rate and giving rise to the summary-explosion problem. We elaborate on this problem in Section 2.

1.2 Our Insight

Our first insight is that accurately modeling side effects requires only identifying which memory objects are modified, without necessarily knowing the memory layouts these modifications point to. By discarding the memory layouts referenced by side effects, we achieve two key benefits: (1) it prevents the inflation of potential memory layouts that a pointer may reference in the caller, thus

avoiding complex analysis of load instructions; and (2) it prevents the accumulation and propagation of inflated memory layouts during bottom-up pointer analysis. However, this design introduces a challenge: if a load instruction references memory within the discarded layout, the corresponding value flow cannot be established. To address this, we propose a novel symbolization scheme, in which values stored in discarded memory layouts are abstracted by symbols created on demand. Each symbol represents an access path that encodes how the corresponding value flow can later be recovered. Base on this symbolization scheme, we construct the LSEG (**L**azy **S**ymbolic **E**xpression **G**raph), an incomplete SVFG, in which the value flows of symbols' node remain unresolved and are lazily constructed only when required. This scheme not only avoids redundant copying of side-effect-related layouts, thereby alleviating the summary-explosion problem, but also supports on-demand recovery of value flows as the analysis proceeds.

Our second insight stems from the observation that, in the context of bug search, only a small fraction of inter-procedural value flows are actually relevant. Specifically, a value flow is considered meaningful only if it lies along a source-sink path or contributes to propagating path conditions necessary to verify the reachability of a source-sink path. Therefore, constructing a complete SVFG is unnecessary for bug search. However, this insight raises a challenge: when a particular value flow is never explicitly constructed, it cannot be traversed in the SVFG, making it difficult to assess its relevance to bug search. Inspired by the first solution, we observe that constructing inter-procedural value flows essentially corresponds to loading a symbol's access path through a memory layout. By examining whether a bug source is reachable from the base pointer of an access path via its offsets in the memory layout, we can determine whether the corresponding value flow is relevant to bug search without the need to construct it explicitly.

To conclude, we implemented Hermes , a path-sensitive SVFA prototype that lazily resolves only bug-relevant inter-procedural value flows. In the evaluation of 12 well-known open-source projects, Hermes reported 56 bugs with a false positive rate of 16.07%, and captured all bugs detected by Falcon, the state-of-the-art SVFA framework [69]. Regarding efficiency, Hermes demonstrates remarkable improvement in pointer analysis, achieving an average speedup of at least 9.84×. The acceleration becomes more pronounced as the code size increases, reaching a maximum of 87.76×. Since Hermes constructs only inter-procedural value flows relevant to bug search, the search space is significantly reduced, leading to an average speedup of 4.79× in the bug-search phase. Overall, Hermes speeds up bug detection by at least 6.80× on average, without sacrificing effectiveness.

In summary, this work makes the following contributions:

- We identify and analyze a key limitation in existing path-sensitive sparse value-flow analysis—namely, the summary-explosion problem.
- We present a symbolization scheme that effectively alleviates the summary-explosion problem. Leveraging this scheme, we construct LSEG, an incomplete SVFG, and design an algorithm to construct the value flows necessary for bug search.
- We implement Hermes, a system that integrates our symbolization scheme and bug-search algorithm, and evaluate it on large-scale real-world projects, demonstrating significant improvements in efficiency without compromising effectiveness.

2 Motivating Example

This section uses an example to motivate our approach by illustrating the limitations of the state-of-the-art fused approach and showing how Hermes addresses them effectively.

```
1  Deeper* global;
2  bool* bar(A* a) {//obj_sym1
3      init(a);
4      fun(a);
5      return *(a->deeper->bool_p);
6  }
```

```
7  void init(A* b) {
8      b->cond1 = true;
9      b->cond2 = false;
10 }
```

```
11  void fun(A* c){//obj_sym2
12    if(c->cond1) {
13      Deeper* d = new Deeper; //obj_1
14      d->bool_p = nullptr;
15      c->deeper = d;
16    } else {
17      c->deeper = global; //obj_2
18      global->bool_p = &c->cond1;
19    }
20    if(c->cond2)
21      c->cond1 = false;
22  }
```

Fig. 1. A code snippet for motivating our approach.

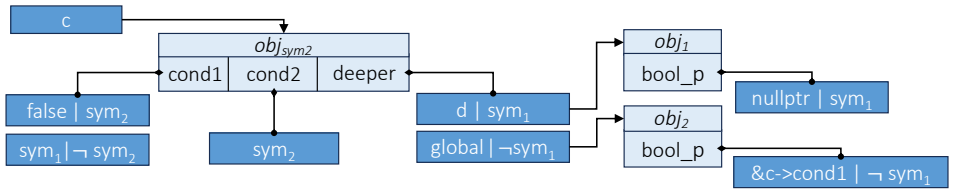


Fig. 2. Memory layout at the return site of *fun* (Hermes vs. the fused approach). Dark blue rectangles denote memory values with guard constraints; light blue rectangles denote memory objects and their fields.

2.1 Summary-Explosion Problem

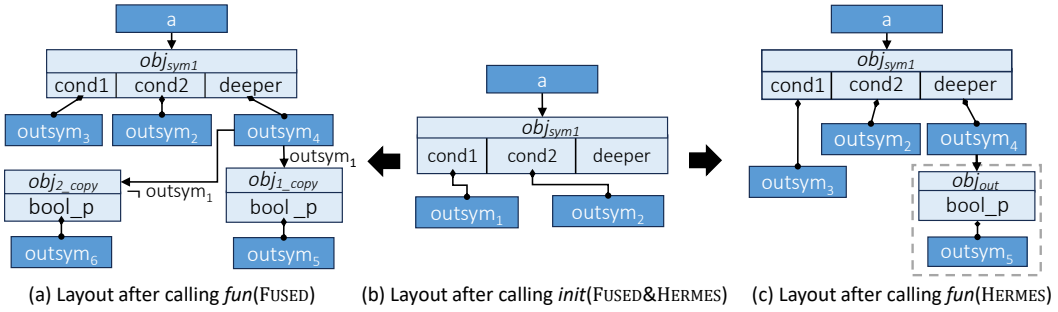
In the fused approach, inter-procedurally propagated values are symbolized, which simplifies inter-procedural value propagation and thus accelerates pointer analysis. Specifically, inter-procedurally propagated values are classified into two categories:

Auxiliary Parameter/Auxiliary Input. Since pointer analysis is performed in a bottom-up manner, it is impossible to predetermine the memory objects to which the pointer parameters point. Therefore, a symbolic object is created to abstract all possible memory objects. Each field of the symbolic object stores a symbol, referred to as an auxiliary parameter, which is used to abstract all possible values that can be loaded. Its counterpart at a callsite is referred to as an auxiliary input.

For example, in the code snippet of Figure 1, the function *fun* modifies the memory via the parameter *c*. The memory layout at the end of the function *fun* is shown in Figure 2, where each dark blue rectangle encodes a memory value with a guard constraint, a set of such rectangles forms a memory value list, and each light blue rectangle denotes a memory object. The parameter *c* points to a symbolic object obj_{sym2}, which abstracts all memory objects that *c* may point to. There are three fields in obj_{sym2}, *cond1*, *cond2*, and *deeper*, each of which has a memory value list associated with it. Since the fields *cond1* and *cond2* are loaded at Line 12 and 20, respectively, the symbols sym_1 and sym_2 are created as auxiliary parameters to represent the values that can be loaded from *c->cond1* and *c->cond2*. The corresponding auxiliary inputs at the callsite (Line 4) are the memory values of *a->cond1* and *a->cond2*. However, the field *deeper* is not loaded but directly overwritten, so there is no need to create a symbol. Since the field *c->cond1* is conditionally modified at Line 21, two values are stored in the field *cond1* of obj_{sym2}: *false* when sym_2 holds true, and sym_1 otherwise.

Auxiliary Return/Auxiliary Output. From the perspective of symbolization, a function not only has a common return value[1] but also auxiliary return values, which include values rewritten into memory objects passed via actual arguments, values in the memory layout accessible by both the common return value and the rewritten values. For example, Figure 3 (b) illustrates the memory

[1]A function may have multiple return instructions, each associated with a different return value. To simplify the analysis, these return values are typically merged into a single, unified representation, referred to as the common return value.



(a) Layout after calling *fun*(FUSED) (b) Layout after calling *init*(FUSED&HERMES) (c) Layout after calling *fun*(HERMES)

Fig. 3. Memory layouts after callsites in the function *bar* (HERMES vs. the fused approach). Dark blue rectangles denote memory values with guard constraints; light blue rectangles denote memory objects and their fields.

layout accessible by *a* after calling *init*, where the fields *cond1* and *cond2* of obj_{sym1} are updated with symbols $outsym_1$ and $outsym_2$ as auxiliary outputs, and the values of *b->cond1* and *b->cond2* at the end of the function *init* are corresponding auxiliary returns.

The fused approach uses a summary-based method. At each callsite, the callee's updates to memory passed through actual arguments, the callee's common return value and the memory layout accessible from those values are all instantiated. Despite using symbolization to abstract memory values in memory locations, it does not mitigate the explosion caused by disjunctions of multiple memory locations pointed to by a single memory value. As a result, the summaries expand quickly, posing scalability challenges for pointer analysis. Consider the memory layout shown in Figure 3 (a). After calling *fun* at Line 4 in Figure 1, the auxiliary outputs $outsym_3$, $outsym_2$, and $outsym_4$ are stored into the fields *cond1*, *cond2*, and *deeper* of obj_{sym_1} in Figure 3 (a), respectively; these auxiliary outputs abstract the values stored in the fields *cond1*, *cond2*, and *deeper* of obj_{sym_2} in Figure 2. Similarly, the auxiliary outputs $outsym_5$ and $outsym_6$ in Figure 3 (a) abstract the values stored in the field *bool_p* of the objects obj_1 and obj_2 in Figure 2. Notably, $outsym_4$ points to both obj_{1_copy} and obj_{2_copy} under the conditions that $outsym_1$ is true and false, respectively. Although the disjunction seems to aid reasoning by providing constraints, the presence of symbols obscures value details, which hinders the effective pruning of spurious edges in the memory layout. For example, the disjunction constraint atom $outsym_1$ originates from the auxiliary return of the function *init*, and its value is always *true*. However, under the fused approach's on-demand strategy, its value will not be inlined into the function *bar* until the bug-search phase. As a result, the spurious edge "$outsym_4 \rightarrow obj_{2\_copy} \mid \neg outsym_1$" cannot be pruned, forcing both obj_{1_copy} and obj_{2_copy}, along with the memory layouts that they can access, to be instantiated within the function *bar*. Even worse, this bulky memory layout—along with other auxiliary returns of *bar*—will be further instantiated at the callsites of *bar*, since $outsym_4$ is an auxiliary return of *bar*, resulting in the summary-explosion problem.

Our Insight. We observe that, for pointer analysis of each caller function, it is sufficient to identify which memory locations passed to the callee have been modified—without requiring knowledge of the memory layouts pointed to by these modifications. Therefore, rather than copying the callee's exposed memory layout, HERMES models side effects symbolically—it uses symbols as placeholders to simulate the callee's side effects, allocates symbolic objects and symbols originating from these placeholders on demand, and ignores the callee's concrete memory layout entirely. This design maximizes the extent to which disjunctions in memory layouts can be merged.

