Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions

WENSHENG TANG*, The Hong Kong University of Science and Technology, China
DEJUN DONG and SHIJIE LI, Ant Group, China
CHENGPENG WANG†, The Hong Kong University of Science and Technology, China
PEISEN YAO, Zhejiang University, China
JINGUO ZHOU, Ant Group, China
CHARLES ZHANG, The Hong Kong University of Science and Technology, China

Value-flow analysis is a fundamental technique in program analysis, benefiting various clients, such as memory corruption detection and taint analysis. However, existing efforts suffer from the low potential speedup that leads to a deficiency in scalability. In this work, we present a parallel algorithm Octopus to collect path conditions for realizable paths efficiently. Octopus builds on the realizability decomposition to collect the intraprocedural path conditions of different functions simultaneously on-demand and obtain realizable path conditions by concatenation, which achieves a high potential speedup in parallelization. We implement Octopus as a tool and evaluate it over 15 real-world programs. The experiment shows that Octopus significantly outperforms the state-of-the-art algorithms. Particularly, it detects NPD bugs for the project llvm with 6.3 MLoC within 6.9 minutes under the 40-thread setting. We also state and prove several theorems to demonstrate the soundness, completeness, and high potential speedup of Octopus. Our empirical and theoretical results demonstrate the great potential of Octopus in supporting various program analysis clients. The implementation has officially deployed at Ant Group, scaling the nightly code scan for massive FinTech applications.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Software safety**.

Additional Key Words and Phrases: Value-flow analysis, parallel computation

ACM Reference Format:
Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Zhang. 2023. Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (February 2023), 32 pages. https://doi.org/xxxxxx/xxxxxx

*Apart from the first and last authors, the remaining authors are listed alphabetically.
†Corresponding author

The authors are supported by the RGC16206517, ITS/440/18FP, and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei.
Authors' addresses: Wensheng Tang, The Hong Kong University of Science and Technology, China, wtangae@cse.ust.hk; Dejun Dong, dejun.ddj@antgroup.com; Shijie Li, Ant Group, China, lishijie.lsj@antgroup.com; Chengpeng Wang, The Hong Kong University of Science and Technology, China, cwangch@cse.ust.hk; Peisen Yao, Zhejiang University, China, pyaoaa@zju.edu.hk; Jinguo Zhou, Ant Group, China, jinguo.zjg@antgroup.com; Charles Zhang, The Hong Kong University of Science and Technology, China, charlesz@cse.ust.hk.

1049-331X/2023/2-ART1 $15.00
https://doi.org/xxxxxx/xxxxxx

1 INTRODUCTION

Value-flow analysis aims to identify whether a value constructed at a program location can flow to a destination location [11, 32, 53, 62]. The analysis of value flows underpins a wide range of applications, such as value-flow bug detection [11, 17, 65], program diagnosis [19, 20, 75], and automated program repair [24]. For example, to detect NULL-pointer-dereference (NPD) bugs, we need to track whether a `NULL` value assigned to a pointer can flow to a pointer dereference statement.

To improve the analysis precision, most existing efforts attempt to achieve context sensitivity to constrain the propagation of values. Specifically, they instantiate the context-sensitivity via the extended Dyck-CFL-reachability [47, 51, 76] over specific graphs (e.g., the super-graph [50] and the program-valid graph [10]). A vertex s is *extended Dyck-CFL-reachable* from a vertex t if and only if there exists a path from s to t in the graph, and the *realized string* of the path formed by the edge labels belongs to the extended Dyck-CFL [51, 76]. Such a path is referred to be *realizable*, indicating that the value flows respect the calling context, which is represented by its realized string. In addition, a realizable path is *feasible* only if its path condition is satisfiable, meaning program execution following such a value flow can happen. Existing analyses determine the feasibility by collecting the conditions of realizable paths between s and t and examining the satisfiability of the logical disjunction of the conditions, which plays a central role in various clients, e.g., value-flow bug detection.

Goals and Challenges. In this paper, we work on context- and path-sensitive value-flow analysis formulated over the guarded value-flow graph [33, 53, 62, 65, 72]. Specifically, we consider the problem of *efficient collection of realizable path condition*: given a source s and a sink t of interest, we aim to efficiently obtain the path conditions of the realizable paths between s and t. After efficiently collecting the path conditions between each s-t pair, we can invoke a solver to determine the feasibility between each s and t. However, this task is challenging as determining realizability and collecting path conditions require different information. Determining realizability between s and t requires transitively collecting realized strings during analysis using an $O(n^3)$ tabulation-based algorithm [48], which presents a complexity barrier for theoretical optimizations [13, 75]. Additionally, collecting path conditions on every realizable path for each s-t pair on the guarded value-flow graph cannot be piggybacked with existing tabulation-based algorithms. Achieving both goals require non-trivial modification to the original algorithms and leads to higher computational complexity, necessitating a more efficient universal algorithm.

Existing Effort. Existing approaches mainly optimize the scalability in the context of accelerating context-sensitive analysis, and, thereby, the overall efficiency of the analysis beyond context sensitivity could be improved. In particular, they propose parallel versions of tabulation-based approaches [1, 39, 55] by identifying the independence of caller-callee relations in a program. During the reasoning of realizability, the path conditions are also encapsulated in the function summaries. Although some functions can be analyzed in parallel, many still have strong dependencies on the ones of its callees, resulting in the *imbalance* in parallelization and further limiting the potential speedup and achieving satisfactory scalability. A recent parallel analyzer, named Coyote, relaxes the dependencies to improve parallelism while it is still distant from reaching the pleasant speedup to achieve satisfactory scalability [55] because there still exist some caller-callee dependencies that cannot be relaxed. Particularly, when the call graph of a program is close to a long chain, Coyote can only generate the summaries of a restricted number of functions at the same time, thus limiting the parallelism brought by Coyote. Besides, there is a line of previous studies targeting the optimization of the path condition collection via slicing [53, 54]. However, they rely on a parallel or serial version of the context-sensitive analysis to collect realizable value-flow paths, limiting their scalability significantly.

Insight and Solution. In this work, we make no claims of breakthroughs to the innate complexity barrier of context-sensitive analysis or the inherent scalability issues of SMT solving. Instead, we aim to scale up the analysis via parallelism but more efficiently. Our idea comes from two key observations.

- First, any realizable path condition comprises intraprocedural path conditions. Consider the program in Figure 1a and its guarded value-flow graph in Figure 1b as an example. The realizable path condition from $\texttt{NULL}_2$ to b_6 take path conditions of two interprocedural paths: $(\phi_1 \wedge \phi_{bar(a)} \dots) \vee (\neg\phi_1 \wedge \phi_{baz(a)} \dots)$ The final path condition is the logical disjunction of the path conditions of the intraprocedural paths across functions foo and bar.
- Second, the realizability of an interprocedural value-flow path does not depend on that of its intraprocedural counterparts. Instead, it only relates to the labeled edges across callers and callees. In Figure 1b, the realizability relies solely on the edges induced by the call site, e.g., $a_3 \xrightarrow{(_3} p_8$, $f_{10} \xrightarrow{)_3} b_3$.

According to the observations, our key insight is to decouple intraprocedural path condition collection from the realizability reasoning. To enable such decoupling, we propose a novel concept named the *realizability decomposition*, which is achieved by the following three aspects. First, we introduce the notion of the *value-flow segment* as the minimum unit that summarizes intraprocedural value-flow paths, which guides the intraprocedural path condition collection. Second, we present a new graph representation, namely the *value-flow segment graph*, to preserve the capability of identifying the realizable paths, further enabling us to construct the realizable path condition by composing intraprocedural path conditions. Third, the realizability of value-flow segments on the value-flow segment graph foresees possible ways to concatenate intraprocedural paths into interprocedural ones, enabling us to collect realizable path conditions efficiently. Based on the abstraction, we present a parallel algorithm named Octopus, which consists of three parts:

- *Value-flow segment graph generation*: We traverse the guarded value-flow graph of each function and construct a value-flow segment graph in parallel, where each segment summarizes the intra-procedural value-flow paths. For example, ❶ in Figure 1b is a value-flow segment that summarizes all intraprocedural paths from $\texttt{NULL}_2$ to a_3. All segments and edges taken from the guarded value-flow graph in Figure 1b constitute a value-flow segment graph in Figure 1c.
- *Realizable segment path search*: We present a parallel algorithm to find the realizable segment paths on the value-flow segment graph. For example, the path ❶→❹→❽ in Figure 1c is a realizable segment path, which compactly summarizes a series of realizable value-flow paths.
- *Path condition collection*: We recover path conditions from the realizable segment paths by exploring each intraprocedural segment on the guarded value-flow graph. For example, we recover conditions for ❶, ❷, ❹, ❻, and ❽ all in parallel, then compose them to obtain the realizable path conditions. Eventually, we obtain the path conditions $(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge (\neg\phi_4 \vee (\phi_3 \wedge \phi_4)))$ for the source-sink pair $(\texttt{NULL}_2, b_6)$.

Once the realizable path conditions are obtained, we can invoke a solver to determine the satisfiability. Notably, our algorithm is less vulnerable to the *imbalance issue* that tabulation-based approaches (e.g., [49, 55]) are prone to for the following reasons. First, searching paths on a value-flow segment graph foresees the underlying realizable paths and avoids redundant intraprocedural path condition collection. Second, each decoupled task can be easily parallelized in balance without being constrained by the caller-callee boundary, achieving a high potential speedup.