As an example, Figure 3 (c) illustrates the memory layout after calling the function *fun* in HERMES. The symbol $outsym_4$ acts as a placeholder to simulate the side effect of invoking *fun*. A symbolic

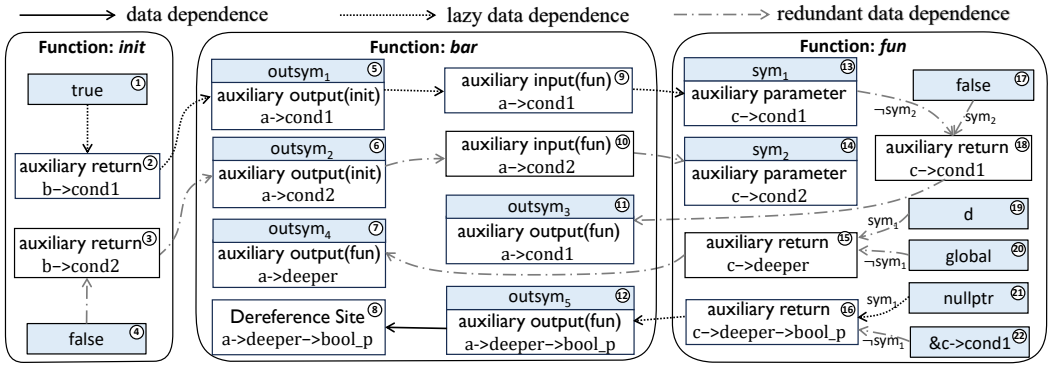


Fig. 4. The SVFG for functions *init*, *bar*, and *fun*. Solid edges represent intra-procedural data dependencies, while dotted and dash-dot edges encode inter-procedural data dependencies. Specifically, dotted edges denote value flows relevant to bug search, which Hermes constructs lazily during the bug search phase. Dash-dot edges represent value flows irrelevant to bug search and are constructed solely by the fused approach. Each node displays its associated value within a blue box and its node type within a white box. The edge labels indicate that the value flows are valid only when the labeled expressions evaluate to *true*.

object, obj_{out}, is allocated to merge the different memory objects pointed to by this symbol. In this case, obj_{1_copy} and obj_{2_copy} in Figure 3 (a) are merged into obj_{out} in Figure 3 (c). It is important to note that memory layouts are loaded on demand in our approach. At the program point before Line 5, the memory layout within the dashed box in Figure 3 (c) has not yet been created. Only after Line 5 is executed, obj_{out} is allocated, and $outsym_5$ is subsequently created and loaded on demand.

This design significantly simplifies both the construction and application of summaries. At each callsite, the pointer analysis only needs to identify the memory locations where symbols should be stored to simulate side effects. This eliminates the need to traverse memory layouts during summary construction and avoids duplicating them when applying summaries. More importantly, by avoiding the propagation of disjunctions in memory layouts from callees to callers, the memory layout accessed by pointers in the caller function becomes more compact. Consequently, the processing of load instructions, which often constitutes a considerable share of the total pointer analysis time, is significantly faster. In our experiments on million-line codebases, Hermes spends less than 2% of the time required by the fused approach to handle load instructions.

It is worth noting that this design defers the construction of symbol mappings—specifically, the mappings between auxiliary returns and auxiliary outputs, as well as between auxiliary parameters and auxiliary inputs. As a result, Hermes does not construct inter-procedural value flows during pointer analysis, but instead lazily resolves them during the bug search phase. We provide a detailed explanation in Section 2.2 and Section 3. For clarity, we refer to the SVFG nodes corresponding to these four types of symbols as interface nodes, as they serve as interfaces for value flows across different functions. We use "*resolving a symbol/interface node*" to denote the process of constructing inter-procedural value flows between a symbol (or interface node) and its corresponding counterpart.

2.2 Redundant Node Resolving

In the fused approach design, at each callsite, the mappings between all auxiliary returns and auxiliary outputs, as well as those between auxiliary parameters and auxiliary inputs, are constructed to build a complete SVFG. For example, Figure 4 illustrates a complete SVFG built by Falcon for the functions in Figure 1. The figure also shows the SVFG built by Hermes, which shares all nodes

but omits the dash-dot edges that are specific to Falcon. In this graph, we can observe an NPD (Null Pointer Dereference) source-sink path ㉑→⑯→⑫→⑧. Beyond that, five auxiliary-return nodes (②, ③, ⑮, ⑯, and ⑱) and two auxiliary-input nodes (⑨ and ⑩) are resolved in the fused approach. Among these seven resolved nodes, four (③, ⑩, ⑮, and ⑱) are irrelevant to the bug search, as they neither form edges along the source-sink path nor propagate path conditions required to verify its reachability. Consequently, the effort spent resolving these nodes is wasted.

Moreover, resolving all auxiliary inputs and auxiliary returns can be extremely expensive. Taking auxiliary inputs as an example, each represents an access path that is read by a direct or indirect callee. When the access path goes deep, it may lead to the aliasing-path-explosion problem. Given an argument *a* and an auxiliary input with an access path *a->b->c*, resolving this auxiliary intput first requires resolving the possible values of *a->b*. Since *a* may point to multiple memory locations, and each location's field *b* may store multiple values, we obtain a series of guarded values: [(val_1, $cond_1$), ..., (val_n, $cond_n$)]. For each guarded value (val_i, $cond_i$), we then need to resolve val_i*->c* under the precondition $cond_i$, in order to complete the resolving of *a->b->c*. That is, not only can the number of guarded values grow exponentially with each level of the access path, but the associated path conditions also become increasingly complex. It is clearly unwise to exhaustively resolve all auxiliary inputs at every callsite, as well as all auxiliary returns.

Our Insight. Through our symbolization scheme, Hermes first defers all the resolving of auxiliary inputs and returns, and then lazily resolves them during the bug-search phase, ensuring that only nodes related to the bug search are resolved.

For example, we can obtain an LSEG based on our symbolization scheme, as shown in Figure 4, with all redundant data dependencies discarded, since they are solely constructed by the fused approach. Solid edges representing intra-procedural value flows are generated during pointer analysis of Hermes, while dotted edges are created lazily during bug search. Additionally, all the interface nodes are explicitly labeled to indicate the callsite from which they originate. For instance, the node corresponding to the value $outsym_2$ is an auxiliary output node derived from the callsite of *init*. For simplicity, we only show the dereference site node of *a->deeper->bool_p*.

We now explain how lazy data dependence is established. During pointer analysis, all side effects are modeled using our symbolization scheme. For example, to model the side effects of the callee *fun* at Line 4 in Figure 1, the field *deeper* of the memory object obj_{sym1} is updated, and the symbol $outsym_4$ is stored as an auxiliary output, as illustrated in Figure 3 (c). Then, through the load instruction at Line 5, we create a symbolic object obj_{out} pointed to by $outsym_4$, and generate another auxiliary output symbol $outsym_5$, which is stored within obj_{out}. In this on-demand process, the data dependencies of all symbols remain uninitialized, and are established only when necessary. In the subsequent bug search phase, when the analysis reaches the function *fun*, we begin by traversing the memory layout pointed to by its side effect, searching for auxiliary returns that could serve as bug sources. In this example, as shown in Figure 2, the field *c->deeper* contains an auxiliary-return value *d*, which can access a null pointer (the source of the NPD) via the access path *d->bool_p*. Thus, we record the access path *c->deeper->bool_p* and create the corresponding auxiliary-return node ⑯, constructing the value-flow path ㉑→⑯ from the source node of the NPD to the auxiliary-return node. Then we make this value-flow path a summary, capturing how a source node can be returned from the callee *fun*. When the bug search reaches function *bar*, the recorded summary is applied, establishing a value flow ⑯→⑫ from the auxiliary-return node to its corresponding auxiliary-output node via the recorded access path, as shown in Figure 4. The bug search then continues from the auxiliary-output node ⑫ to the dereference-site node ⑧, ultimately discovering the full source-sink path: ㉑→⑯→⑫→⑧. To verify the reachability of this path, the analysis collects the path condition along the path—specifically, the symbol sym_1, whose

$$\textit{Program}\quad P := F+ \qquad\qquad \textit{Function}\quad F := f(v_1, v_2, \ldots,)\{\, S;\,\}$$

$$\begin{aligned}\textit{Statements}\quad S :=\ & v = alloc(sz_1, \ldots, sz_n) \quad|\quad v_1 = \phi((v_1, \varphi_1), \ldots, (v_n, \varphi_n)) \quad|\quad p = GEP(q, k)\\ |\ & v_1 = uniop\ v_3 \quad|\quad v_1 = v_2\ binop\ v_3 \quad|\quad *q = p \quad|\quad p = *q \quad|\quad return\\ |\ & call(v_1, v_2, \ldots) \quad|\quad if(\ldots)\ \{\, S_1;\,\}\ else\ \{\, S_2;\,\} \quad|\quad S;\ S\end{aligned}$$

Fig. 5. The syntax of the language.

corresponding auxiliary-parameter node is ⑬. Since its data dependence is unresolved, we lazily resolve the corresponding auxiliary-input node ⑨, which originates from an auxiliary-output node ⑤ at the callsite of *init*. Further on-demand interface-node resolving reveals that the condition encoded by node ⑤ evaluates to *true*, and the NPD bug is successfully detected.

In summary, we first use symbols to simulate the memory side effects of callee functions, creating symbolic objects and their internal symbols on demand to avoid copying memory layouts. Second, we avoid exhaustive interface-node resolving by lazily loading them and thus establishing value flows during bug search, thereby further enhancing the scalability of SVFA-based bug detection.

3 Approach

In this section, we formally present the design of our method. We first perform symbolization-based pointer analysis to construct an incomplete SVFG, namely the LSEG, by abstracting inter-procedurally passed values with a symbolization scheme. Then, we introduce a summary-based bug search algorithm that operates on the LSEG, resolving only bug-relevant value flows.

3.1 Preliminaries

3.1.1 Language. To formally introduce our method, we define a C-like static single-assignment (SSA) language, as illustrated in Figure 5. In this context, *uniop* and *binop* represent unary and binary operations, respectively, and we omit specific details as they are not relevant to the subsequent formalization. The *alloc* statement is used to allocate memory. Its operand is an array of type sizes since the size of an aggregate type can be decomposed into a series of atomic type sizes. Without loss of generality, all *call* statements do not return values directly. Instead, return values are obtained through arguments. For example,"*r = call()*;" is semantically equal to "*call(arg); r = *arg;*". The semantics of *GEP(p, k)* is to get the pointer to the k-th element of memory pointed to by *p*. The semantics of other statements are standard and thus omitted. The language has no looping semantics, meaning it is bounded. This standard setting in the bug-finding system follows the fused approach [69]. In our experiments, we unroll loops twice to satisfy the boundedness of programs.

3.1.2 Abstract Domains. To formally define our symbolization-based pointer analysis, we introduce a series of concepts and domains, as listed in Table 1. The domain $\mathcal{S}$ consists of symbols generated during pointer analysis, divided into two subsets: $\mathcal{S}_{param}$ and $\mathcal{S}_{out}$. The subset $\mathcal{S}_{param}$ includes both auxiliary and formal parameters, where the formal parameters are symbolic representations that abstract all possible actual arguments. The subset $\mathcal{S}_{out}$ contains auxiliary outputs. The symbols in $\mathcal{S}$, together with the program values, constitute the domain $\mathcal{V}$. The domain $\mathcal{O}$ is used to represent the memory objects allocated during pointer analysis, among which $\mathcal{O}_{con}$ denotes the concrete objects created when handling the alloc statements. $\mathcal{O}_{sym}$, which consists of the two subsets $\mathcal{O}_{sym_p}$ and $\mathcal{O}_{sym_o}$, denotes the symbolic memory objects pointed to by the parameters in $\mathcal{S}_{param}$ and the auxiliary outputs in $\mathcal{S}_{out}$, respectively. $\mathcal{L}_{sym_p}$, $\mathcal{L}_{sym_o}$, and $\mathcal{L}_{con}$ denote the locators of the objects in

Table 1. The abstract domains and their meanings.

Notation	Definition	Meaning
$\mathcal{S}$: Symbol	$\mathcal{S} = \mathcal{S}_{param} \cup \mathcal{S}_{\text{out}}$	Set of symbols generated during pointer analysis, including parameter symbols $\mathcal{S}_{param}$ and auxiliary output symbols $\mathcal{S}_{\text{out}}$.
$\mathcal{V}$: Value	$\mathcal{V} \supseteq \mathcal{S}$	Domain of program values, extended with all symbols in $\mathcal{S}$.
O: Object	$O = O_{sym_p} \cup O_{sym_o} \cup O_{\text{con}}$	Set of memory objects allocated during pointer analysis, including concrete objects O_{con} and symbolic objects O_{sym_p}, O_{sym_o}.
$\mathcal{L}$: Locator	$\mathcal{L} = \mathcal{L}_{sym_p} \cup \mathcal{L}_{sym_o} \cup \mathcal{L}_{\text{con}}$	Set of locators; each locator identifies a field inside some object in O.
Ψ: Guard	$\Psi = Fml(\mathcal{V})$	Set of first-order formulas over values $\mathcal{V}$, used as path conditions.
$\mathbb{S}$: Store	$\mathbb{S} : \mathcal{L} \mapsto 2^{\mathcal{V}\times\Psi}$	Store $\mathbb{S}$ mapping each locator l to a set of guarded values (v, φ) stored at that memory location.
$\mathbb{L}$: Location	$\mathbb{L} : \mathcal{L} \mapsto O \times \mathbb{N}$	Location $\mathbb{L}$ mapping each locator l to the object it belongs to and the corresponding field index.
$\mathbb{E}$: Environment	$\mathbb{E} : \mathcal{V} \mapsto 2^{\mathcal{L}\times\Psi}$	Environment $\mathbb{E}$ mapping each value v to guarded locators (l, φ) that it may point to.
$\mathbb{C}$: Control dependence	$\mathbb{C} : Stmt \mapsto \Psi$	Control dependence $\mathbb{C}$ mapping each statement to the guard under which it executes.
$\mathbb{D}$: Data dependence	$\mathbb{D} : \mathcal{V} \mapsto 2^{\mathcal{V}\times\Psi}$	Guarded value-flow relation $\mathbb{D}$, where $(v_2, \varphi) \in \mathbb{D}(v_1)$ indicates a data dependence $v_1 \xrightarrow{\varphi} v_2$ (i.e., v_1 may flow to v_2 under guard φ).
$\mathbb{M}$: Side-effect summary	$\mathbb{M} : F \mapsto 2^{\mathcal{L}_{sym_p}}$	Side-effect summary $\mathbb{M}$ mapping each function f to the set of symbolic parameter locators that may be modified by f.

O_{sym_p}, O_{sym_o}, and O_{con}, respectively, and each locator locates a field within an object. Ψ represents the set of first-order logic expressions formed by values $v \in \mathcal{V}$ and logical operators.

Location $\mathbb{L}$ maps a locator l to the object it belongs to and its field index. $\mathbb{C}$ records the results of control-dependence analysis, mapping each statement to the guard under which it executes. In contrast, $\mathbb{D}$ captures data dependencies between values. Unlike $\mathbb{C}$, which is given as input, $\mathbb{D}$ is constructed and updated during pointer analysis. $\mathbb{E}$ and $\mathbb{S}$ are two key data structures in pointer analysis: $\mathbb{E}$ records the guarded object locators pointed to by values, and $\mathbb{S}$ records the guarded values stored within object locators.

$$Solve_\varphi(\mathbb{S}(l)) = \{(v,\ \pi \wedge \varphi) \mid (v,\ \pi) \in \mathbb{S}(l),\ Sat(\pi \wedge \varphi)\}$$

$$Solve_\varphi(\mathbb{E}(v)) = \{(l,\ \pi \wedge \varphi) \mid (l,\ \pi) \in \mathbb{E}(v),\ Sat(\pi \wedge \varphi)\}$$

$$Load_{AP}(val, off_1, \ldots, off_n) = Load_{AP}(Load_{AP}(val, off_1, \ldots, off_{n-1}),\ off_n)$$

$$Load_{AP}(val, off) = \{(v, \delta) \mid (v, \delta) \in Solve_\pi(\mathbb{S}(l_{+off})),\ (l, \pi) \in Solve_\varphi(\mathbb{E}(v')),\ (v', \varphi) \in val\}$$

We also define operators to facilitate pointer analysis. For operator *Solve*, *Sat* performs constraint solving, using several simple yet effective rules to reject obviously unsatisfiable constraints [45, 69]. Constraints that cannot be immediately denied are tentatively considered satisfiable and are solved using an SMT solver at the bug-search phase. Additionally, we define a recursive operator $Load_{AP}$ to query the data dependence of an access path, in which l_{+k} is a notation satisfying $o_i = o_j \wedge k = off_j - off_i$ where $(o_i, off_i) = \mathbb{L}(l)$ and $(o_j, off_j) = \mathbb{L}(l_{+k})$. With these preliminaries, we now describe our symbolization-based pointer analysis.

3.2 Symbolization-Based Pointer Analysis

The rules regarding the pointer analysis are listed in Figure 6. For clarity, we use $\mathbb{E}[v \mapsto locs]$ to denote that the mapping of v in $\mathbb{E}$ is updated to $locs$. In contrast, $\mathbb{E}[v \sqsupseteq locs]$ indicates that each

$$\frac{s : v = alloc(sz_1, \ldots, sz_n) \quad \varphi_s = \mathbb{C}(s)}{\mathbb{L} \sqsupseteq \{(l_1, (o_s, 0)), \ldots, (l_n, (o_s, sz_{n-1}))\} \quad \mathbb{E}[v \mapsto \{(l_1, \varphi_s)\}]} \textsc{Alloc}$$

$$\frac{\varphi_s = \mathbb{C}(s) \quad s : v_0 = \phi((v_1, \varphi_1), \ldots, (v_n, \varphi_n))}{\mathbb{E}\left[v_0 \mapsto \bigcup_{i=1}^{n} Solve_{\varphi_i \wedge \varphi_s}(\mathbb{E}(v_i))\right]} \textsc{Phi}$$

$$\frac{s : p = GEP(q, k) \quad \mathbb{C}(s) = \varphi_s}{\mathbb{E}[p \mapsto \{(l_{+k}, \varphi) \mid (l, \varphi) \in Solve_{\varphi_s}(\mathbb{E}(q))\}]} \text{GEP}$$

$$\frac{s : v = *q \quad q \in \mathcal{S}_{param} \quad |\mathbb{E}(q)| = 0}{\mathbb{E}[q \mapsto \{(AllocSymObject(q), true)\}]} \textsc{Load-1}$$

$$\frac{s : *q = v \quad \varphi_s = \mathbb{C}(s) \quad (l, \pi) \in Solve_{\varphi_s}(\mathbb{E}(q))}{\mathbb{S}[l \mapsto \{(v, \pi)\} \cup \{(v', \pi') \in \mathbb{S}(l) \mid |\mathbb{E}(q)| > 1\}] \quad \mathbb{M}[fun \sqsupseteq \{l \mid l \in \mathcal{L}_{sym_p}\}]} \textsc{Store}$$

$$\frac{s : v = *q \quad \varphi_s = \mathbb{C}(s) \quad (l, \varphi) \in Solve_{\varphi_s}(\mathbb{E}(q)) \quad l \notin \mathcal{L}_{con} \quad |\mathbb{S}(l)| = 0}{\mathbb{S}[l \mapsto \{(AllocSym(l), true)\}]} \textsc{Load-2}$$

$$\frac{s : v = *q \quad \varphi_s = \mathbb{C}(s) \quad (l, \varphi) \in Solve_{\varphi_s}(\mathbb{E}(q)) \quad (v_i, \pi) \in Solve_{\varphi}(\mathbb{S}(l))}{\mathbb{E}\left[v_i \mapsto \{(AllocSymObject(v_i), true) \mid v_i \in \mathcal{S},\ |\mathbb{E}(v_i)| = 0\}\right] \quad \mathbb{D}\left[v_i \sqsupseteq \{(v, \pi)\}\right]} \textsc{Load-3}$$

$$\frac{s : call(arg_1, \ldots, arg_n) \quad (f, \delta) \in resolveCallee(s) \quad (param_i, off_1, \ldots, off_k) \Leftrightarrow smry \in \mathbb{M}(f) \quad (v, \varphi) \in Load_{AP}(\{(arg_i, \delta)\}, off_1, \ldots, off_{k-1}) \quad sym = AllocSym(smry) \quad (l, \pi) \in Solve_{\varphi}(\mathbb{E}(v))}{\mathbb{S}[l_{+off_k} \mapsto \{(sym, \pi)\}] \quad \mathbb{M}[fun \sqsupseteq \{l_{+off_k} \mid l_{+off_k} \in \mathcal{L}_{sym_p}\}]} \textsc{Call}$$

$$\frac{s : v = *q \quad \varphi_s = \mathbb{C}(s) \quad (l,\ \varphi) \in Solve_{\varphi_s}(\mathbb{E}(q)) \quad (v', \pi) \in Solve_{\varphi}(\mathbb{S}(l)) \quad (l', \delta) \in Solve_{\pi}(\mathbb{E}(v'))}{\mathbb{E}\left[v \sqsupseteq \{(l', \delta)\}\right]} \textsc{Load-4}$$

Fig. 6. Basic rules for updating $\mathbb{E}$ and $\mathbb{S}$.

element in *locs* is added to the existing mapping of v in $\mathbb{E}$. Similarly, $\mathbb{M} \mapsto locs$ denotes that $\mathbb{M}$ is updated to *locs*, while $\mathbb{M} \sqsupseteq locs$ indicates that each element in *locs* is added to $\mathbb{M}$.

The handling of Alloc, Phi, and GEP instructions follows standard practices. Rules related to symbolization are applied during the handling of store, load, and call instructions. In particular, by applying Load-1 through Load-4 in sequence, symbolic objects and symbols are allocated on demand to abstract concrete objects and program values propagated inter-procedurally. The rules Store and Call are responsible for generating side-effect summaries and applying them at callsites.

3.2.1 Symbolization. The rule Load-1 handles cases where there is no record of the loaded parameter pointer in $\mathbb{E}$. In such cases, a symbolic object is allocated to abstract all concrete objects that the parameter may point to. For example, when encountering the load statement at Line 12 in Figure 1, the object obj_{sym2} is allocated for the loaded parameter pointer *c* as shown in Figure 2.

The rule Load-2 handles cases where there is no record of the loaded symbolic locator in $\mathbb{S}$. In such cases, a symbol is allocated: this symbol denotes an auxiliary parameter when the symbolic object of the locator abstracts objects passed from a parameter and an auxiliary output when the symbolic object of the locator abstracts objects returned by a callee function. For example, upon encountering the load statement at Line 20 in Figure 1, the symbol sym_2 is allocated for the field *cond2* of obj_{sym2} as an auxiliary parameter, as shown in Figure 2. Similarly, when encountering the load statement at Line 5 in Figure 1, the symbol $outsym_5$ is allocated for the field *bool_p* of obj_{out} as an auxiliary output, as shown in Figure 3(c).

The rule Load-3 establishes value flows for each guarded value obtained by querying $\mathbb{E}$ and $\mathbb{S}$. Additionally, if the value is a pointer symbol with no existing record in $\mathbb{E}$, a symbolic object is allocated for it. For example, when encountering the Load statement *a->deeper* at Line 5 in Figure 1, the symbolic object obj_{out} is allocated for the loaded symbol $outsym_4$, as illustrated in Figure 3(c). This ensures that the environment $\mathbb{E}$ is correctly updated when applying the rule Load-4, which is adopted from the fused approach and facilitates sparse propagation of points-to information.

Remark 3.1. Load-1, Load-2, and Load-3 specify how symbols and symbolic objects are allocated on demand. Each symbol points to a unique symbolic locator recorded in $\mathbb{E}$ (Load-1, Load-3). Each symbolic object locator stores at most one corresponding symbol (Load-2). Locators within the same object can reach each other through an offset. As a result, each symbol corresponds to a unique access path, which dictates how it can be resolved. For example, the auxiliary parameter sym_2 in Figure 2 corresponds to the access path *c->cond2*. Specifically, if a symbol denotes an auxiliary parameter, the base pointer of the access path is a formal parameter. If a symbol denotes an auxiliary output, the base pointer is a symbol allocated by applying side-effect summary.

3.2.2 Side-Effect Summary. As explained in Section 2, the fused approach suffers from the summary-explosion problem due to exhaustive duplication of the memory layout. In response, our summary records only modifications to memory objects passed to callee functions, excluding the memory layout that is accessible from the modifications. When the modifications are loaded within the caller function, Hermes only generates symbols on demand by applying the rule Load-2 and Load-3. These symbols, referred to as auxiliary outputs, have their data dependencies (the corresponding auxiliary returns) left unresolved. Through this summary design, we avoid exhaustive duplication of the memory layout accessible from callsite output values, encapsulating pointer analysis for every single function, thus ensuring the efficiency of pointer analysis.

The summary generation is driven by updates to the abstract domain $\mathbb{S}$ via the rules Store and Call. Specifically, whenever a value is stored in an object locator belonging to $\mathcal{L}_{sym_p}$, it indicates that memory objects passed in by parameters have been modified. Thus, the locator is added to the side-effect summary $\mathbb{M}$ associated with the current function *fun*, as shown in the rule Store and Call. For example, in the function *fun* in Figure 1 and its memory layout in Figure 2, the fields *deeper* and *cond1* of obj_{sym2} are added to $\mathbb{M}(fun)$ as side-effect summaries, because they are modified by the store statements at Line 15, 17 and 21 in Figure 1.