We implement Octopus as a tool and evaluate it over 15 real-world programs. It is shown that Octopus achieves the highest speedup on a 40-core workstation. On average, it achieves 123.1× speedup to the state-of-the-art (SOTA) parallel value-flow analyzer Coyote under the 40-thread setting. Besides, the mostly-linear speedup is observed when providing more threads. The evaluation also shows that Octopus occupies almost all CPU cores during the whole analysis process. We also estimate and prove the lower bound of the potential speedup of each of the three subtasks theoretically. The Octopus's implementation is officially landed at Ant Group, dramatically accelerating the nightly code scan of FinTech applications. To summarize, this paper makes the following contributions:

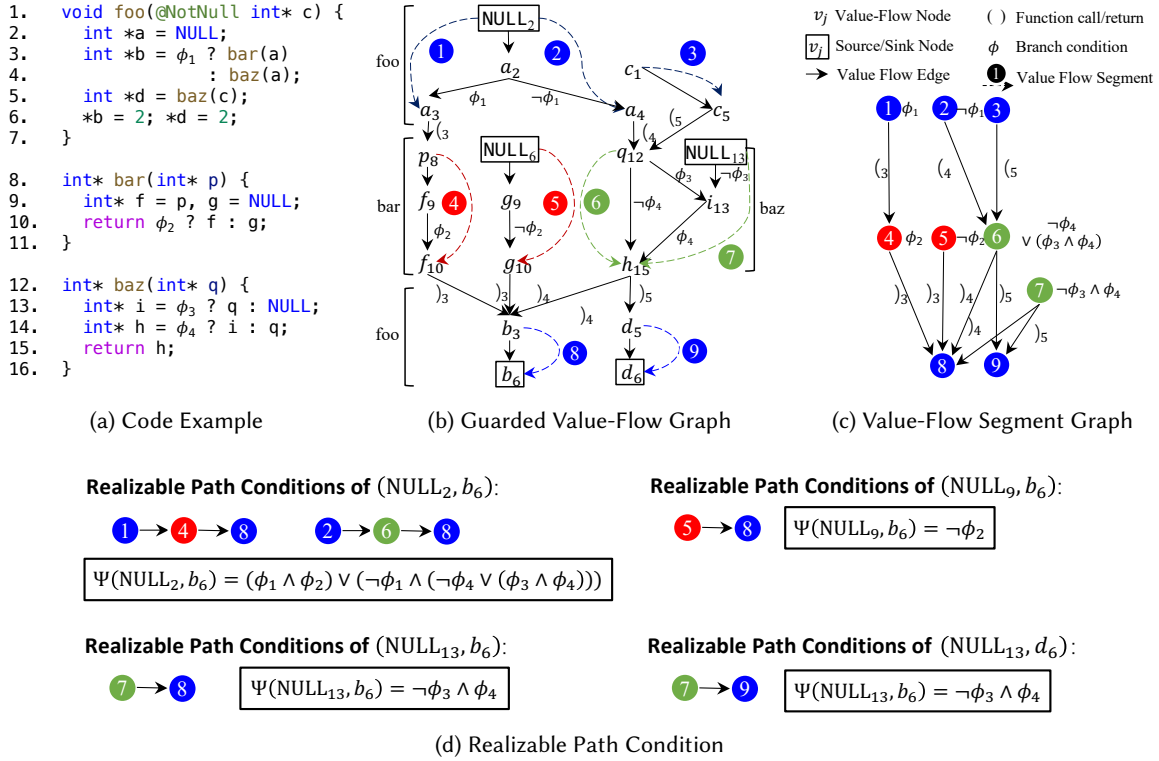


Fig. 1. A motivating example of Octopus. To simplify the example, we omit the caller of the function foo, which will never pass a NULL value to the parameter c at line 1, whose notation is following [21], where the black solid arrows ("⟶") represent value-flow edges, dashed curved arrows ("⇢") represent value-flow segments, and text enclosed in a box ("v_i") denotes the realizable path conditions between two nodes.

- We propose a novel concept of the realizability decomposition and the value-flow segment graph to support the efficient parallelization of realizable path condition collection.
- We design a parallel algorithm to collect realizable path conditions with high potential speedup, benefiting various clients, such as NPD detection and taint analysis.
- We implement our approach as a tool named Octopus, which outperforms the SOTA parallel value-flow analysis, accelerating the NPD detection by 123.1× on average with 40 physical cores.
- We sketch the proofs for the soundness and completeness of our approach and estimate the lower bound of the potential speedup theoretically.

2 OCTOPUS IN A NUTSHELL

In this section, we motivate the problem of path-sensitive value-flow analysis with an example, demonstrate the design of our approach, and highlight its key merits at the end.

Motivating Example. Figure 1a shows a code snippet with three functions. The program contains two NPD bugs, which are caused by the dereference of the pointers b and d at line 6. For example, the NULL value returned by the function bar and baz can be propagated to the variable b at line 3, which is dereferenced at line 6. Notably, the way of the value propagation can be different under different path conditions. As depicted in Figure 1b,

the value propagation of the actual parameter p diverges in different branches at line 10 inside function bar. In addition, the variable c is also passed to a call instruction at line 5 calling function baz, which cannot finally constitute a NULL pointer dereference, as the parameter c is never assigned with NULL value.

Motivated by the NPD detection, we follow the methodology of value-flow analysis in [11, 53, 55], which first constructs a guarded value-flow graph of the program and then finds bugs via graph traversals. Figure 1b shows the guarded value-flow graph of the program in Figure 1a. Specifically, the framed nodes with NULL indicate the NULL values at different program locations as the sources, while b_6 and d_6 are regarded as a sink. For example, we can find value-flow paths from the source to the sink that may lead to the NPD bug. To address context sensitivity, we need to ensure that the value-flow path is realizable, meaning that the contexts of all call sites, formulated as label strings, are realizable under Dyck-CFL [47]. For example, the value-flow path $(\mathsf{NULL}_2, a_2, a_3, p_8, f_9, f_{10}, b_3, b_6)$ is realizable while another $(\mathsf{NULL}_2, a_2, a_4, q_{12}, h_{13}, h_{15}, d_5, d_6)$ is not[1]. Apart from the realizability, the bug only exists when the disjunction of the conditions of realizable paths from a source to a sink is satisfiable. That is, we must compute the path condition $\Psi(\mathsf{NULL}_2, b_6)$ shown in Figure 1d when considering NULL as the source and b_6 as the sink. If the answer is *sat*, we conclude such a bug exists.

Tabulation-based Approach. Existing efforts often adopt tabulation-based methods to detect such bugs as they have been proven to have cubic-time complexity [47] and can easily adopt parallelism [49, 55]. In particular, the tabulation-based approach analyzes the program by functions in a topological order on the call graph. That is because analyzing one function often requires reachability information when analyzing its callees. Consider the motivating example program in Figure 1a. We cannot discover the fact that the value of the variables a at lines 3 and 4 can reach the value of the variable b at line 4 until we analyze the functions bar and baz. Therefore, the tabulation-based approach often analyzes and summarizes the callee first so the caller can reuse the results.

According to this intuition, many techniques [1, 55] use parallel programming to improve the scalability of tabulation-based analysis. Their key idea is that when analyzing the callee functions without caller-callee relations can be analyzed in parallel. In particular, these methods first analyze functions bar and baz. Using the symbol $\rightsquigarrow$ to denote if two values are reachable, the tabulation-based analysis generates the following summaries:

① $(p_8 \rightsquigarrow f_{10})$: The formal argument of function *bar* can propagate its value to the return value.
② $(\mathsf{NULL}_9 \rightsquigarrow g_{10})$: The NULL value of function *bar* can propagate to the return value.
③ $(q_{12} \rightsquigarrow h_{15})$: The formal argument of function *baz* can propagate its value to the return value.
④ $(\mathsf{NULL}_{13} \rightsquigarrow h_{15})$: The NULL value of function *baz* can propagate to the return value.

Next, it analyzes the function foo by a graph traversal on the guarded value-flow graph. Since there is a NULL pointer assigned to the variable a at line 2, by inlining the above summaries ① and ③, we further identify four potential NPD value-flows and a new summary ⑤:

❶ $\mathsf{NULL}_2, a_3, (p_8 \rightsquigarrow f_{10})_{\text{bar}}, b_3, b_6$
❷ $\mathsf{NULL}_2, a_3, (q_{12} \rightsquigarrow h_{15})_{\text{baz}}, b_3, b_6$
❸ $(\mathsf{NULL}_9 \rightsquigarrow g_{10})_{\text{bar}}, b_3, b_6$
❹ $(\mathsf{NULL}_{13} \rightsquigarrow i_{14})_{\text{baz}}, b_3, b_6$
⑤ $c_1, c_5, (q_{12} \rightsquigarrow h_{15})_{\text{baz}}, d_5, d_6$

With all these paths collected, the analysis will generate the path conditions for realizable paths and use a solver to determine their path feasibility, which finally indicates whether a bug exists. For instance, the path condition of ❶ is $\phi_1 \wedge \phi_2$. Since there is no NULL value passing to c, the redundant summary ⑤ is generated when analyzing the callers of foo.

Notably, the above approaches can first simultaneously analyze the two functions bar and baz in parallel, but after that, they can allocate only one single thread to analyze the function foo. In such functional dependencies, it suffers from the *unbalance* issue, where the concurrent analysis task cannot occupy all CPU cores, preventing

[1]The former forms a label string $(_3)_3$, denoting that the value is passing to a function call to bar at line 3 and returns to the same location, but the latter yields a label string $(_4)_3$ that does not belong to Dyck-CFL.

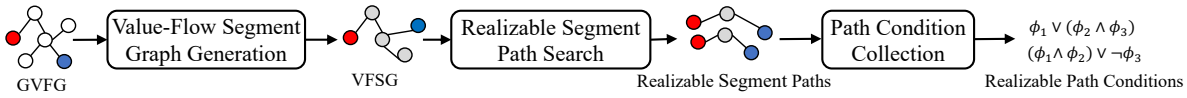


Fig. 2. The overview of Octopus

a high degree of parallelism. Although there are some enhancements [55] that can simultaneously summarize the ①-④ in parallel, its potential speedup is still constrained by the caller-callee relations. In addition, some additional value-flows (e.g., ⑤) are collected during the bottom-up summary inlining, which will never reach from an NPD source, resulting in *summary redundancy*.