The summary application is defined in the rule Call. For each potential callee *f*, its side-effect summary $\mathbb{M}(f)$ is queried. For each locator in the summary, we compute its access path $[param_i, off_1, \ldots, off_k]$, where $param_i$ denotes the i-th formal parameter of the callee *f*. The computation of the access path is trivial and can be derived from the Remark 3.1, so we omit the details. The operator $Load_{AP}$ takes the access path as input to calculate the locators modified by the callee—specifically, those pointed to by values of arg_i->off_1-> $\ldots$ ->off_{k-1} plus the offset off_k. Since these locators represent the targets of the side effect, a symbol is then allocated and stored in them. Consider Line 4 in Figure 1. For *c->deeper*, one of the side-effect summaries of the callee *fun*, $outsym_4$ is allocated and stored in the field *deeper* of the object pointed to by *a*, namely obj_{sym1}.

Remark 3.2. By leveraging side-effect summaries, Hermes enables modular and efficient bottom-up pointer analysis. Unlike prior approaches, Hermes avoids eager inter-procedural propagation of points-to information and data dependencies, and instead captures such propagation symbolically—using symbols and symbolic objects to abstract inter-procedurally propagated program values and concrete objects, respectively.

Using the above rules, Hermes efficiently performs path-sensitive, field-sensitive, and context-sensitive symbolization-based pointer analysis. Load-1 and Call initialize the foundation of

symbolization by modeling all formal parameters and callee side effects as symbols—these symbols serve as entry points for accessing the other inter-procedurally propagated values, including auxiliary parameters and auxiliary outputs. Building on this foundation, Load-2 and Load-3 allocate symbols and symbolic objects on demand, thereby abstracting all inter-procedural values into symbols. When constructing value flows via Load-3, all flows are built between symbols and intra-procedural program values, and each symbol is tied to a unique access path that enables their resolving when needed. As a result, Hermes does not eagerly build inter-procedural value flows, but leaves their construction to the bug search phase.

Remark 3.3. Constructing inter-procedural value flows, i.e., resolving the data dependencies of symbols, amounts to performing load operations along their access paths. This process, implemented via the operator $Load_{AP}$, can trigger a rapid performance explosion: a single value may point to multiple guarded object locators, each of which may store multiple guarded values. To resolve deeper levels, the analysis must enumerate all guarded locators reached by the guarded values from the previous level, as well as the guarded values within those locators. As the path depth increases, the number of disjunction cases grows exponentially, and reasoning over them can incur significant overhead. Hermes avoids this cost by resolving symbols only when necessary for bug search.

3.3 Summary-Based Bug Search

Hermes conducts bug search in a summary-based manner, following the standard setup of SVFA-based bug search frameworks [69]. However, the SVFG constructed by Hermes—based on the rules in Figure 6—does not directly support traditional summary-based bug search, since it does not construct inter-procedural value flows. We first introduce the Symbolic Expression Graph (SEG), which serves as the SVFG instance in the fused approach, along with its associated summary-based bug search algorithm. Then, we explain how LSEG—the incomplete SVFG built by Hermes—is adapted to summary-based bug search by incrementally constructing inter-procedural value flows.

3.3.1 Preliminaries.

Definition 3.1 *(Symbolic Expression Graph).* The Symbolic Expression Graph for a program P, which is an instance of SVFG adopted by the fused approach, is $(\mathcal{N}, \mathcal{E}_d, \mathcal{E}_c, \mathcal{L}_d)$, where:

- The set $\mathcal{N}$ comprises all nodes in the graph. A node in $\mathcal{N}$ either represents a variable defined by a program statement or serves as an interface node. The interface nodes include parameter nodes (representing formal and auxiliary parameters), input nodes (nodes corresponding to parameter nodes), auxiliary-return nodes, and auxiliary-output nodes, denoted as $\mathcal{N}_{param}$, $\mathcal{N}_{in}$, $\mathcal{N}_{ret}$, and $\mathcal{N}_{out}$, respectively. The value flows associated with these interface nodes are fully resolved. For clarity, $\mathcal{N}^f$ is used to denote the set of nodes in a function f.
- $n_1 \rightarrow n_2 \in \mathcal{E}_d \subseteq \mathcal{N} \times \mathcal{N}$ represents a value flow, or data dependence.
- $n_1 \rightarrow n_2 \in \mathcal{E}_c \subseteq \mathcal{N} \times \mathcal{N}$ encodes the control dependence in the graph, depicting that n_1 is reachable if and only if n_2 is satisfiable.
- $\mathcal{L}_d$ maps each value flow in $\mathcal{E}_d$ to the constraint $\varphi \in \psi$ under which the value flow holds.

Example 3.1. Figure 4 shows the SEG constructed from the code in Figure 1. All nodes in the graph form the set $\mathcal{N}$, all types of data dependence edges collectively form $\mathcal{E}_d$, and the guard constraints associated with these edges are recorded in $\mathcal{L}_d$.

Definition 3.2 *(Source-Sink Specification).* A source-sink specification τ is a triplet $(\sigma_{src}, \sigma_{sink}, \delta_{reach})$ that describes the properties of a bug. The predicates σ_{src} and σ_{sink} take nodes from the SEG as input and determine whether a given node represents a source or a sink, respectively. The indicator δ_{reach} is a boolean flag that determines the reachability condition for the bug:

- If δ_{reach} is *true*, a bug occurs when there exists a reachable value-flow path from a source node to a sink node. This type of bug is referred to as the *source-must-not-sink* bug.
- If δ_{reach} is *false*, a bug occurs when a value-flow path originating from a source node fails to reach a corresponding sink node. This type of bug is referred to as the *source-must-sink* bug.

Example 3.2. For the NPD (Null Pointer Dereference) bug, $\sigma_{src}(v)$ holds iff v is a null pointer assignment, $\sigma_{sink}(v)$ holds iff v is a load instruction that dereferences a pointer, and δ_{reach} is *true*. For the ML (Memory Leak) bug, $\sigma_{src}(v)$ holds iff v is a heap allocation, $\sigma_{sink}(v)$ holds iff v is a memory deallocation, and δ_{reach} is *false*.

Definition 3.3 *(Static Bug Check).* Given a SEG $(\mathcal{N}, \mathcal{E}_d, \mathcal{E}_c, \mathcal{L}_d)$ and a bug specification $\tau = (\sigma_{src}, \sigma_{sink}, \delta_{reach})$ of a program P, for any path $\pi = n_1 \rightsquigarrow n_k$ we define its *path condition*

$$\Phi_\pi = \Big(\bigwedge_{i=1}^{k-1} L_d(n_i \to n_{i+1})\Big) \wedge \Big(\bigwedge_{i=1}^{k} \bigwedge_{n_c \in \{\, n \mid n_i \to n \in \mathcal{E}_c \,\}} n_c\Big) \wedge \sigma_{\text{src}}(n_1) \wedge \sigma_{\text{sink}}(n_k).$$

Let $\text{Sat}(\cdot)$ and $\text{Valid}(\cdot)$ denote satisfiability and validity, respectively. Then the static bug check is defined as follows:

- If $\delta_{reach} = true$ (source-must-not-sink bug), a bug is detected when there exists a path π such that $\text{Sat}(\Phi_\pi)$, i.e., some feasible path connects a source node to a sink node.
- If $\delta_{reach} = false$ (source-must-sink bug), a bug is detected when there exists a source node n_{src} such that $\neg\text{Valid}\big(\bigvee_{n_{\text{sink}}} \Phi_{n_{\text{src}} \rightsquigarrow n_{\text{sink}}}\big)$, i.e., under some feasible path the source node fails to reach any sink.

Example 3.3. Given the NPD specification $\tau = (\sigma_{\text{src}}, \sigma_{\text{sink}}, \delta_{\text{reach}})$ in Example 3.2 with $\delta_{\text{reach}} = true$, the path $\pi =$ ㉑→⑯→⑫→⑧ in Figure 4 is identified as an NPD bug according to Definition 3.3, because its path condition Φ_π is satisfiable: (i) σ_{src} holds at node ㉑; (ii) σ_{sink} holds at node ⑧; and (iii) the control dependence along π are met (in particular, the constraint on sym_1 holds with node ⑬ evaluating to *true*). Equivalently, a source node reaches a sink node along a feasible path.

3.3.2 Traditional Summary-Based Bug Search.

Definition 3.4 *(Function Summary).*

- *Input Summary*. Given a source-sink specification $\tau(\sigma_{src}, \sigma_{sink}, \delta_{reach})$, an input summary of a function f is a value-flow path $n_1 \rightsquigarrow n_k$, where $n_1 \in \mathcal{N}^f_{param}$ is a parameter node of f and $\sigma_{sink}(n_k) = true$, indicating that n_k is a sink node.
- *Output Summary*. Given a source-sink specification $\tau(\sigma_{src}, \sigma_{sink}, \delta_{reach})$, an output summary of a function f is a value-flow path $n_1 \rightsquigarrow n_k$, where $\sigma_{src}(n_1) = true$, indicating that n_1 is a source node, and $n_k \in \mathcal{N}^f_{ret}$ is a auxiliary-return node of f.
- *Transfer Summary*. A transfer summary of a function f is a value-flow path $n_1 \rightsquigarrow n_k$, where $n_1 \in \mathcal{N}^f_{param}$ is a parameter node of f and $n_k \in \mathcal{N}^f_{ret}$ is a auxiliary-return node of f.

Example 3.4. In Figure 4, the path ⑬→⑱ is a transfer summary, capturing how the callee returns a parameter. The path ㉑→⑯ is an output summary for NPD detection, capturing how a *nullptr* flows to the return. For clarity, many dereference sites are omitted from Figure 4, but they can be observed in Figure 1, where every function parameter is dereferenced. Thus, input summaries exist for all parameters.

Algorithm 1 (excluding Line 4) illustrates the standard summary-based bug search procedure on the SEG. To make it applicable to the LSEG, Line 4 is added to construct inter-procedural value flows relevant to bug search. We elaborate on this in Section 3.3.3.

Algorithm 1: Algorithm for bug search while creating and utilizing summary

```
1  toCheck = [ ]; // list of source-sink paths to check;
2  Function BugSearch(τ, G):
3      for f in bottom-up order do
4          ResolveFlows(f, G) // inserted when G is a LSEG;
5          for n ∈ {n ∈ N^f | δ_src(n)} do
6              SearchFromSrc (n, [ ]);
7          for n ∈ {n ∈ N^f_out | hasOutputSmry(n)} do
8              SearchFromSrc (n, getOutputSmry(n));
9          for n ∈ N^f_param do
10             SearchFromParam (n, [ ]);
11 Function SearchFromSrc(node, trace):
12     trace ← trace + [node];
13     if δ_sink(node) then
14         toCheck ← toCheck + [trace];
15     if node ∈ N^f_in ∧ hasInputSmry(node) then
16         toCheck ⊒ trace + getInputSmry(node);
17     if node ∈ N^f_ret then
18         createOutputSmry(trace);
19     // Use transfer summaries and edges in E_d for traversal;
20     // E^+_d = E_d ∪ {(n_in, n_out) ∈ N^f_in × N^f_out | ∃ n_param ⇝ n_ret};
21     for node → chlid ∈ E^+_d do
22         SearchFromSrc (child, trace);
23 Function SearchFromParam(node, trace):
24     trace ← trace + [node];
25     if δ_sink(node) then
26         createInputSmry(trace);
27     if node ∈ N^f_in ∧ hasInputSmry(node) then
28         createInputSmry(trace + getInputSmry(node));
29     if node ∈ N^f_ret then
30         createTransferSmry(trace);
31     for node → chlid ∈ E^+_d do
32         SearchFromParam (child, trace);
```

In bottom-up bug search, two types of nodes are selected as the starting points to search for each analyzing function *f*. The first type includes source nodes—either those identified by the predicate σ_{src} or auxiliary-output nodes whose corresponding auxiliary-return nodes have output summaries. The second type consists of parameter nodes, which may propagate bug sources. For example, when detecting NPD, the null pointer assignment at Line 14 in Figure 1 is identified as a first-type node. The auxiliary-output node of *a->deeper->bool_p* is also identified as a first-type node, as its corresponding auxiliary-return node has an output summary returning *nullptr*. The nodes corresponding to the parameter *a* and auxiliary parameter *a->deeper* in the function *bar* are identified as second-type nodes, which may serve to transfer bug sources.

During searching from the first-type nodes (Line 6 and Line 8), encountering a sink node yields a source-sink value-flow path, which is added to the *toCheck* list (Line 14). This list records all source-sink value-flow paths, which can ultimately be input for bug validation following the Definition 3.3. When the search reaches the function boundary, specifically an input node or an auxiliary-return node, the input summary is applied to stitch together a source-sink value-flow path (Line 16), and an output summary is generated (Line 18), respectively.

The reason for starting the search from parameter nodes is to generate input and transfer summaries. When a sink node is encountered, an input summary for a parameter node is generated (Line 26). If an input node is reached and its corresponding parameter node in the callee has an input summary, a new input summary is created accordingly (Line 28). When the search reaches a auxiliary-return node, a transfer summary is generated (Line 30).

As can be observed, Algorithm 1 presupposes that the auxiliary-input and auxiliary-return nodes have been resolved, e.g., the creation of the input summary requires the auxiliary-input nodes to be resolved (Line 28), and the creation of the output summary requires the return node to be resolved (Line 18). As described in Remark 3.3, the overhead of resolving all these nodes is prohibitively high and unnecessary. Our insight is that only two types of interface nodes require resolving: (1) those involved in source-sink transfer (intermediate nodes along a source-sink path), and (2) those involved in source-sink reachability (nodes whose data dependencies serve as path conditions).

Algorithm 2: Algorithm for resolving source-sink transfer node

```
1  Function ResolveFlows(f, Ĝ):
2      Ĝ = (N̂, Ê_d, Ê_c, L̂_d);  Out_src ← ∅ ;
3      for (callsite, callee) of f in topological order do
4          InSmry, TranSmry, OutSmry = getSmry(callee);
5          for n_src ⇝ n_ret ∈ OutSmry do
6              n_out ← mapping of n_ret;
7              Out_src ⊒ n_out;
8          for n_param ⇝ n ∈ InSmry ∪ TranSmry do
9              ap = (param_i, off_1, ..., off_k) ← getAP(n_param);
10             G̃ ← (Ñ_k ← ∅, Ñ_a ← ∅, Ẽ ← ∅, L̃ ← ∅);
11             buildVAGWithAP(arg_i, ap, 0, G̃);
12             n_in ← mapping of n_param;
13             for (v, φ) ∈ Load_AP({arg_i}, off_1, ..., off_k, G̃) do
14                 n' ← nodeOf(v);
15                 Ê_d, L̂_d ⊒ (n', n_in), ((n', n_in), φ);
16                 if n_param ⇝ n ∈ TranSmry then
17                     n_out ← mapping of n;
18                     Out_src ⊒ n_out;
19     for v ∈ S(l ∈ M(f)) do
20         G̃ ← (Ñ_k ← ∅, Ñ_a ← ∅, Ẽ ← ∅, Ẽ ← ∅);
21         n_1 ← nodeOf(v);
22         buildVAGWithoutAP(v, f, G̃);
23         for n_1 ⇝ n_k ∈ G̃ where n_k ∈ Ñ_k do
24             n_ret ← mapping of (l, ..., L̃((n_{k-1}, n_k)));
25             for (v', φ) ∈ Load_AP({v}, ..., L̃((n_{k-1}, n_k)), G̃) do
26                 n' ← nodeOf(v');
27                 Ê_d, L̂_d ⊒ (n', n_ret), ((n', n_ret), φ);
28 Function buildVAGWithoutAP(v, f, G̃):
29     ap = (ptr, off_1, ..., off_k);  G̃ = (Ñ_k, Ñ_a, Ẽ, L̃);
30     n ← nodeOf(v);
31     if n ∈ Ñ_k ∪ Ñ_a then
32         return;
33     if trackToKey(n, f) then
34         Ñ_k ⊒ n;
35     if v ∈ S_out then
36         if ∃ s derived from v, nodeOf(s) ∈ Out_src
37          then
38             ap ← (v, off_1, ..., off_k);
39             buildVAGWithAP(v, ap, k, G̃);
40         return;
41     if v ∈ S_param then
42         return;
43     for off that is valid do
44         val ← Load_FS(v, off);
45         for v' ∈ val do
46             buildVAGWithoutAP(v', G̃);
47             n' ← nodeOf(v');
48             if n' ∈ Ñ_k ∪ Ñ_a then
49                 Ñ_a, Ẽ, L̃ ⊒ n, (n, n'), ((n, n'), off);
50 Function trackToKey(n, f):
51     if ∃n' ⇝ n, σ_src(n') ∨ n' ∈ Out_src ∪ N̂^f_param then
52         return true;
53     return false;
```

Next, we give the definition of LSEG and introduce how to augment it by invoking `ResolveFlows` (Algorithm 1, Line 4), so that LSEG can be adapted to traditional summary-based bug search.

3.3.3 *Lazy Node Resolving.*

Definition 3.5 *(Lazy Symbolic Expression Graph).* The Lazy Symbolic Expression Graph $\hat{G}$ for a program P, denoted as $(\hat{\mathcal{N}}, \hat{\mathcal{E}}_d, \hat{\mathcal{E}}_c, \hat{\mathcal{L}}_d)$, is a subgraph of the SEG $(\mathcal{N}, \mathcal{E}_d, \mathcal{E}_c, \mathcal{L}_d)$, where $(\mathcal{N} \setminus \hat{\mathcal{N}}) \subseteq (\mathcal{N}_{in} \cup \mathcal{N}_{ret})$, with edges $\hat{\mathcal{E}}_d \subseteq \mathcal{E}_d$, $\hat{\mathcal{E}}_c \subseteq \mathcal{E}_c$, and labels $\hat{\mathcal{L}}_d \subseteq \mathcal{L}_d$. This implies that interface nodes that are irrelevant to bug search and their value flows are discarded from the LSEG while remaining in the SEG. Before bug search, $(\mathcal{N} \setminus \hat{\mathcal{N}}) = (\mathcal{N}_{in} \cup \mathcal{N}_{ret})$. As the bug search advances, the bug-related interface nodes and their value flows are progressively incorporated into the LSEG.

Example 3.5. Figure 4 shows the LSEG constructed for the code in Figure 1. Before bug search begins, the graph contains only an intra-procedural value flow ⑫→⑧, and none of nodes from $\mathcal{N}_{ret}$ or $\mathcal{N}_{in}$ are present. As bug search proceeds, interface nodes relevant to bug search—i.e., ⑯, ⑨, and ②—are resolved and added to $\hat{\mathcal{N}}$. Their associated value flows are then added to $\hat{\mathcal{E}}_d$, with corresponding guard constraints recorded in $\hat{\mathcal{L}}_d$.

Algorithm 1 reveals that only interface nodes related to source-sink transfer directly impact the bug search process, whereas those related solely to source-sink reachability do not. Accordingly, we

first resolve the interface nodes related to source-sink transfer to ensure that *toCheck* in Algorithm 1 is complete. Then, interface nodes related to source-sink reachability are resolved according to Definition 3.3. However, a paradox arises: determining which interface nodes are involved in source-sink transfer requires knowledge of the value flows of all interface nodes—without such knowledge, their relevance to source-sink transfer cannot be established through graph traversal. This requirement directly contradicts our goal of resolving only those nodes that are relevant to bug search. We observe that resolving an interface node amounts to performing a load operation on its access path (see Remark 3.1 and $Load_{AP}$). As discussed in Section 2.2, such operation is a computation process of state explosion, primarily because the computation of the same value under different guard constraints cannot be merged. As the access path deepens, constraints multiply and become more complex, making even lightweight constraint solving expensive. Our insight is that flow-sensitive points-to information—easily derived from path-sensitive pointer analysis—can be used to prune during loading an access path, thus accelerating interface-node resolving. Next, we introduce the Value Access Graph (VAG), explain how to construct it using flow-sensitive queries, and present a pruning strategy guided by VAG.

Definition 3.6 *(Value Access Graph).* The Value Access Graph $\tilde{G}$, which encodes value access relation at a program point $\mathfrak{p}$, is $(\tilde{\mathcal{N}}_k, \tilde{\mathcal{N}}_a, \tilde{\mathcal{E}}, \tilde{\mathcal{L}})$, where:

- $\tilde{\mathcal{N}}_k$ and $\tilde{\mathcal{N}}_a$ are subsets of $\hat{\mathcal{N}}$ in LSEG, representing two types of nodes. $\tilde{\mathcal{N}}_k$ includes **k**ey nodes that satisfy certain properties, such as those identified by σ_{src}. $\tilde{\mathcal{N}}_a$ includes nodes whose values can **a**ccess those of nodes in $\tilde{\mathcal{N}}_k$ via the memory layout at $\mathfrak{p}$. For example, in Figure 2, d can access *nullptr*. Thus, the node of d belongs to $\tilde{\mathcal{N}}_a$ if the node of *nullptr* belongs to $\tilde{\mathcal{N}}_k$.
- $n_1 \rightarrow n_2 \in \tilde{\mathcal{E}} \subseteq (\tilde{\mathcal{N}}_k \cup \tilde{\mathcal{N}}_a) \times (\tilde{\mathcal{N}}_k \cup \tilde{\mathcal{N}}_a)$ represents an access path between values.
- $\tilde{\mathcal{L}}$ maps each edge in $\tilde{\mathcal{E}}$ to an offset. For example, in Figure 2, the edge from the node of d to the node of *nullptr* is labeled by *bool_p*, because d could access *nullptr* by offset *bool_p*.

Algorithm 2 defines the function `ResolveFlows`, which is invoked at Line 4 in Algorithm 1. It describes how source-sink transfer interface nodes in LSEG are lazily resolved, and how the VAG is leveraged to prune disjunction cases during loading an access path. When performing bottom-up bug search, for each analyzing function f (Algorithm 1, Line 3), we assume the summaries for all callee functions (i.e., input summary, output summary, and transfer summary) are complete (Algorithm 2, Line 4). If we can establish all summaries for f and detect all the source-sink paths within the function f, then our approach is self-consistent.

We begin by describing how to construct the necessary value flows for auxiliary-input nodes. This is handled in the for-loop at Line 8 of Algorithm 2, which traverses input and transfer summaries $n_{param} \rightsquigarrow n$ for each callsite in the topological order of basic blocks within f. For each parameter node n_{param}, its corresponding access path is given by $(param_i, off_1, ..., off_k)$, where $param_i$ denotes the i-th formal parameter of the callee. This access path indicates how the auxiliary-input node should be resolved. Since only value flows relevant to source-sink transfer are needed, we construct a VAG indexed by the access path (Line 11). The definition of `buildVAGWithAP` is omitted, as it largely mirrors `buildVAGWithoutAP`. Like the latter, `buildVAGWithAP` leverages flow-sensitive pointer analysis to retrieve load results for each access step without constraint reasoning. For each node n corresponding to value in flow-sensitive analysis result, if it can backtrack to a potential source node in the LSEG—including parameter nodes, σ_{src}-accepted nodes, or nodes in Out_{src}— it is added to N_k of the VAG (Line 34). Notably, Out_{src} includes both auxiliary-output nodes whose corresponding auxiliary-return nodes have output summaries (Line 7) and potential source nodes propagated through transfer summaries (Line 18). After adding a potential source node n to N_k,

every step to access n is recorded (Line 49). The constructed VAG is then used to guide the path-sensitive access-path loading (Line 13), ensuring that only relevant value flows are constructed (Line 15). The pruning process is governed by the operator $Load_{AP}(val, off, \tilde{G})$ below. Since nodes in the VAG either represent potential source nodes or nodes that can reach them on the memory layout (as encoded in $\tilde{\mathcal{E}}$), any loaded value that cannot reach a source node can be pruned. This eliminates the need to exhaustively enumerate and reason about all guarded values while loading an access path. Specifically, for an access path $val\text{->}off_1\text{-> }\ldots\text{->}off_n$, at each level off_i, pruning a loaded value v avoids the cost of resolving $v\text{->}off_{i+1}\text{-> }\ldots\text{->}off_n$.

$$Load_{AP}(val, off_1, \ldots, off_n, \tilde{G}) = Load_{AP}(Load_{AP}(val, off_1, \ldots, off_{n-1}),\ off_n, \tilde{G})$$

$$Load_{AP}(val, off, \tilde{G}) = \{(v, \pi) \mid ((v, v'), off) \in \tilde{\mathcal{L}},\ (v',\ \pi) \in Load_{AP}(\{(v, \varphi)\},\ off), (v, \varphi) \in val\}$$

Next, we construct the necessary auxiliary-return nodes to complete the output and transfer summaries for the function f. For each locator recorded in the side-effect summary of f, each of its stored values v serves as an interface that returns the memory layout to the caller. To identify whether such returned memory layout may propagate bug sources, Hermes analyzes the memory layout reachable from each v by using `buildVAGWithoutAP` (Line 22). This function operates similarly to `buildVAGWithAP`, but without indexing by a specific access path. Because symbolic objects and locators can be traversed indefinitely under the rules Load-2 and Load-3, Line 35 and 41 of Algorithm 2 ensure the traversal stops in time. During the traversal, if an auxiliary output v is encountered and there exists a symbol s whose corresponding node belongs to Out_{src} and is reachable from v via an access path, then `buildVAGWithAP` is invoked to construct a complete VAG (Line 39). Otherwise, the traversal can be safely terminated. When auxiliary parameters are encountered, Section 4.3 offers a detailed explanation of why terminating at Line 42 does not compromise correctness. Finally, each access path $(l,\ \tilde{\mathcal{L}}((n_1, n_2)),\ \ldots,\ \tilde{\mathcal{L}}((n_{k-1}, n_k)))$ that may reach a potential source node is loaded in a path-sensitive manner under VAG guidance (Line 25). The node of the loaded value is then connected to the corresponding auxiliary-return node (Line 26-27). This process ensures that all auxiliary-return nodes and value flows that may carry potential bug sources are resolved, enabling the complete construction of output and transfer summaries for f. Subsequently, f can be analyzed using the traditional summary-based bug search (Algorithm 1).