Our Approach. To bridge the research gap, we introduce the notion of *realizability decomposition* to parallelize the path-sensitive analysis in three subtasks. Basically, we attempt to decompose the condition collection of realizable paths from the realizability reasoning. More specifically, giving the observation that realizable value-flow paths are concatenated via either formal or actual parameters/return values, we introduce the notion of *value-flow segment* to summarize intraprocedural value-flow paths. With this abstraction, we propose a novel graph representation named *value-flow segment graph*, such that any realizable path on the value-flow segment graph represents several realizable paths in the guarded value-flow graph. Such decomposition also enables load balance in parallelization since both the segment collection and concatenation can be parallelized effectively. Owing to the above benefits, the realizable path conditions can be collected in parallel. Based on the above idea, our parallel algorithm is sketched with three stages, of which the workflow is shown in Figure 2.

- First, we simultaneously traverse each function's guarded value-flow graphs to summarize intraprocedural paths with value-flow segments. For instance, in the function bar in Figure 1a, we obtain five value-flow segments displayed in blue dashed arrows in Figure 1b. By collecting all the value-flow segments and concatenating them using interprocedural edges from the original guarded value-flow graph, we can construct its corresponding value-flow segment graph (Figure 1c).
- Second, we explore the value-flow segment graph in parallel to retrieve all the realizable paths, where each realizable segment path summarizes concrete realizable value-flow paths. For example, we discover the realizable segment paths like ❶$\xrightarrow{(_3}$❹$\xrightarrow{)_3}$❽ in Figure 1c, while discarding segment paths like ❷$\xrightarrow{(_4}$❻$\xrightarrow{)_5}$❾ due to its unrealized strings of edge labels.
- Third, we parallelize the path condition collection from the realizable segment paths and determine their path feasibility of particular source-sink pairs by an SMT solver. For example, the realizable path conditions $(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge (\neg\phi_4 \vee (\phi_3 \wedge \phi_4)))$ for $(\texttt{NULL}_2, b_6)$ comes from the disjunction of path conditions from two realizable segment paths: ❶→❹→❽ and ❷→❻→❽.

Based on the collected realizable path condition, we can further concurrently solve the realizable path conditions and report them as bugs if satisfiable.

Benefits. Due to our technical design, our approach has the following benefits, promoting its practical value in analyzing large-scale programs.

- *Low redundancy:* The abstraction of value-flow segments foresees underlying value-flow paths that can reach sinks. Therefore, the reasoning of realizability and the collection of path conditions are only conducted on segment paths that originate from the sources or reach the sinks.
- *High potential speedup:* Each subprocedure achieves *balanced parallelism* with high potential speedup. The first and the third subtasks can be parallelized at the function level since the computations mainly occur in each function independently. Besides the second task, formulated as the path search problem, can be efficiently solved by adopting an existing parallel graph algorithm with high speedup [29, 44].

$$
\begin{aligned}
\text{Program } P &:= F+ \\
\text{Function } F &:= f(v_1, v_2, \ldots)\{\ S;\ \} \\
\text{Statement } S &:= v_1 = l \mid v_1 = v_2 \mid v_1 = \ v_2 \oplus v_3 \\
&\quad \mid v_1.a = v_2 \mid v_1 = v_2.a \mid v_1 = *\, v_2 \mid \ *v_1 = v_2 \\
&\quad \mid S_1; S_2 \mid \textbf{if}\ (v_1 \odot v_2)\{\ S_1;\ \}\ \textbf{else}\ \{\ S_2;\ \} \\
&\quad \mid v_1 = \ f(v_1, v_2, \ldots) \mid \textbf{return}\ v \\
\text{Arithmetic } \oplus &:= + \mid - \mid \times \mid \div \\
\text{Comparison } \odot &:= \equiv \mid \neq \mid > \mid < \\
\text{Variable } V &:= v_1 \mid v_2 \mid \cdots \\
\text{Field } A &:= a_1 \mid a_2 \mid \cdots \\
\text{Literal } L &:= \text{NULL} \mid l_1 \mid l_2 \mid \cdots
\end{aligned}
$$

Fig. 3. Language syntax

Notably, we do not only partially relax caller-callee dependencies via discovering certain types of independent summary as in previous work [55]. Instead, we relax the dependencies completely via the parallel collection of value-flow segments at the function level. In this manner, superior scalability can be achieved through a high degree of parallelism.

3 PROBLEM FORMULATION

This section presents the basic preliminaries (§ 3.1), formulates the realizable path condition collection problem (§ 3.2), and states the central issue to be solved in this work (§ 3.3).

3.1 Preliminaries

In what follows, we discuss the program syntax and the guarded value-flow graph as the preliminaries.

Syntax. Figure 3 formulates our analysis using a call-by-value language [45]. The semantics of the statements are standard. Following many existing efforts [53, 55], we assume that the program is in SSA form such that every variable is defined only once [16]. We unroll loops a fixed number of times, following the settings of many static analyzers [11, 53] and bounded model checkers [28].

Guarded Value-Flow Graph. For a given program, we say the value of a variable a flows to a variable b if the value of a propagates to b directly (via assignments, such as $b = a$) or indirectly (via indirect dereference, such as $*c = a; d = c; b = *d$ and $c.f = a; d = c; b = d.f$). We adopt existing field-sensitive pointer analyses [32, 53] to resolve pointer relations and covert the indirect value propagation to direct assignments. To depict the value propagation, we construct a guarded value-flow graph, of which an edge abstracts how a value at a program location can be propagated to another under a specific condition. The conditions on edges is generated via existing quasi-path-sensitive analysis [53].

Definition 3.1. (Guarded Value-Flow Graph) The guarded value-flow graph (GVFG) of a program P is a labeled directed acyclic graph $G = (V, E, \mathcal{L}, \Phi)$, where

- $v@s \in V$ denotes the value v defined at the statement s.
- $(v_i@s_i, v_j@s_j) \in E$ indicates that the value v_i is passed to v_j from the statement s_i to s_j.
- $\mathcal{L}$ maps each edge $e = (v_i@s_i, v_j@s_j)$ to a label c, where
  - c is $(_k$ or $)_k$ if $e \in V_{\text{ap}} \times V_{\text{fp}}$ or $e \in V_{\text{fr}} \times V_{\text{ar}}$, respectively. k is a unique id of the call site.

– Otherwise, c is ε, where ε is an empty string.
Here, $V_{\mathsf{fp}}, V_{\mathsf{fr}} \subseteq V$ contain the formal parameters and return values, respectively. $V_{\mathsf{ap}}, V_{\mathsf{ar}} \subseteq V$ contain the actual parameters and return values, respectively. These sets also include the aforementioned auxiliary variables, which model the access and the modification of pointers and fields of structures.

- Φ maps each edge $e = (v_i@s_i, v_j@s_j)$ to a constraint ϕ over V, indicating that the value flow occurs only when ϕ is satisfied.

Intuitively, we can obtain a GVFG G_f for each function f and then construct the GVFG for a whole program P by concatenating all the GVFGs of its functions. Specifically, we can add the edges between the actual/formal parameters/return values with the corresponding labels. As mentioned above, we formulate the problem upon the programs in SSA form. For simplification, we use v_i to indicate a value on GVFG for shorthand and omit the statement defining the value. In the rest of the paper, we demonstrate the examples with the simplified notation.

Example 3.2. Figure 1b shows the GVFG of the program in Figure 1a. Functions bar and baz are invoked by foo. The edges (a_3, p_8), (a_4, p_8), and (a_4, q_{12}) are labeled with $(_3$, $(_4$, and $(_5$, respectively, indicating that the values of actual parameters are passed to the formal parameters at lines 3, 4, and 5, respectively. The edges (f_{10}, b_3), (g_{10}, b_3), (h_{15}, b_3), (i_{14}, b_3), and (h_{15}, d_5) indicate the value flows from formal return values to their receivers. For clarity, we omit the empty label ε for the edges in the GVFG.

3.2 Realizable Path Condition Collection

Given the GVFG of a program and a set of sources and sinks, we can find the paths between the sources and sinks via graph traversal. To filter away spurious paths which cannot occur in the executions, it is also crucial to constrain the value propagation by investigating the conditions of realizable paths. Formally, we formalize the problem of the *realizable path condition collection* as follows.

Definition 3.3. (Realizable Path Condition Collection) Let $G = (V, E, \mathcal{L}, \Phi)$ be a GVFG and $V_{\mathsf{src}}, V_{\mathsf{sink}}$ be the sources and sinks of interest, respectively. A problem instance of realizable path condition collection, denoted as $C = (G, V_{\mathsf{src}}, V_{\mathsf{sink}})$, is to compute a mapping Ψ:

$$\Psi(v_{\mathsf{src}}, v_{\mathsf{sink}}) = \bigvee_{p \in \mathcal{P}(v_{\mathsf{src}}, v_{\mathsf{sink}})} (\Phi(v_1, v_2) \wedge \Phi(v_2, v_3) \wedge \cdots \wedge \Phi(v_{n-1}, v_n)),\ (v_{\mathsf{src}}, v_{\mathsf{sink}}) \in V_{\mathsf{src}} \times V_{\mathsf{sink}}$$

Here, the path set $\mathcal{P}(v_{\mathsf{src}}, v_{\mathsf{sink}}) := \{p : (v_1, \cdots, v_n) \mid (v_i, v_{i+1}) \in E,\ \mathcal{R}(p) \in \mathsf{Dyck\text{-}CFL},\ v_1 = v_{\mathsf{src}},\ v_n = v_{\mathsf{sink}}\}$. $\mathcal{R}(p) := \mathcal{L}(v_1, v_2) \circ \cdots \circ \mathcal{L}(v_{n-1}, v_n)$ is the realized string of the path p. Particularly, $\Psi(v_{\mathsf{src}}, v_{\mathsf{sink}})$ is the realizable path condition for the source-sink pair $(v_{\mathsf{src}}, v_{\mathsf{sink}})$.