Ultimately, for each source-sink path whose reachability needs to be validated, the path condition comes from the guards of value flows along the source-sink path, as well as the control dependencies of each path node, as described in Definition 3.3. These control dependencies could require data dependencies of unresolved interface nodes. Therefore, Hermes fully resolves them on demand. This process is relatively straightforward, and we illustrate it with an example.

Example 3.6. Consider the code snippet in Figure 1. The function *fun* rewrites *c->deeper* at Line 15 and 17. According to Line 22 of Algorithm 2, Hermes constructs a VAG $(\tilde{\mathcal{N}}_k, \tilde{\mathcal{N}}_a, \tilde{\mathcal{E}}, \tilde{\mathcal{L}})$ with *global* and *d* as starting points. Since the memory pointed to by *d->bool_p* stores *nullptr*—the bug source for NPD—we have $n_{nullptr} \in \tilde{\mathcal{N}}_k$, $n_d \in \tilde{\mathcal{N}}_a$, with an edge $n_d \rightarrow n_{nullptr} \in \tilde{\mathcal{E}}$, and the access label $\tilde{\mathcal{L}}(n_d \rightarrow n_{nullptr}) = bool\_p$. In resolving the auxiliary-return node, the load from *global* is pruned due to VAG guidance, and the node $n_{nullptr}$ flows to the auxiliary-return node ⑯ in Figure 4. During the bug search for the function *bar*, the search starts from the corresponding auxiliary-output node ⑫, reaches a dereference site ⑧, and forms a complete source-sink path. Along this path, a guard constraint is collected—namely, sym_1 represented by the auxiliary-parameter node ⑬. To validate the reachability of the path, Hermes successively resolves its data dependencies: the auxiliary-parameter ⑨, the auxiliary-input node ⑤, and the auxiliary-return node ②. This completes the control dependencies for checking the reachability of the source-sink path.

With the above interface-node resolving rules, each interface node is resolved only when it contributes to either source-sink transfer or source-sink reachability. This design ensures the soundness of bug search while avoiding the redundancy of constructing a complete SVFG.

Detection Equivalence Statement

Given the same source-sink specification τ, Hermes running on the LSEG produces the same bug reports as the SEG-based search under the assumptions listed in Section 4.1.

Proof sketch. We show a bijection between source-sink paths collected by the two searches via path isomorphism that preserves control dependencies, which implies the equality of path conditions. The full development (definitions, lemmas, and proof) is deferred to Appendix, including: (i) path condition preservation under path isomorphism, (ii) path-set isomorphism implies detection equivalence, and (iii) path-set equivalence between SEG and LSEG.

4 Implementation

4.1 Soundness

Since Hermes builds on Falcon, it is soundy [31] in the same sense as Falcon [69] and other prior bug-finding tools [1, 7, 11, 44–46, 56, 57, 65, 66, 68]. Specifically, it shares several standard assumptions with the prior techniques [1, 7, 11, 44–46, 56, 57, 65, 66, 68] as follows.

- Values passed into and out of library calls are conservatively treated as free variables.
- Arrays are handled through their base pointers during pointer analysis.
- Function parameters are assumed not to alias each other initially.
- Recursion is unrolled to avoid the path-explosion problem, as explained in Section 3.1.1.
- Indirect calls are resolved using a Steensgaard-style, flow- and context-insensitive analysis [71], which has been shown to be precise enough for C-like programs in prior work [16].

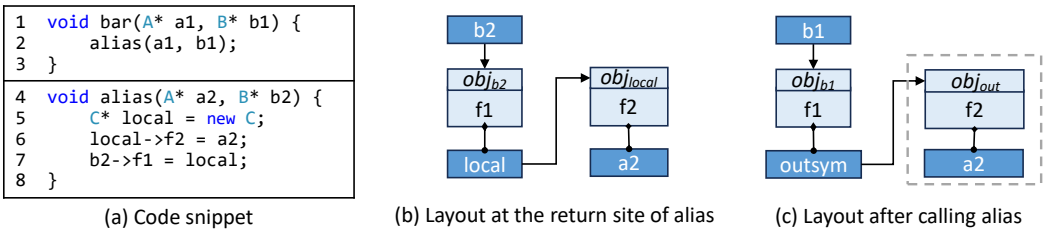
4.2 Checker Instantiation

Similar to prior SVFA-based bug-finding tools (e.g., SVF [55], Pinpoint [45], Falcon [69]), Hermes supports detecting bugs that can be modeled by the source-sink framework [44]. This framework covers two major categories of bugs: *source-must-sink* and *source-must-not-sink*, as formally defined in Definition 3.2. We implement two checkers: one for null pointer dereference (NPD), representing the source-must-not-sink category, and another for memory leak (ML), representing the source-must-sink category. These two bug types are among the most common memory errors in real-world C/C++ programs, and typically elicit prompt responses from developers.

4.3 Alias Processing

In practice, auxiliary returns may alias auxiliary arguments, which complicates the analysis. During `buildVAGWithoutAP` in Algorithm 2, such aliasing can be detected when a parameter symbol $s \in \mathcal{S}_{param}$ is reached via an access path ap. This suggests that the auxiliary output corresponding to ap may alias the input corresponding to the parameter s. Since parameters in $\mathcal{S}_{param}$ can propagate bug sources, the VAG must include their associated nodes in $\tilde{\mathcal{N}}_k$ by definition. However, parameters can be derived indefinitely via any access path, leading to a potentially unbounded traversal when constructing the VAG. Specifically, encountering $s \in \mathcal{S}_{param}$ via ap requires including in the VAG all auxiliary returns corresponding to extended access paths of the form $ap\text{->}f_1\text{->}\ldots\text{->}f_k$, as they may alias $s\text{->}f_1\text{->}\ldots\text{->}f_k$ and thus carry bug sources.

To address this without compromising precision, we explicitly decouple auxiliary parameters from auxiliary returns, ensuring auxiliary outputs abstract only callee-side values. Specifically, at the end of each function's pointer analysis, Hermes traverses the side-effect summary to detect



(a) Code snippet (b) Layout at the return site of alias (c) Layout after calling alias

Fig. 7. A code snippet and memory layouts at key program points illustrating alias processing

auxiliary returns that may alias parameters and records such aliasing. During summary application, this aliasing is explicitly exposed. For example, in Figure 7 (a), function *alias* modifies field *f1* of parameter *b2* (Line 7), which may reach parameter *a2* (Figure 7 (b)). As a result, after Line 2, the auxiliary output *b1->f1->f2* aliases *a1*. To decouple them, HERMES explicitly exposes the aliasing at the callsite: it traverses the memroy layout starting from the locators in side-effect summaries (e.g., the field *f1* of obj_{b2}), and if it reaches a parameter symbol (e.g., *a2*), it records an aliasing triple: (access path, target symbol, constraint), e.g., (*f2*, *a2*, *true*). Then, during summary application, HERMES uses the aliasing triple to proactively allocate symbolic objects and symbols. Following rule CALL, it stores *outsym* into *b1->f1* to simulate the side effect. In addition, based on the recorded aliasing triple (*f2, a2, true*), HERMES first resolves the input value of the target symbol *a2* at the callsite, which is *a1*. Then it exposes the value at the correct memory location by creating obj_{out} and storing *a1* into the field *f2* under the constraint *true*.

As a result, the traversal in Algorithm 2 can safely terminate at Line 42 because all aliasing has been exposed. For clarity, this handling is not formalized in Figure 6, but described separately here.

5 Evaluation

In this section, we aim to evaluate the performance of our approach HERMES by investigating the following research questions:

- **RQ1**: How effective is HERMES in detecting bugs?
- **RQ2**: How efficient is HERMES in detecting bugs?
- **RQ3**: How does the VAG-based pruning benefit the loading of access paths?

5.1 Experimental Setup

5.1.1 Subject Collection. We selected 12 well-known open-source projects, as shown in Table 2. These projects span diverse domains, including database engines, version control systems, MP3 encoders, neural network libraries, and I/O libraries, with sizes ranging from 27K to 4.37M lines of code. Their source code repositories have been hosted in either GitHub or SourceForge, which facilitates easy downloading and bug detection. Moreover, these projects are popular and have been extensively analyzed in prior studies as experimental subjects, indicating their high quality. In addition, Table 2 reports the numbers of source and sink nodes for both NPD and ML queries, characterizing the scale of the bug-search space.

5.1.2 Bug Type Selection. We selected to check Null Pointer Dereference (NPD) and Memory Leak (ML), which represent two distinct and classic bugs types under the source-sink framework: the *source-must-not-sink* and *source-must-sink* bug types, respectively. These two bug checkers are run separately on the same LSEG within a single analysis run.

Table 2. The statistics of projects. **C&V** represents the commit id or version of the project. **Star** represents the number of stars a project has received. **Size** represents the number of lines of code (**KLoC**). **#Src/Sink** reports the numbers of sources and sinks for **NPD** and **ML**. **#BUG** represents the total numbers of **NPD** and **ML** bugs detected by Hermes or Falcon. Each entry in **#BUG** is in the format #FP/#TTL, where #FP denotes the number of false positives, and #TTL denotes the total number of bugs.

Project	C&V	Star	Size	#Src/Sink		#BUG	
				NPD	ML	Hermes	Falcon
htop	68c970c	6.4k	27	591/31,496	5/376	0/4	0/4
lame	3.100	-	29	231/55,731	34/88	0/1	0/0
memcached	b1aefcd	13.5k	31	623/33,219	89/291	1/2	0/1
darknet	d02cc3a	25.8k	37	536/98,539	55/461	0/6	0/4
libuv	5cc7175	24.1k	62	603/29,230	27/139	0/1	0/1
dynamips	ab50526	352	83	2,526/123,512	220/791	0/5	0/4
openldap	5b5337b	494	164	5,947/242,520	33/1,287	0/1	0/1
ppsspp	16d97aa	11.2k	389	9,102/631,770	2,503/32,649	0/5	0/3
git	477ce5c	52.2k	430	9,527/422,494	19/3,245	2/5	2/3
binutils	a9d9a10	572	699	12,028/762,640	153/10,830	3/17	2/14
MariaDB	2db692f	5.9k	1,813	28,450/1,217,747	559/23,083	2/5	0/0
wine	3364df0	3.3k	4,374	20,552/1,438,398	2,018/13,947	1/4	1/4
Total				90,716/5,087,296	5,715/87,187	9/56	5/39

5.1.3 *Baseline.*

- Falcon: The state-of-the-art sparse-value flow analysis technique [69], which is also the basis of our implementation.
- Hermes^{-}: A variant of Hermes that disables VAG-based pruning when loading access paths, aiming to evaluate the impact of pruning during $Load_{AP}$ operations.

5.1.4 Environment. All experiments were run on an Ubuntu 20.04 server with an Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz 40-core processor and 512G of memory. Ten threads were allocated for running. Following the practice of the prior studies [64], we set a 12-hour timeout period, consistent with the typical nightly build duration.

5.2 RQ1: Effectiveness

In our experiments, Hermes detected a total of 56 bugs, including 49 NPD bugs and 7 ML bugs, as shown in the **#BUG** column of Table 2. This result is consistent with previous studies [64], which indicate that NPD is the most prevalent type of memory bug. Through manual inspection, we found that 9 of the reported bugs were false positives, resulting in a false positive rate of 16.07% (9/56). To avoid overwhelming developers with excessive warnings, we merged reports with similar source-sink paths before submitting them. Of the 47 submitted reports, 30 have been confirmed, and the rest are awaiting developer response.

Across 12 projects, Hermes outperformed Falcon in our evaluation: it detected all bugs found by Falcon and additionally uncovered 17 more, including 13 true positives and 4 false positives.

The relatively lower effectiveness of Falcon can be attributed to two reasons. First, Falcon hit the 12-hour timeout and failed to complete pointer analysis for MariaDB, causing it to miss five bugs that Hermes detected in that project. Second, after consulting with the developers of Falcon, we learned that to mitigate the summary-explosion problem in pointer analysis, Falcon imposes a limit on the number of auxiliary returns each function can produce. During summary application, it only copies the memory layouts associated with this limited set of auxiliary returns. Although such handling improves scalability, it sacrifices recall, causing Falcon to detect fewer bugs than Hermes. Taken together, Falcon and Hermes only exhibit equivalent effectiveness when sufficient

Table 3. Time and memory comparison between Hermes and Falcon. **PTA**, **BS**, **TTL**, and **MEM** denote pointer analysis time, bug search time, total time, and memory usage, respectively. **TS** is the time speedup of Hermes over Falcon. **OOT** means timeout (12h). **Average** is calculated using the geometric mean.

Project	Falcon				Hermes						
	PTA	BS	TTL	MEM	PTA	TS_{PTA}	BS	TS_{BS}	TTL	TS_{TTL}	MEM
htop	19s	1.57m	1.88m	2.54G	5.26s	3.61	20.74s	4.54	26s	4.34	1.79G
lame	13s	2.05m	2.27m	2.43G	8.4s	1.55	44.6s	2.76	53s	2.57	2.19G
memcached	25s	2.07m	2.48m	2.16G	6.64s	3.77	28.36s	4.38	35s	4.25	1.70G
darknet	35s	2.13m	2.72m	3.50G	14.22s	2.46	58.78s	2.17	1.22m	2.23	2.83G
libuv	21s	1.77m	2.12m	1.87G	2.31s	9.09	24.69s	4.30	27s	4.71	1.84G
dynamips	1.52m	8.60m	10.12m	4.76G	15.26s	5.98	1.35m	6.37	1.58m	6.41	3.67G
openldap	8.85m	13.88m	22.73m	17.94G	41.18s	12.89	1.68m	8.26	2.37m	9.59	9.10G
ppsspp	1.30h	1.36h	2.66h	56.70G	4.07m	19.16	12.85m	6.35	16.92m	9.43	49.75G
git	1.19h	1.40h	2.60h	62.30G	3.22m	22.17	7.48m	11.39	10.70m	14.58	41.13G
binutils	3.92h	48.98m	4.74h	37.30G	2.68m	87.76	15.11m	3.24	17.67m	16.10	24.00G
MariaDB	OOT	NA	OOT	65.16G	16.87m	>42.68	35.35m	NA	52.02m	>13.84	84.79G
wine	3.01h	2.56h	5.57h	109.06G	5.48m	32.96	26.55m	5.79	32.03m	10.43	83.47G
Average						>9.84		4.79		>6.80	

resources are available. However, under practical resource constraints, Falcon's lower efficiency leads to reduced effectiveness in practice.

We further investigated the false positives reported by Hermes. One false positive in the ML category was caused by the lack of modeling for third-party library functions. When a tracked heap object is passed into a library function, the checker aggressively assumes that the library does not free it. The remaining eight false positives in the NPD category were due to the imprecise modeling of containers, such as arrays and lists—an inherited limitation of the underlying SVFA framework. These limitations are orthogonal to our technique and could be addressed by enhancing the underlying SVFA framework.

5.3 RQ2: Efficiency

Due to the symbolization scheme and the lazy node resolving introduced in section 3, Hermes' primary contribution lies in improving the efficiency of bug detection. The experimental results listed in Table 3 demonstrate that Hermes enhances not only the efficiency of pointer analysis but also the efficiency of bug search, ultimately enhancing the overall efficiency.

Specifically, compared to Falcon, Hermes achieves varying degrees of speedup across different phases of all evaluated projects. On average, the speedup achieved by Hermes in pointer analysis, bug search, and overall execution is 9.84×, 4.79× and 6.80×, respectively. The speedup in pointer analysis increases as the codebase grows, reaching a maximum of 87.76× in binutils. Additionally, Hermes completes the pointer analysis of each project within an hour, whereas Falcon struggles with large projects, even timing out during pointer analysis on MariaDB.

For the speedup in bug search, we found that it primarily stems from two aspects. First, Hermes constructs more concise inter-procedural value flows for source-sink transfer. Consider the *for* loop at Line 8 of Algorithm 2—Hermes only constructs an inter-procedural value flow from a source node to a callee input node if the input node contributes to propagating the source node to a sink site. In contrast, Falcon establishes an inter-procedural value flow whenever a value within the callee has a data dependence on the value of a source node. Consequently, Falcon explores more paths when searching from a source node. Even though the path is short, only from the source node to the input node, Falcon must invoke an SMT solver to prune infeasible paths, leading to higher computational overhead in bug search. Second, Falcon incurs significant overhead when handling auxiliary-return nodes. At the end of the bug search for each function f, Falcon collects all auxiliary-return nodes' data dependencies and encodes them into SMT expressions, which are

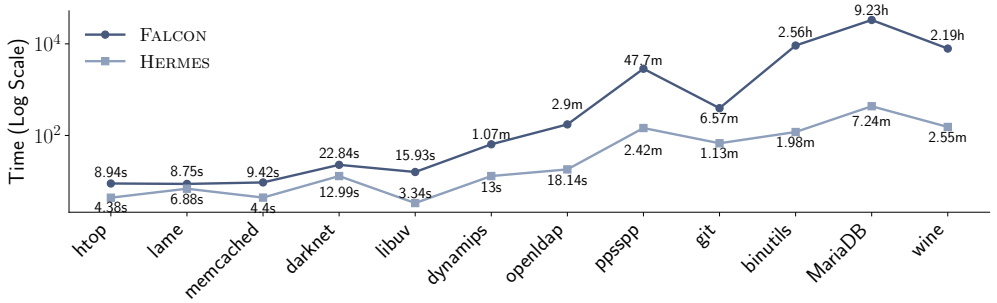


Fig. 8. Comparison of load instruction processing time (s) between Hermes and Falcon.

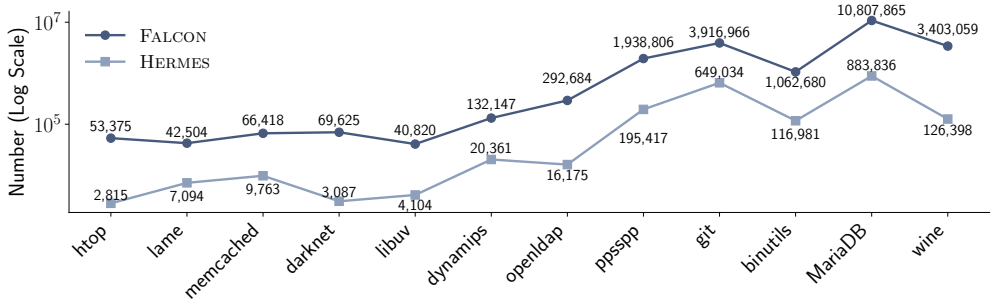


Fig. 9. Comparison of the number of resolved interface nodes between Hermes and Falcon.

used for reachability verification in the subsequent bug search for caller functions of *f*. This step is expensive because an auxiliary-return node may depend on the auxiliary-return node of the callee functions of *f*. As a result, it requires instantiating the corresponding return node's SMT expressions, essentially performing a deep copy of these expressions. In contrast, Hermes constructs only the auxiliary-return nodes essential for source-sink transfer and reachability. This significantly reduces the number of return nodes whose SMT expressions need to be collected. As a result, even without optimizing the searching process, Hermes achieves considerable speedup in bug search.

Through further investigation, we found that Hermes requires less time to process load instructions during pointer analysis, and this difference becomes more pronounced as the project scale increases. In the evaluation of binutils, MariaDB, and wine projects, Hermes only takes 1.28%, at least 1.31%, and 1.94% of the time taken by Falcon, respectively, as shown in Figure 8. This is primarily due to Falcon's pointer analysis summary, which copies the memory layout of the callee functions. As a result, the loaded pointer in Falcon points to a more complex memory structure, leading to slower load times. In contrast, Hermes' symbolization scheme avoids copying the memory layout, allowing it to process load statements more efficiently. Furthermore, Falcon resolves all interface nodes during pointer analysis, while Hermes only resolves the necessary interface nodes. Figure 9 shows the number of interface nodes resolved by Falcon and Hermes, respectively. On average, Hermes processes only 9.45% as many interface nodes as Falcon. Since resolving an interface node essentially involves loading an access path, and given the previously discussed reasons, Hermes processes load operations more efficiently, accelerating interface node resolving, averaging only 5.15% of the pointer analysis time.

There is little difference between Hermes and Falcon in terms of memory consumption. The primary reason is that memory consumption is mainly determined by the construction of the path condition node and the storage of SMT expressions. Thus, although Hermes generates fewer points-to relations, it does not significantly affect the memory usage.

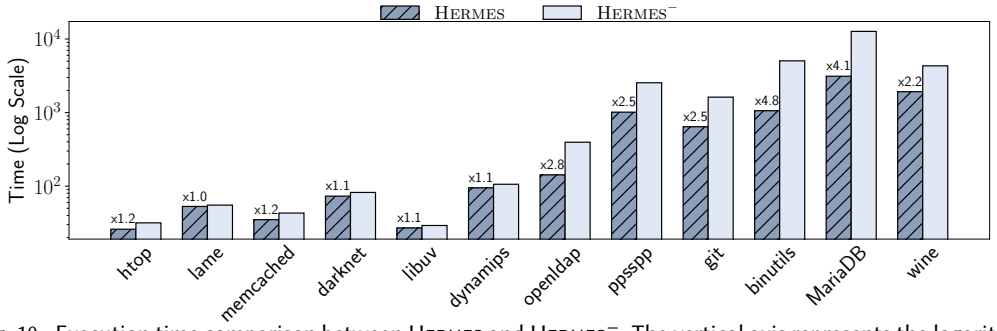


Fig. 10. Execution time comparison between Hermes and Hermes⁻. The vertical axis represents the logarithm of time in seconds.

5.4 RQ3: Ablation Study

To evaluate the benefits of VAG-based pruning for loading access paths, we implemented Hermes⁻, which follows the same pointer analysis and bug search design as Hermes, but without constructing or using VAG for pruning. Specifically, it does not execute the *buildVAGWithAP* and *buildVAGWithoutAP* in Algorithm 2, and replaces $Load_{AP}(val, off_1, ..., off_n, G)$ defined in Section 3.3.3 with $Load_{AP}(val, off_1, ..., off_n)$ defined in Section 3.1.2.

Figure 10 shows the execution time comparison between Hermes and Hermes⁻, as well as the speedup of Hermes over Hermes⁻. No significant difference in memory consumption was observed, so those results are omitted. It can be observed that Hermes consistently outperforms Hermes⁻ in all the projects, with an average speedup of 1.85×. When the code size is small, the speedup is negligible. However, as the code size increases, the speedup becomes more significant. This is because, in large-scale projects, the side effects of call instructions become increasingly numerous and complex. In pointer analysis, this is reflected in the fact that a certain memory location may be modified multiple times by callee functions under different conditions, with increasingly complex conditions and more frequent modifications. When the depth of the access path increases, loading it typically involves enumerating all disjunction cases and solving the corresponding path conditions, which is very time-consuming, as described in Section 2.2. With the VAG-based pruning, the disjunction cases unrelated to source-sink transfer can be efficiently pruned, thereby ensuring scalability in large-scale projects.

6 Related Work

6.1 Pointer Analysis

Hermes performs pointer analysis that propagates points-to information sparsely and calculates points-to results on demand based on inter-procedural value-flow-construction queries. Thus, we separately discuss sparse pointer analysis and demand-driven pointer analysis.

Sparse Pointer Analysis. To improve scalability, this category of pointer analysis techniques performs sparse propagation of points-to information, thereby avoiding propagation along control-flow edges. These approaches generally fall into two categories.

The first category [15, 53, 54, 68] follows the refinement strategy: first perform an imprecise but scalable pointer analysis to build coarse-grained def-use chains, ensuring that points-to information can be sparsely propagated. However, these imprecise def-use chains often introduce a large amount of spurious and redundant propagation, thus hurting scalability.