For each source-sink pair $(v_{\mathsf{src}}, v_{\mathsf{sink}})$, the collected path condition $\Psi(v_{\mathsf{src}}, v_{\mathsf{sink}})$ poses restrictions on the paths from two aspects. First, the realized string belongs to the Dyck-CFL, which ensures context sensitivity. Second, the realizable path condition is a logical disjunction of the conditions of multiple realizable paths, which start from v_{src} and end with v_{end}, achieving the path sensitivity of the analysis. In practice, realizable path condition collection generally enables a wide range of value-flow analysis clients [19, 24, 36, 38]. For example, detecting value-flow bugs, such as NULL pointer dereference [52, 68], memory leaks [11, 65], taint vulnerabilities [3, 4, 55], require specific instantiations of V_{src} and V_{sink}. Without loss of generality, we adopt the realizable path condition collection of the NPD detection as an example in the rest of the paper.

Example 3.4. Given the GVFG $G = (V, E, \mathcal{L}, \Phi)$, the NPD detection can be reduced to collecting the conditions of realizable paths, for the instance $C_{\mathsf{NPD}} = (G, V_{\mathsf{src}}, V_{\mathsf{sink}})$, where

$$V_{\mathsf{src}} = \{v \in V \mid \mathsf{mayNull}(v)\},\quad V_{\mathsf{sink}} = \{v \in V \mid \mathsf{deref}(v)\}$$

Algorithm 1: Collecting realizable path conditions

```
1  Procedure collectRealizablePathCond(G, V_src, V_sink):
2      Ψ ← [(v_src, v_sink) ↦ false | (v_src, v_sink) ∈ V_src × V_sink];
3      forall v_1 ∈ V_src do
4          WL ← {(v_1)};
5          while WL ≠ ∅ do
6              (v_1, ..., v_n) ← poll (WL);
7              if v_n ∈ V_sink and R(v_1, ..., v_n) ∈ Dyck-CFL :
8                  Ψ(v_1, v_n) ← Ψ(v_1, v_n) ∨ (Φ(v_1, v_2) ∧ ··· ∧ Φ(v_{n−1}, v_n));
9              forall v_{n+1} ∈ next(v_n) do
10                 WL ← WL ∪ {(v_1, ..., v_n, v_{n+1})};
11     return Ψ
```

Here, mayNull(v) depicts whether v is equal to NULL, and deref(v) depicts whether v is dereferenced. By solving the collected condition, we can determine whether there exists a feasible path from a source to a sink, which indicates the presence of an NPD.

3.3 Problem Statement

To solve the realizable path condition collection problem, a naive approach is to instantiate a worklist algorithm [25], which is shown as Algorithm 1. For each source $v_1 \in V_{\text{src}}$, we can traverse the GVFG to collect all the realizable value-flow paths that start with v_1 and ends with a sink $v_n \in V_{\text{sink}}$. Furthermore, we can compute the path condition for each path and create the disjunction with all the conditions of realizable paths for each source-sink pair. Finally, the value of the mapping Ψ reveals whether or not a source can reach a sink in any concrete execution of the program, which can guide various clients, such as value-flow bug detection.

Unfortunately, the worklist algorithm suffers exponentially huge overhead when analyzing large-scale programs. First, when we compute the path conditions for intraprocedural paths, the exponential number of the paths can result in high complexity barrier. Second, the problem becomes more difficult when considering the interprocedural path search along with the realizability reasoning. To improve efficiency, one typical way is to design a parallel version of the worklist algorithm. Existing studies attempt to parallelize tabulation-based program analysis techniques [2, 55] for performance improvement. However, the potential speedup is strictly restricted by the caller-callee relations in the program, making the techniques suffer the load imbalance in the parallelization. To fill the research gap, we aim to tackle the following problem in this work.

> Given a GVFG G, V_{src}, and V_{sink}, compute the realizable path condition $\Psi(v_{\text{src}}, v_{\text{sink}})$ for each source-sink pair $(v_{\text{src}}, v_{\text{sink}}) \in V_{\text{src}} \times V_{\text{sink}}$ in a parallel manner, which achieves balanced parallelism for high speedup.

Roadmap. In what follows, we present a parallel algorithm for collecting realizable path conditions in the value flow analysis. Specifically, we introduce a novel abstraction of the realizable paths (§ 4), enabling the decoupling of the intraprocedural path exploration from the realizability reasoning. To resolve the imbalance issue, we generate a value-flow segment graph (§ 5.1), where each value-flow segment summarizes the intraprocedural value-flow paths. Furthermore, we parallelize the realizable segment path exploration (§ 5.2) and finally collect realizable path conditions on demand (§ 5.3). Each stage features a high potential speedup, which ensures the high efficiency of the overall parallel algorithm.

4 DECOMPOSING REALIZABLE PATH CONDITIONS

In this section, we establish the fundamental decomposition of our realizable path conditions to convenience the parallelization of the condition collection. We first present the realizability decomposition by introducing the notion of the value-flow segment (§ 4.1), and then demonstrate the basic design of collecting realizable path conditions with the guidance of value-flow segments (§ 4.2), which resolves the imbalance issue caused by caller-callee relations. We also formulate the soundness and completeness of the realizability decomposition, providing the theoretical foundation of our parallel algorithm.

4.1 Realizability Decomposition

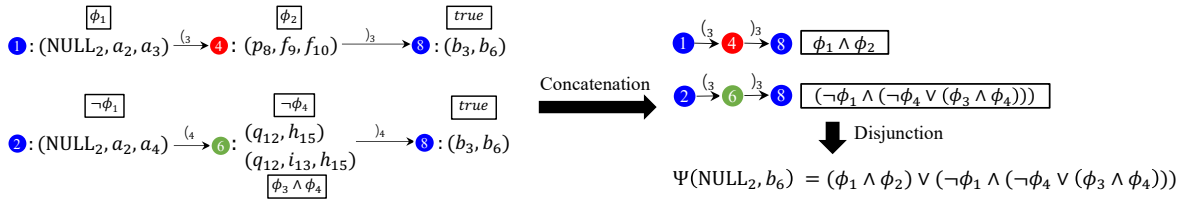
As shown by Definition 3.3, the realizable path condition is induced by the set of realizable paths starting from v_{src} and ending with v_{sink}. This implies that we have to ensure the realizability of each reachable path from v_{src} and v_{sink} in the path condition collection. However, it is non-trivial to guarantee the realizability of a source-sink path in a parallel value-flow analysis. Typically, a tabulation-based analysis has to explore the paths of a callee before analyzing its caller, which is referred to as *the limits of function boundaries* in [55]. Obviously, the limits of function boundaries significantly restrict the potential speedup of parallelization. Even if we have a sufficiently large number of threads, the efficiency of the parallel algorithm can not be further improved, which bears a low CPU utilization.

To break the limits of function boundaries, we propose the concept of realizability decomposition, which decouples the realizability reasoning from the intraprocedural path exploration. Specifically, we observe that the intraprocedural value-flow paths form the realizable paths across the functions via the concatenation. This suggests that there are actually three counterparts in realizable path condition collection: (1) Explore the intraprocedural paths of each function; (2) Enumerate all the possible ways of concatenating the intraprocedural value-flow paths to enforce the realizability. (3) Construct the path conditions by utilizing the relevant intraprocedural value-flow paths and the ways of concatenation. Notably, searching intraprocedural paths has already born the exponential time complexity, while determining reachability is much cheaper. Therefore, we can first use the reachability of specific nodes to summarize the intraprocedural paths in the GVFG and postpone the exploration of the relevant value-flow paths and the collection of path conditions. Formally, we propose a critical notion of the *value-flow segment* to summarize intraprocedural paths and depict the reachability relation.

Definition 4.1. (Value-Flow Segment) Given a function f and its GVFG G_f, a value-flow segment $\ell : v_1 \rightsquigarrow v_2$ is the shortcut summarizing the value-flow paths from v_1 to v_2, where $v_1 \in (V^f_{\text{src}} \cup V^f_{\text{fp}} \cup V^f_{\text{ar}})$ and $v_2 \in (V^f_{\text{sink}} \cup V^f_{\text{fr}} \cup V^f_{\text{ap}})$. Here V^f_{src} is the set of sources specified in G_f, and other notations follow similar meanings as in Definition 3.1. We denote the set of the value-flow segments in f by S_f.

Intuitively, a value-flow segment depicts the reachability between v_1 and v_2 without enumerating the paths. The set S_f actually abstracts the semantics of a single function, showing how the values of interest are generated and propagated. If a value-flow segment starts from a source $v_1 \in V^f_{\text{src}}$ or ends with a return value $v_2 \in V^f_{\text{fr}}$, it can be used to form the value-flow paths across different functions. Also, it is worth noting that the value-flow segments in different functions are independent since they are essentially abstractions of the GVFGs of single functions. This property permits us to parallelize the value-flow segment generation at the function level, which ensures a high potential speedup of summarizing intraprocedural paths.

Example 4.2. Consider Figure 1a and we aim to detect NPD bugs. There are nine value-flow segments in the three functions. Let us denote the value-flow segments labeled with ❶, ❷, ❹, ❻, and ❽ by ℓ_1, ℓ_2, ℓ_4, ℓ_6, and ℓ_8, respectively. The rest of the paper follows similar notations. Figure 4a shows the intraprocedural value-flow paths summarized by the four value-flow segments. Specifically, ℓ_1 summarizes the path $(\texttt{NULL}_2, a_2, a_3)$, showing



(a) Intraprocedural paths summarized by value-flow segments

(b) Realizable path conditions for each segment path and a source-sink pair

Fig. 4. Examples of value-flow segments and realizable paths derived from segment concatenation.

that the NULL value at line 2 is used as the actual parameter of bar at line 3 and is similar for ℓ_2. The value-flow segments ℓ_4 and ℓ_5 summarize the value-flow paths from the parameter to the two return values of bar and baz, respectively. ℓ_8 summarizes the path (b_3, b_6), and b_6 is further dereferenced at line 6.

With the value-flow segments, we can determine how any realizable path connecting a source to a sink forms in the GVFG. For each function, the values of interest can only be propagated to other functions via function calls and returns. When an edge connects the function g's formal parameter v and the function f's actual parameter u, the value-flow segments in the two functions can form a longer value-flow path. Similar propagation also occurs upon formal and actual return values. Based on this intuition, we formalize the segment concatenation as follows.

Definition 4.3. (Segment Concatenation) Given the GVFG $G = (V, E, \mathcal{L}, \Phi)$ and a string s, the value-flow segments $\ell_f := v_1^f \rightsquigarrow v_2^f \in S_f$ and $\ell_g := v_1^g \rightsquigarrow v_2^g \in S_g$ are *concatenable* under s if and only if $e = (v_2^f, v_1^g) \in E$ and $s \circ \mathcal{L}(e)$ belongs to the extended Dyck-CFL [51]. In other sections of the paper, we refer to Dyck-CFL as the extended Dyck-CFL.

Example 4.4. Consider the function bar invoked by the function foo in Figure 1a. ℓ_1 and ℓ_4 are concatenable under $(_3$, and ℓ_4 and ℓ_8 are concatenable under $)_3$. Therefore, they form a realizable path from NULL_2 to b_6, which reveals a possible NPD bug at line 6. For other value-flow segments, we can also conduct the segment concatenation and obtain the other realizable paths shown in Figure 4.

The segment concatenation enables a compositional approach to identify realizable paths. Using the value-flow segments of different functions, we can always recover all realizable paths connecting sources and sinks. To solidify the theoretical foundation, we formulate the following two theorems, stating the soundness and completeness of the realizability decomposition, respectively.

Theorem 1. (Soundness) Let $G = (V, E, \mathcal{L}, \Phi)$ be the GVFG of a given program P. If we select k value-flow segments $\ell_i : u_i \rightsquigarrow w_i$ $(1 \le i \le k)$, which satisfy the following conditions:

- $u_1 = v_{\text{src}}$ and $w_k = v_{\text{sink}}$;
- ℓ_i and ℓ_{i+1} are concatenable under $\mathcal{L}(w_1, u_2) \circ \cdots \circ \mathcal{L}(w_{i-1}, u_i)$.

then $p_1 \uplus p_2 \uplus \cdots \uplus p_k$ is a realizable path connecting v_{src} and v_{sink}, where p_j is an intra-procedural value-flow path from u_j to w_j. The operator $\uplus$ is the concatenation operator of two value-flow paths $p : (v_1, v_2, \cdots, v_n)$ and $p' : (v_1', v_2', \cdots, v_{n'}')$, where $v_n = v_1'$.

Proof. For each $\ell_i : u_i \rightsquigarrow w_i$, we have the intra-procedural value-flow path p_i connecting u_i and w_i, which is summarized by ℓ_i. Based on Definition 4.3, w_i can reach u_{i+1} because ℓ_i and ℓ_{i+1} are concatenable. Based on the transitivity of reachability, v_{src} and v_{sink} are reachable. Also, ℓ_{k-1} and ℓ_k are realizable under $\mathcal{L}(w_1, u_2) \circ \cdots \circ$

$\mathcal{L}(w_{k-2}, u_{k-1})$. Based on Definition 4.3, the realized string $\mathcal{R}(p_1 \uplus p_2 \uplus \cdots \uplus p_k) = \mathcal{L}(w_1, u_2) \circ \cdots \circ \mathcal{L}(w_{k-2}, u_{k-1}) \circ \mathcal{L}(w_{k-1}, u_k)$ is in the extended Dyck-CFL. Therefore, $p_1 \uplus p_2 \uplus \cdots \uplus p_k$ is a realizable path. □

Theorem 2. (Completeness) Consider any realizable path $p := p_1 \uplus p_2 \uplus \ldots \uplus p_k$ connecting v_{src} and v_{sink}, where p_i is an intraprocedural value-flow path for each $1 \leq i \leq k$. Then we can always find the value-flow segments $\ell_i : u_i \rightsquigarrow w_i$ $(1 \leq i \leq k)$, such that

- p_i starts with u_i and ends with w_i;
- $\ell_1 = v_{\text{src}} \rightsquigarrow w_1$ and $\ell_k = u_k \rightsquigarrow v_{\text{sink}}$, where $w_1, u_k \in V$;
- ℓ_i and ℓ_{i+1} are concatenable under $\mathcal{L}(w_1, u_2) \circ \cdots \circ \mathcal{L}(w_{i-1}, u_i)$. (revised: "$\mathcal{L}(w_0, u_1) = \varepsilon$" removed)

Proof. If the theorem does not hold, we can find a value-flow path $p^* : v^*_{\text{src}} \rightsquigarrow v^*_{\text{sink}}$ which is not a concatenation of a finite number of value-flow segments. Consider that v^*_{src} lies in the GVFG of the function f_1. We remove the successive nodes in the prefix of p^* which are in the GVFG nodes of f_1, and obtain an empty value-flow path or another value-flow path $p^* : v^*_2 \rightsquigarrow v^*_{\text{sink}}$, where $v^*_2 \in V_{\text{fp}} \cup V_{\text{ar}}$. In the first case, p^* must be summarized by a value-flow segment of f_1, which contracts with our assumption. In the second case, we obtain a shorter value-flow path p^* which is not a concatenation of the paths summarized by value-flow segments. We repeat the removal for the second case, while it cannot be repeated infinite times due to the finite length of p^*. The assumption does not hold, and thus, the theorem is proved. □

Theorems 1 and 2 establish the theoretical foundation of our realizability decomposition. No matter how value-flow paths concatenate and in which order value-flow segments are generated, we can always examine the same set of realizable paths $\mathcal{P}(v_{\text{src}}, v_{\text{sink}})$ in the collection of path conditions. Hence, the correctness of the parallelization can be theoretically guaranteed.

4.2 From Value-Flow Segments to Realizable Path Conditions

As demonstrated in § 4.1, value-flow segments summarize value-flow paths in single functions, which enables us to determine how realizable paths are formed across multiple functions via concatenation. Once we obtain the concatenation of the value-flow segments, we can collect the path condition of the intraprocedural paths summarized by value-flow segments and create their logical disjunctions in a parallel manner. Hence, we only need to focus on the parallelization of identifying how to concatenate the value-flow segments. Based on intuition, we propose a novel abstraction named the *value-flow segment graph* as follows.

Definition 4.5. (Value-Flow Segment Graph) Given a program P and its GVFG $G = (V, E, \mathcal{L}, \Phi)$, the value-flow segment graph (VFSG) of P is $\overline{G} = (\overline{V},\ \overline{E},\ \overline{\mathcal{L}})$, where

- The segment set $\overline{V} = \bigcup_{f \in P} S_f$ contains all the value-flow segments in G.
- The set $\overline{E}$ contains the edges of the VFSG, and $\overline{L}$ maintains the label of each edge. For each $\ell_i = v_i \rightsquigarrow u_i$ and $\ell_j = v_j \rightsquigarrow u_j$, we have $(\ell_i, \ell_j) \in \overline{E}$ and $\overline{\mathcal{L}}(\ell_i, \ell_j) = \mathcal{L}(u_i, v_j)$ if and only if $(u_i, v_j) \in E \cap ((V_{\text{fp}} \times V_{\text{ap}}) \cup (V_{\text{fr}} \times V_{\text{ar}}))$.

Particularly, a value-flow segment $\ell_1 : v_1 \rightsquigarrow v_2 \in \overline{V}$ is a source in the VFSG if and only if $v_1 \in V_{\text{src}} \subseteq V$, while a value-flow segment $\ell_2 : u_1 \rightsquigarrow u_2 \in \overline{V}$ is a sink in the VFSG if and only if $u_2 \in V_{\text{sink}} \subseteq V$. The sets of sources and sinks in the VFSG are denoted as $\overline{V_{\text{src}}}$ and $\overline{V_{\text{sink}}}$, respectively.

Intuitively, the VFSG abstracts away how the values are propagated within single functions and only exposes the value propagation across the functions. More concretely, it leverages a value-flow segment to summarize the intraprocedural value-flow paths with the same start and end points. Based on the VFSG, we can establish the connection between realizable paths in the GVFG with a specific segment path in the VFSG, of which the realized string is in the extended Dyck-CFL. According to this intuition, we define the notion of the *realizable segment path*.

Definition 4.6 (Realizable Segment Path)**.** A segment path $\overline{p} = (\ell_1, \ell_2, \ldots, \ell_n)$ in the VFSG $\overline{G}$ is a realizable segment path if and only if $\overline{\mathcal{L}}(\ell_1, \ell_2) \circ \cdots \circ \overline{\mathcal{L}}(\ell_{n-1}, \ell_n)$ belongs to the extended Dyck-CFL.

A realizable segment path is a more compact representation of the realizable paths in the GVFG. According to Theorem 2 and Definition 4.5, each realizable path can be projected to the realizable segment path in the VFSG. Meanwhile, Theorem 1 indicates that by replacing each value-flow segment with an intraprocedural value-flow path, any realizable segment path can induce a set of realizable paths. Therefore, the VFSG enables us to resolve the realizability from a high-level perspective, blurring the details of single functions and focusing solely on possible realized strings of realizable segment paths. In this way, we can determine the set of realizable paths in Definition 3.3 for the condition collection.

Example 4.7. Figure 1c shows the VFSG of the program in Figure 1a. The segments ℓ_1, ℓ_2 are sources on VFSG, while the segment ℓ_8 is a sink. Each segment summarizes one or more intraprocedural value-flow paths with corresponding path conditions. We notice that there exist two realizable segment paths starting from ℓ_1 and from ℓ_2 and end with ℓ_8 in the VFSG. According to the concatenation process shown in Figure 4, we can recover two realizable segment paths (ℓ_1, ℓ_4, ℓ_8) and (ℓ_2, ℓ_6, ℓ_8) as well as their corresponding path conditions $\phi_1 \wedge \phi_2$ and $\neg\phi_1 \wedge (\neg\phi_4 \vee (\phi_3 \wedge \phi_4))$, respectively. Since both ℓ_1 and ℓ_2 originates from the same source value $\texttt{NULL}_2$, we can obtain the realizable path condition for the source-sink pair $(\texttt{NULL}_2, b_6)$ by the disjunction the above conditions: $(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge (\neg\phi_4 \vee (\phi_3 \wedge \phi_4)))$.