The second category [27, 58, 70] adopts on-the-fly strategies, where the def-use chains are constructed incrementally along with the pointer analysis. Most of these methods are path-insensitive. As for path-sensitive methods, SPAS [58] encodes path conditions by Binary Decision Diagram (BDD) to eliminate mutually exclusive paths efficiently. However, the points-to information computation remains exhaustive, which significantly limits scalability. Falcon [69] improves upon these by employing symbolic representations to isolate inter-procedural points-to propagation, deferring the resolution of symbolic information to client analyses. This design achieves better scalability, and Hermes adopts this fundamental idea. However, Falcon's approach still performs exhaustive propagation of bottom-up points-to information by copying entire memory layouts, which results in the summary-explosion problem. Hermes addresses this limitation through a novel symbolization scheme that avoids unnecessary copying.

Demand-Driven Pointer Analysis. Most existing on-demand pointer analyses [12, 19, 36, 41, 42, 50–52, 67, 72] are flow-insensitive. They typically rely on structures that lose control-flow information, such as pointer expression graphs. For the flow-sensitive methods, Sui et al. [54, 68] leverage the refinement-based strategy to construct scalable but imprecise value flows. This enables sparse and on-demand backtracking, while pruning spurious edges during query resolution. However, the analysis still suffers from the aliasing-path-explosion problem. Falcon also incorporates an on-demand resolution of points-to information. However, Falcon requires to construct all inter-procedural value flows, which limits its effectiveness when analyzing codebases exceeding millions of lines of code.

Additionally, extensive research on Java pointer analysis has focused on demand-driven context length decision [17, 18, 20–23, 26, 28–30, 34, 35, 48, 60], commonly known as selective context-sensitive pointer analysis. These approaches aim to allocate more computational resources to important functions. To achieve this, the decision procedures often employ heuristic tactics [17, 48], data-driven tactics [21, 23], or hybrid tactics [60] that combine multiple decision procedures. These analyses are typically flow-insensitive.

6.2 SVFA-Based Bug Detection

Current SVFG-based bug detection approaches can be broadly categorized into two types. The first category [7–9, 32, 40, 49, 55, 57] adopts a layered design, where pointer analysis, SVFG construction, and bug search are performed independently. These approaches cannot achieve both path-sensitive precision and scalability on million-line codebases during the pointer analysis phase, which ultimately impacts the overall detection effectiveness.

The second category [45] employs a holistic design that avoids building global value flows in SVFG. Instead, these approaches construct only the inter-procedural value flows between a caller and callee at each callsite. During the bug-search phase, they stitch the relevant global value flows and solve the path constraints. By restricting inter-procedural value flows to each direct caller-callee pair, these methods alleviate the burden on pointer analysis and enable path-sensitive precision.

Despite the differences in SVFG instances among the above approaches, they all construct a complete SVFG, which places an unnecessary burden on pointer analysis. In contrast, our approach lazily constructs only the inter-procedural value flows relevant to bug search, significantly improving scalability.

7 Conclusion

In this paper, we introduced Hermes, an innovative approach addressing scalability in path-sensitive pointer analysis within Sparse Value-Flow Analysis (SVFA). Hermes uses a novel symbolization scheme and selectively resolves value flows to efficiently reduce computational overhead. Our evaluations confirm that Hermes achieves a substantial speedup (at least 9.84× for pointer analysis, 4.79×

for bug search) without sacrificing the effectiveness of bug detection. Overall, Hermes significantly enhances performance and scalability for detecting software security bugs in large codebases.

Data Availability

All artifacts related to this work are publicly available at the following GitHub repository: https://github.com/SoulBoyHere/hermes-data-availability. Specifically, the repository includes: (1) a list of all reported bugs along with their validation status; (2) a formal proof document (`proof.pdf`) establishing the detection equivalence between Hermes and the baseline approach; and (3) detailed instructions for obtaining and running a pre-built Docker image containing the compiled binary of Hermes and the complete experimental dataset. Due to intellectual property agreements with our industry partner, the source code of Hermes cannot be made publicly available.

Acknowledgement

We sincerely thank the anonymous reviewers for their valuable and insightful feedback. This research was supported by the Natural Science Foundation of China (Grant No. 62272400) and Fujian Provincial Natural Science Foundation of China (Grant No. 2025J010002). Rongxin Wu is the corresponding author and works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University.

References

[1] Domagoj Babic and Alan J. Hu. 2008. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 211–220. https://doi.org/10.1145/1368088.1368118

[2] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: practical type-based null safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE '19)*. 740–750. https://doi.org/10.1145/3338906.3338919

[3] Jeffrey M. Barth. 1978. A practical interprocedural data flow analysis algorithm. *Commun. ACM* 21, 9 (Sept. 1978), 724–736. https://doi.org/10.1145/359588.359596

[4] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. 275–286. https://doi.org/10.1145/2491956.2462186

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI '08)*. 209–224.

[6] Yuandao Cai, Chengfeng Ye, Qingkai Shi, and Charles Zhang. 2022. Peahen: fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE '22)*. 784–796. https://doi.org/10.1145/3540250.3549110

[7] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. 480–491. https://doi.org/10.1145/1250734.1250789

[8] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. 57–68. https://doi.org/10.1145/512529.512538

[9] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA) *(ISSTA '04)*. 12–22. https://doi.org/10.1145/1007512.1007515

[10] G.W. Ernst and R.B. Dannenberg. 1982. Formal Program Verification Using Symbolic Execution . *IEEE Transactions on Software Engineering* 8, 01 (Jan. 1982), 43–52. https://doi.org/10.1109/TSE.1982.234773

[11] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. 72–82. https://doi.org/10.1109/ICSE.2019.00025

[12] Yu Feng, Xinyu Wang, Isil Dillig, and Calvin Lin. 2015. EXPLORER: query- and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA '15)*. 520–534. https://doi.org/10.1145/2814270.2814284

[13] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

[14] Yiyuan Guo, Jinguo Zhou, Peisen Yao, Qingkai Shi, and Charles Zhang. 2022. Precise divide-by-zero detection with affirmative evidence. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. 1718–1729. https://doi.org/10.1145/3510003.3510066

[15] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. 289–298.

[16] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 289–298.

[17] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP '17)*. 13–18. https://doi.org/10.1145/3088515.3088519

[18] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2023. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis . *IEEE Transactions on Software Engineering* 49, 02 (Feb. 2023), 719–742. https://doi.org/10.1109/TSE.2022.3162236

[19] Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. 24–34. https://doi.org/10.1145/378795.378802

[20] Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 13 (June 2019), 41 pages. https://doi.org/10.1145/3293607

[21] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276510

[22] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428247

[23] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133924

[24] Vineet Kahlon. 2008. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. 249–259. https://doi.org/10.1145/1375581.1375613

[25] John B. Kam and Jeffrey D. Ullman. 1977. Monotone data flow analysis frameworks. *Acta Inf.* 7, 3 (Sept. 1977), 305–317. https://doi.org/10.1007/BF00290339

[26] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. 423–434. https://doi.org/10.1145/2491956.2462191

[27] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. 343–353. https://doi.org/10.1145/2025113.2025160

[28] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276511

[29] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE '18)*. 129–140. https://doi.org/10.1145/3236024.3236041

[30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 10 (May 2020), 40 pages. https://doi.org/10.1145/3381915

[31] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a

manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[32] V. Benjamin Livshits and Monica S. Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Helsinki, Finland) *(ESEC/FSE '11)*. 317–326. https://doi.org/10.1145/940071.940114

[33] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. 2008. Verifying dereference safety via expanding-scope analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. 213–224. https://doi.org/10.1145/1390630.1390657

[34] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA) *(SAS '21)*. 261–285. https://doi.org/10.1007/978-3-030-88806-0_13

[35] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

[36] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An incremental points-to analysis with CFL-Reachability. In *Proceedings of the 22nd International Conference on Compiler Construction* (Rome, Italy) *(CC '13)*. 61–81. https://doi.org/10.1007/978-3-642-37051-9_4

[37] Sen Ma, MingYang Jiao, ShiKun Zhang, Wen Zhao, and Dong Wei Wang. 2015. Practical null pointer dereference detection via value-dependence analysis. In *Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (ISSREW '15)*. 70–77. https://doi.org/10.1109/ISSREW.2015.7392049

[38] Ravichandhran Madhavan and Raghavan Komondoor. 2011. Null dereference verification via over-approximated weakest pre-conditions analysis. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. 1033–1052. https://doi.org/10.1145/2048066.2048144

[39] Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate Interprocedural Null-Dereference Analysis for Java. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. 133–143. https://doi.org/10.1109/ICSE.2009.5070515

[40] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. 229–238. https://doi.org/10.1145/2254064.2254092

[41] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. 117–128. https://doi.org/10.1145/1069774.1069785

[42] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. 264–274. https://doi.org/10.1145/2259016.2259050

[43] Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. 2022. Indexing the extended Dyck-CFL reachability for context-sensitive program analysis. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 176 (Oct. 2022), 31 pages. https://doi.org/10.1145/3563339

[44] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the extensional scalability problem for value-flow analysis frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. 812–823. https://doi.org/10.1145/3377811.3380346

[45] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI '18)*. 693–706. https://doi.org/10.1145/3192366.3192418

[46] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI '21)*. 930–943. https://doi.org/10.1145/3453483.3454086

[47] Qingkai Shi and Charles Zhang. 2020. Pipelining bottom-up data flow analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. 835–847. https://doi.org/10.1145/3377811.3380425

[48] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 485–495. https://doi.org/10.1145/2594291.2594320

[49] Gregor Snelting, Torsten Robschink, and Jens Krinke. 2006. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.* 15, 4 (Oct. 2006), 410–457. https://doi.org/10.1145/1178625.1178628

[50] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. 387–400. https://doi.org/10.1145/1133981.1134027

[51] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. 59–76. https://doi.org/10.1145/1094811.1094817

[52] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems (BRACIS '14)*. 451–460. https://doi.org/10.1109/ICPP.2014.54

[53] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. 160–170. https://doi.org/10.1145/2854038.2854043

[54] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE '16)*. 460–473. https://doi.org/10.1145/2950290.2950296

[55] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. 265–266. https://doi.org/10.1145/2892208.2892235

[56] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA '12)*. 254–264. https://doi.org/10.1145/2338965.2336784

[57] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis . *IEEE Transactions on Software Engineering* 40, 02 (Feb. 2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

[58] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems* (Kenting, Taiwan) *(APLAS '11)*. 155–171. https://doi.org/10.1007/978-3-642-25318-8_14

[59] Yi Sun, Chengpeng Wang, Gang Fan, Qingkai Shi, and Xiangyu Zhang. 2024. Fast and Precise Static Null Exception Analysis With Synergistic Preprocessing . *IEEE Transactions on Software Engineering* 50, 11 (Nov. 2024), 3022–3036. https://doi.org/10.1109/TSE.2024.3466551

[60] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (Oct. 2021), 27 pages. https://doi.org/10.1145/3485524

[61] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. 350–360. https://doi.org/10.1145/3180155.3180251

[62] Chengpeng Wang, Wenyang Wang, Peisen Yao, Qingkai Shi, Jinguo Zhou, Xiao Xiao, and Charles Zhang. 2023. Anchor: Fast and Precise Value-flow Analysis for Containers via Memory Orientation. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 66 (April 2023), 39 pages. https://doi.org/10.1145/3565800

[63] Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned memory models for program analysis. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI '17, Proceedings*. 539–558. https://doi.org/10.1007/978-3-319-52234-0_29

[64] Rongxin Wu, Yuxuan He, Jiafeng Huang, Chengpeng Wang, Wensheng Tang, Qingkai Shi, Xiao Xiao, and Charles Zhang. 2024. LibAlchemy: A Two-Layer Persistent Summary Design for Taming Third-Party Libraries in Static Bug-Finding Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Article 105, 13 pages. https://doi.org/10.1145/3597503.3639132

[65] Yichen Xie and Alex Aiken. 2005. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/1040305.1040334

[66] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. 512–523. https://doi.org/10.1109/ICSE.2019.00063

[67] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. 155–165. https://doi.org/10.1145/2001420.2001440

[68] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. 327–337. https://doi.org/10.1145/3180155.3180178

[69] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. *Proc. ACM Program. Lang.* 8, PLDI, Article 170 (June 2024), 26 pages. https://doi.org/10.1145/3656400

[70] 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* (Toronto, Ontario, Canada) *(CGO '10)*. 218–229. https://doi.org/10.1145/1772954.1772985

[71] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 435–446. https://doi.org/10.1145/2491956.2462159

[72] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. 197–208. https://doi.org/10.1145/1328438.1328464