Owning to our realizability decomposition, we decouple the path condition collection from the realizability reasoning. Such a separation is beneficial for the following three reasons:

- First, the decomposition enables us to foresee the possible ways of concatenating intraprocedural paths to a realizable counterpart without enumerating them, thus, avoiding examining redundant intraprocedural value-flow paths. In Figure 1, for instance, we do not explore intraprocedural paths summarized by ℓ_6, as it can not form any realizable segment path.
- Second, the realizability reasoning only happens once when multiple value-flow paths share the same realized strings, which can effectively avoid redundancy. For example, once we obtain a segment path (ℓ_1, ℓ_6, ℓ_8), we examine the realized string "$(_4)_4$" and find that it belongs to the extended Dyck-CFL. We do not need to check the string again for different value-flow paths summarized in a segment path, which can further improve efficiency.
- Third, the decoupled computation can be efficiently parallelized in a balanced manner, achieving a high speedup. In particular, computing the value-flow segments and collecting path conditions with the guidance of realizable segment paths can be easily parallelized in a balanced manner because each function and segment can be processed independently.

In what follows, we provide the technical details of our parallel algorithm, including the value-flow segment graph generation (§ 5.1), the realizable segment path search (§ 5.2), and the path condition collection (§ 5.3), all of which can be parallelized with the high potential speedup.

5 PARALLEL ALGORITHM

This section demonstrates the parallel algorithm Octopus to collect realizable path condition (§ 5.1–§ 5.3), and summarizes the benefits as discussions (§ 5.4).

5.1 Value-Flow Segment Graph Generation

According to the realizability decomposition proposed in §4, we know that any realizable path in the GVFG can be projected to a realizable segment path in the VFSG. This implies that the realizable segment paths in the VFSG express all the way how intraprocedural paths form the realizable paths in the GVFG. Meanwhile, the realizable

Algorithm 2: Generating value-flow segment graph

1 **Procedure** genVFSegmentGraph($P, G = (V, E, \mathcal{L}, \Phi)$):
2 $\quad \overline{V} \leftarrow \emptyset$; $\overline{E} \leftarrow \emptyset$; $\overline{\mathcal{L}} \leftarrow \bot$;
3 $\quad$ **foreach** $f \in P$ **do in parallel**
4 $\quad\quad (V_f, E_f) \leftarrow$ getGVFGForFunction(G, f);
5 $\quad\quad$ **foreach** $v_i \in V^f_{\text{src}} \cup V^f_{\text{fp}} \cup V^f_{\text{ar}}$ **do in parallel**
6 $\quad\quad\quad$ **foreach** $v_j \in$ visitedInBFS(G_f, v_i) :
7 $\quad\quad\quad\quad$ **if** $v_j \in V^f_{\text{sink}} \cup V^f_{\text{ap}} \cup V^f_{\text{fr}}$:
8 $\quad\quad\quad\quad\quad \overline{V} \leftarrow \overline{V} \cup \{v_i \rightsquigarrow v_j\}$;
9 $\quad$ **foreach** $(v_i, v_j) \in E$ **and** $\mathcal{L}(v_i, v_j) \neq \varepsilon$ **do in parallel**
10 $\quad\quad$ **if** $\ell_1, \ell_2 \in \overline{V}$ **and** last$(\ell_1) = v_i$ **and** first$(\ell_2) = v_j$:
11 $\quad\quad\quad \overline{E} \leftarrow \overline{E} \cup \{\ell_1 \rightsquigarrow \ell_2\}$;
12 $\quad\quad\quad \overline{\mathcal{L}}(\ell_1, \ell_2) \leftarrow \mathcal{L}(v_i, v_j)$;
13 $\quad$ **return** $\overline{G} \leftarrow (\overline{V}, \overline{E}, \overline{\mathcal{L}})$;

segment paths summarize context-sensitive analysis results and pose the potential of path condition recovery. Based on this intuition, we concentrate on efficiently generating the VFSG first, which serves as the essential ingredient for further path exploration.

Notably, we realize that value-flow segments of different functions do not depend on each other, making function-level parallelism easy to achieve. Leveraging the independence of the value-flow segments in different functions, we propose the parallel algorithm to construct the VFSG, which is formulated as Algorithm 2. Specifically, it takes as input a program and its GVFG and returns the VFSG of the program. It generates value-flow segments for each function simultaneously with different workers. For each function f, a worker traverses the GVFG G_f from each value v_i in $V^f_{\text{src}} \cup V^f_{\text{fp}} \cup V^f_{\text{ar}}$, and determines whether any value v_j in $V^f_{\text{sink}} \cup V^f_{\text{ap}} \cup V^f_{\text{fr}}$ can be reached. This can be achieved by standard breath-first search (BFS), initialized as `visitedInBFS` function, which returns the values in a BFS order. Finally, it collects all the value-flow segments in any single function (lines 3–8). After obtaining the value-flow segments, it connects any pair of segments ℓ_1 and ℓ_2 with an edge labeled with $\mathcal{L}(v_i, v_j)$, where v_i and v_j are the end point and start point of the paths summarized by ℓ_1 and ℓ_2, respectively (lines 9–12). In this way, the constructed VFSG preserves the calling context of the program P, enabling us to achieve context sensitivity in the subsequent stages.

Example 5.1. When exploring segments of the function bar in Figure 1a for NPD detection, Algorithm 2 starts from the formal parameter p_8 and finally reaches f_{10}, yeilding a value-flow segment $\ell_4 : p_8 \rightsquigarrow c_{10}$. Similarly, it collects the other nine value-flow segments by processing the three functions in a parallel manner. By taking the interprocedural edges and their edge labels from the original GVFG, we can construct the VFSG shown in Figure 1c.

As shown in Algorithm 2, each function can be processed independently during the value-flow segment generation. For a specific function f, it also processes each pair $v_1 \in (V^f_{\text{src}} \cup V^f_{\text{fp}} \cup V^f_{\text{ar}})$ simultaneously. Besides, it adds the edges in the VFSG by examining the edges in the GVFG parallelly. Benefiting from our fine-grained task decomposition, we can achieve a high degree of parallelism in the VFSG generation. Formally, we provide a theorem to show the potential speedup of Algorithm 2.

Algorithm 3: Searching realizable segment paths

Procedure exploreRSPaths($\overline{G} := (\overline{V}, \overline{E}, \overline{\mathcal{L}})$):
  $\overline{\mathcal{A}} \leftarrow \emptyset$; $n \leftarrow 0$;
  $F_0 \leftarrow \{(\ell) \mid \ell \in \overline{V_{\text{src}}}\}$;
  $\mathcal{R} \leftarrow [\bar{p} \mapsto \varepsilon \mid \bar{p} \in F_0]$;
  while $F_n \neq \emptyset$ **do**
    $F_{n+1} \leftarrow \emptyset$;
    foreach $\bar{p} := (\ell_i, \cdots, \ell_j) \in F_n$ **do in parallel:**
      if $\ell_j \in \overline{V_{\text{sink}}}$ **then**
        $\overline{\mathcal{A}} \leftarrow \overline{\mathcal{A}} \cup \{(\ell_i, \cdots, \ell_j)\}$
      foreach $(\ell_j, \ell_k) \in \overline{E}$ **do in parallel:**
        $s \leftarrow \mathcal{R}(\ell_i, \cdots, \ell_j)\overline{\mathcal{L}}(\ell_j, \ell_k)$;
        if s *is realizable* **then**
          $F_{n+1} \leftarrow F_{n+1} \cup \{(\ell_i, \cdots, \ell_j, \ell_k)\}$;
          $\mathcal{R} \leftarrow \mathcal{R}[(\ell_i, \cdots, \ell_j, \ell_k) \mapsto s]$;
    $n \leftarrow n + 1$;
  return $\overline{\mathcal{A}}$;

Theorem 3. Given a program P and its GVFG $G = (V, E, \mathcal{L}, \Phi)$, the potential speedup of Algorithm 2 is $\Omega(\frac{|V_{\text{src}}| \cdot (|V|+|E|)}{N_v+N_e})$. Here, N_v and N_e are the maximal numbers of vertices and edges in the GVFGs of single functions, and $|V_{\text{src}}|$ is the number of the sources in the GVFG G.

Proof. First, the total work T_1 includes at least two parts: (1) the traversal upon the GVFG for each source for the value-flow segment generation; and (2) one round scan of the edge set E and the mapping $\mathcal{L}$ to instantiate the map $\overline{\mathcal{L}}$. Thus, we have $T_1 = \Omega(|V_{\text{src}}| \cdot (|V| + |E|) + |E|)$. Second, the span of the work T_∞ is exactly the maximal cost of one round BFS upon the GVFG of a single function for segment generation. The span of the one round scan of E and $\mathcal{L}$ is $O(1)$ as it only performs a one-time query to $\mathcal{L}$. Thus, we have $T_\infty = O(N_v + N_e)$. Finally, the potential speedup is $\frac{T_1}{T_\infty} = \Omega(\frac{|V_{\text{src}}| \cdot (|V|+|E|)+|E|}{N_v+N_e}) = \Omega(\frac{|V_{\text{src}}| \cdot (|V|+|E|)}{N_v+N_e})$.

□

It is worth mentioning that Theorem 3 does not provide a tight estimation theoretically. In extreme cases where $N_v = |V|, N_e = |E|$, the potential speedup is $\Omega(|V_{\text{src}}|)$. However, N_v and N_e are considered much smaller than $|V|$ and $|E|$ in real-world cases, as a program is unlikely to have a function of which the size is close to the total size of the program. Therefore, Algorithm 2 can achieve high speedup in practice. In addition, even if such an extreme case exists, $|V_{\text{src}}|$ is generally larger than the number of available threads of a workstation. Thus, we can achieve maximum speedup when analyzing real-world programs.

Besides, this stage can involve redundancy since all types of value-flow segments (Definition 4.1) are collected no matter whether they contribute to a feasible value-flow path. However, It completely relaxes the caller-callee dependencies and brings almost maximum available parallelism compared to previous work only doing partial relaxation [55]. In addition, such redundancy is subtle to the overall computation effort since the segment collection is a pure reachability graph search, which is considered very efficient.

5.2 Realizable Segment Path Search

The realizable segment paths in the VFSG depict all the possible concatenations of intraprocedural paths on the GVFG. In what follows, we demonstrate how to parallelize the realizable segment path search in the VFSG with a high potential speedup.

Our key idea to achieve parallelism is to blend the realizability reasoning with the conventional parallel BFS-based algorithms [6, 31, 73]. Algorithm 3 demonstrates the technical details. Initially, we collect all the sources on the VFSG to form the 0-length paths (line 3). Specifically, we use F_n stores all the intermediate paths with a length of n, of which the realized strings belong to the Dyck-CFL. We also introduce a mapping $\mathcal{R}$ to store realized strings of the segment paths (line 4). When F_n is not empty, Algorithm 3 processes each segment path $\bar{p} \in F_n$ in a parallel manner, which consists of two phases:

- It examines whether the last segment ℓ_j in the path $\bar{p}$ is a sink in the VFSG. If yes, it adds $\bar{p}$ to the set of realizable segment paths $\overline{\mathcal{A}}$ (line 8).
- It extends the segment path $\bar{p}$ by concatenating it with ℓ_k if and only if $\mathcal{R}(\bar{p})\overline{\mathcal{L}}(\ell_j, \ell_k)$ is realizable, where ℓ_k is the successor of the last segment ℓ_j of the segment path $\bar{p}$ (lines 10–14).

By parallelizing the two phases, Algorithm 3 searches the segment paths by iterations until no new intermediate path exists. Finally, $\overline{\mathcal{A}}$ collects all realizable segment paths from $\overline{V_{\text{src}}}$ to $\overline{V_{\text{sink}}}$ in the VFSG.

Example 5.2. Consider the VFSG shown in Figure 1c. Initially, we collect (ℓ_1), (ℓ_2), (ℓ_5), and (ℓ_7) to form F_0. After the first round of the iteration, we have $F_1 = \{(\ell_1, \ell_4), (\ell_2, \ell_6), (\ell_5, \ell_8), (\ell_3, \ell_6), (\ell_7, \ell_8), (\ell_7, \ell_9)\}$. There are already three paths ending with the sink in F_1, namely (ℓ_2, ℓ_6), (ℓ_7, ℓ_8), and (ℓ_7, ℓ_9). In the second round, we find that "$(_4)_4$" is realizable while "$(_4)_5$" is not realizable, so we have $F_2 = \{(\ell_1, \ell_4, \ell_8), (\ell_2, \ell_6, \ell_8)\}$. Due to $\ell_8 \in \overline{V_{\text{sink}}}$, we add the two paths in F_2 to $\overline{\mathcal{A}}$. Obviously, F_3 is empty, as ℓ_8 does not have a successor. Finally, we collect five realizable segment paths in the VFSG and store them in $\overline{\mathcal{A}}$ in parallel.

Here, we provide the theorem on the potential speedup of the parallel Algorithm 3.

Theorem 4. Consider the VFSG $\overline{G} = (\overline{V}, \overline{E}, \overline{\mathcal{L}})$ for a given program. The potential speedup of Algorithm 3 is $\Omega(\frac{|\overline{V_{\text{src}}}| \cdot (|\overline{V}| + |\overline{E}|)}{\overline{D}})$, where $|\overline{V_{\text{src}}}|$ is the number of the value-flow segments starting from sources, and $\overline{D}$ is the largest length of the segment paths in the VFSG.

Proof. First, the total work is at least $T_1 = \Omega(|\overline{V_{\text{src}}}| \cdot (|\overline{V}| + |\overline{E}|))$. This is because, we need to perform at least one graph traversal for each source value in V_{src} to determine whether there exists at least one segment path from V_{src}, while the total work can be higher there exist multiple paths reachable from a source value. Similar to the proof of Theorem 3, we have $T_\infty = O(\overline{D})$ with a sufficient number of threads. Thus, the potential speedup of Algorithm 3 is $\frac{T_1}{T_\infty} = \Omega(\frac{|\overline{V_{\text{src}}}| \cdot (|\overline{V}| + |\overline{E}|)}{\overline{D}})$. □

Note that, in extreme cases where $|\overline{D}| = O(|\overline{E}|)$, we still have a potential speedup $\Omega(\frac{|\overline{V_{\text{src}}}| \cdot (|\overline{V}| + |\overline{E}|)}{\overline{D}}) > \Omega(|\overline{V_{\text{src}}}|)$, which is also unlikely to be smaller than the number of available physical threads of a server. Thus, we also can achieve maximum speedup in practice.

Essentially, Algorithm 3 fuses parallel BFS algorithms to the VFSG with realizablity reasoning, enforcing the realized strings of the segment paths which belong to the Dyck-CFL. The realizablity reasoning does not bring extra overhead, since we can utilize a stack to store each realized string and match the parentheses directly, i.e., it can be done in $O(1)$ time for each path. Also, it is worthy noting that we only search from segments starting from sources, and thus segments like ℓ_3 will not be counted. This also removes redundancies brought by tabulation-based methods which cannot foretell that ℓ_2 will never form in a source-sink path.

Algorithm 4: Collecting realizable path conditions.

```
1  Procedure collectPathCond(𝒜̄, G):
2      Ψ ← [(v_src, v_sink) ↦ false | (v_src, v_sink) ∈ V_src × V_sink];
3      foreach p̄ ∈ 𝒜̄ do in parallel
4          v_src ← first(p̄);
5          v_sink ← last(p̄);
6          φ_p̄ ← true;
7          if (v_src, v_sink) ∉ dom(Ψ) then
8              Ψ(v_src, v_sink) ← false;
9          foreach ℓ_i ∈ VFSegments(p̄) do
10             φ_ℓi ← getPathCond(ℓ_i, G);
11             φ_p̄ ← φ_p̄ ∧ φ_ℓi;
12         Ψ(v_src, v_sink) ← Ψ(v_src, v_sink) ∨ φ_p̄;
13     return Ψ;
```

5.3 Path Condition Collection

Recall that a realizable segment path only abstracts how a source v_{src} and a sink v_{sink} are realized interprocedurally, leaving its path feasibility unknown. To determine whether a pair of the source and the sink indicates the existence of a bug, we have to efficiently collect the conditions of the paths summarized by each value-flow segment and then concatenate them according to realizable segment paths in the VFSG.

Algorithm 4 shows the parallel algorithm of collecting path conditions for bug detection. For each unique pair of $(v_{\text{src}}, v_{\text{sink}})$ in the segment paths, it attempts to generate the corresponding path condition $\Psi(v_{\text{src}}, v_{\text{sink}})$. It takes as input the set of the realizable segment paths $\overline{\mathcal{A}}$, which are returned by Algorithm 3. Here, `VFSegments` returns a sequence of value-flow segments in the realizable segment path $\bar{p}$. For each value-flow segment $\ell_i : u \rightsquigarrow w$ occurring in each realizable segment path $\bar{p} \in \overline{\mathcal{A}}$, `getPathCond` collects the path conditions of u reaching w (line 10) on the GVFG in a disjunctive form. Eventually, it disjuncts all path conditions with the same sources and sinks (line 12). Once the path condition is *sat*, we conclude that there is a feasible path from the source to the sink, indicating the presence of a value-flow bug.

Example 5.3. As shown in Example 5.2, we have $\overline{\mathcal{A}} = \{(\ell_1, \ell_4, \ell_8), (\ell_2, \ell_6, \ell_8), (\ell_5, \ell_8), (\ell_7, \ell_8), (\ell_7, \ell_9)\}$ for the motivating example in Figure 1. Suppose we collect path conditions for $(\texttt{NULL}_2, b_6)$ as examples, Algorithm 4 processes two paths simultaneously, yielding two path conditions: $\phi_1 \wedge \phi_2$ and $\neg\phi_4 \vee (\phi_3 \wedge \neg\phi_4)$. Taking the disjunction of them, we have $\Psi(\texttt{NULL}_2, b_6) = (\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge (\neg\phi_4 \vee (\phi_3 \wedge \neg\phi_4)))$. In this way, we also process path conditions for other sources and sinks in parallel and report only the feasible paths in $\overline{\mathcal{A}}$.

It is worth noting that Octopus can generate the same path condition as other variants (e.g., bottom up analysis [55]) of the analysis targeting the same source-sink problem. As demonstrated by Theorems 1 and 2, Octopus collects the same value-flow paths as other algorithms [53, 55] do. By recovering the path conditions on the same GVFG, which is the input of our problem, the recovered path conditions are also the same for any value-flow paths. Therefore, the logical disjunctions of all the path conditions are equivalent to the one obtained by the analysis of another variant, which provides the correctness guarantee of our algorithm.

Besides, it is remarkable that we only compute the path conditions for realizable segment paths without introducing redundant computation. Specifically, we only compute the path conditions of each intraprocedural path summarized by a value-flow segment occurring in a realizable segment path (line 11). For example, assuming ℓ_n

is not present in any segment path, its path condition ϕ_{ℓ_n} will never be computed. Finally, we can formulate the following theorem to demonstrate the potential speedup of Algorithm 4.

Theorem 5. Given the VFSG $\overline{G} = (\overline{V}, \overline{E}, \overline{\mathcal{L}})$, the potential speedup of Algorithm 4 is $\Omega(|\overline{V}|)$, where $\overline{V}$ is the number of value-flow segments.

Proof. According to [54], the time complexity for path condition collection is $O(|V|+|E|)$. Since path conditions of particular values can be cached without recomputing, `getPathCond` also takes at most $O(|V_f| + |E_f|)$ time for each segment ℓ in the function f, and we can have at most $|\overline{V}|$ in total. Thus, we have $T_1 = \Omega(|\overline{V}| * (|V| + |E|))$ since we can clone the constraints for each function multiple times. Similar to the proof of Theorem 3, we have $T_\infty = O(|V| + |E|)$ with sufficient threads. Finally, we have a potential speed up $\frac{T_1}{T_\infty} = \Omega(|\overline{V}|)$. In practice, the $|\overline{V}|$ is unlikely to be smaller than the number of available physical threads of a server, giving a maximum speedup. □

5.4 Summary and Discussion

At the end of the section, we summarize our parallel algorithm Octopus with the highlights of its benefits and potential redundancy issues.

Benefits. Our parallel algorithm Octopus benefits from our realizability decomposition presented in § 4, which enables us to decouple the path condition collection of single functions from the realizability reasoning, improving overall efficiency from the following two perspectives:

- First, we only preserve realizable segment paths instead of blindly caching intraprocedural paths in the GVFG, avoiding redundantly collecting value-flow paths that never form a realizable source-sink path.
- Second, all three stages can be parallelized in balance either by task composition and adaptions to parallel graph traversal algorithms, yielding an efficient parallel algorithm for our problem.

Particularly, the potential speedups of Algorithms 2 and 4 are determined by the number of sources V_{src}, which is of great quantity according to our evaluation (§7). Meanwhile, the potential speedup of Algorithm 3 is determined by the number of interprocedural segment paths from BFS, which is even larger than the number of segments.

Redundancy. Although we avoid unnecessary intraprocedural path exploration in the first stage, the redundancy issue still exists in the value-flow segment graph generation, as not all the segments can contribute to a realizable segment path. For example, we generate the value-flow segments ℓ_3 redundantly for the example program in Figure 1 as Coyote does. However, such redundancy does not bring significant overhead in the parallelization for the following three reasons:

- First, the value-flow segments starting from a given source are generated by a linear-time graph traversal. Algorithm 2 takes at most quadratic time to the size of the GVFG of a single function, which is generally smaller than the complexity of solving the CFL-reachability problem.
- Second, the value-flow segment graph generation can be fully parallelized at the function level because the computations are intraprocedural. Given adequate physical CPU cores, the time overhead of computing redundant value-flow segments can be negligible.
- Third, the value-flow segments that are redundantly generated do not participate in the concatenation of path conditions in Algorithm 4, which does not propagate the redundancy in the subsequent stages.

Therefore, the redundancy issue does not bother Octopus's performance. Instead, it gains maximum potential speedup via completely relaxing caller-callee dependencies when collecting value-flow segments, unlike previous work, can have limited speedup due to partial relaxation [55].

6 INSTANTIATION AND IMPLEMENTATION

This section presents two instantiations of value-flow analysis and provides the implementation details.

6.1 Instantiations

To show the capability of Octopus, we instantiate two clients, namely NPD detection and taint analysis. Particularly, both analyses are context, flow-sensitive, and path-sensitive value-flow analysis, featuring the high precision in the bug detection. In what follows, we demonstrate more details of instantiating the two clients.

NPD Detection. As discussed in Example 3.4, we need to instantiate the unary predicates $\mathsf{mayNull}(v)$ and $\mathsf{deref}(v)$ in the NPD Detection. Apart from explicitly specifying the `NULL` values as the sources for instantiating the predicate $\mathsf{mayNull}(v)$, we additionally specify the library function calls that could return a `NULL` value as sources. For example, we regard the return values of malloc-like memory allocation functions as the sources, such as malloc, memcpy, and memset, as the memory allocations may fail in a concrete execution of the program. Also, we set $\mathsf{deref}(v)$ to be true for the pointers dereferenced by the instructions. Similarly, we regard the parameter passed to library function calls that would be dereferenced later as sinks. Benefiting from Octopus, we can determine if sources and sinks are reachable by collecting the realizable path conditions efficiently to Ψ and solving them with an SMT solver with high precision. Meanwhile, the paths from $\overline{\mathcal{A}}$ allow the developer to diagnose the bug if the path conditions are satisfiable.

Taint Analysis. To demonstrate our broad applications, we also instantiate taint analysis [3, 4] based on Octopus. Similar to NPD detection, we need to specify the forms of sources and sinks, while they often vary in different taint bugs. For example, when we detect *relative path traversal* bugs [15], we should model the uncontrolled inputs as the sources, such as the return value of fgets, and regard the directory variables used in the file open operations as the sinks. Particularly, the propagation of values often involves library functions, of which the semantics is summarized by the taint specification [3, 12]. Intuitively, the taint specification decides whether or not an edge should be added to the GVFG. Lastly, we often need to resolve the sanitizers, which can terminate the value propagation. Benefiting from our parallel path- and context-sensitive analysis, developers can efficiently validate if certain program locations can contain sensitive values.

6.2 Implementation

We have implemented Octopus and the two instantiations on top of the LLVM framework [30] and use Z3 [18] as the default theorem prover to analyze C/C++ programs. Currently, Octopus has been integrated into the static analysis platform Pinpoint [53] in Ant Group to assist with the nightly scan of code repositories. For the consideration of commercial use, we will not release the artifact of Octopus upon acceptance. To convenience the reproduction of our results in the future study, we provide more implementation details of Octopus as follows.

Parallelization. The number of tasks generated in Algorithm 3 can be extremely large since the concatenation can happen simultaneously many times. Instantiating too many workers can easily decrease the performance of thread pools due to context-switching [14]. To resolve the performance issue, we implement a customized thread pool to support task submission in batch to avoid performance drop. In our implementation, we dynamically adjust the batch size according to the number of available threads. Specifically, the number of tasks in a batch equals to the number of all the subtasks divided by the number of available threads. Meanwhile, we also adopt the work-stealing algorithm [8] in our batch-enabled thread pool. That is, if a batch is empty, it can automatically steal tasks from other batches. Such design can keep maximum efficiency at all times effectively.

Path sensitivity. To achieve the best efficiency for path-sensitive analysis, we also adopt the *no-caching* strategy such that the path conditions are never encapsulated in the intermediate summaries [54]. In contrast, we only recover the path conditions when a path is obtained. Evidence has shown that such a strategy can speed up path-sensitive analysis by several times and avoid redundant computation and potential out-of-memory issues in practice [54]. To make a fair comparison, this strategy is applied to both the baseline approach Coyote [55] used

for evaluation and our algorithm (detailed in Algorithm 4). For practicality reasons, we set the solver timeout to 10 seconds per query for both Coyote and Octopus.

Soundiness. Octopus does not target the rigorous program verification, and thus, makes a few reasonably unsound choices (i.e., soundy [35]) following the previous tools [1, 53, 55]. For instance, we unroll each loop twice on the control flow graph, do not handle inline assembly, and manually model several but not all standard C/C++ libraries. When constructing the GVFG, we use a flow-insensitive pointer analysis to resolve the function pointers [76] and adopt the assumption that the pointer parameters of any function are not aliased, which are commonly adopted in many existing studies [1, 53].

7 EVALUATION

To evaluate the effectiveness of Octopus, we investigate the following research questions:

- **(RQ1)** What is the speedup ratio of Octopus? (§ 7.2)
- **(RQ2)** Can Octopus outperform the most recent parallel analysis regarding efficiency and memory? (§ 7.3)
- **(RQ3)** What is Octopus's CPU utilization rate? (§ 7.4)

Result Highlights. In summary, our parallel algorithm stands out with the following features:

- *High speedup.* In the NPD detection, Octopus achieves at most 5.2× speedup with 40 threads compared to the analysis with 10 threads. The speedup is almost linear to the available threads, posing its sustainable scalability in analyzing large-scale programs with more CPU cores.
- *Overwhelming superiority over the SOTA parallel analysis.* Octopus achieves 123.1× average speedup compared with the SOTA parallel value-flow analyzer [55] under the 40-thread setting for NPD detection.
- *Almost full CPU utilization.* Octopus leverages the computation resources of multi-core machines thoroughly, almost occupying the CPU cores during the whole analysis process.

As a general value-flow analysis framework, Octopus has been deployed in the international company, Ant Group, conducting the code quality and security scan in a scheduled manner.

7.1 Experimental Setup

Baselines. We compare Octopus against the SOTA parallel context- and path-sensitive value-flow analysis Coyote [55], which improves the bottom-up-style parallelism of Pinpoint [53]. Additionally, as also explained in §6.2, our approach does not cache SMT constraints until the last stage like previous work [51]. To have a fair comparison, we also apply its *no-caching* strategy to Coyote to make a fair comparison. Without loss of generality, we choose NPD detection as our client. We do not compare with pure CFL-reachability approaches, as we have not found such publicly available tools resolving context- and path sensitivity simultaneously. However, the comparison with Coyote should be sufficient to show our strength and demonstrate our contribution to the program analysis community. To show the generality of our approach, we also evaluate the performance of Octopus in the taint analysis and compare it with Coyote of which the result is briefed at the end of the evaluation. To remove the impact from the execution environment, we repeat the experiments in each stage and calculate their average performance and speedup.

Subjects. To show our wide applicability in analyzing diverse real-world applications, we adopt all large programs (>100 KLoC) of SPEC CPU® 2017 benchmark [9]. In addition, we select three up-to-date open-source programs whose sizes are also greater than one MLoC, namely postgres, ffmpeg, and llvm, which are popular and fundamental software systems. All the evaluation subjects are shown in Table 1. Notably, we leverage GLLVM [57] to compile each project into a whole-program LLVM bitcode, which is fed to Octopus as its input for analysis.

Environment. Our experiments are conducted on a workstation with forty "Intel® Xeon® CPU E5-2698 v4 @2.20GHz" physical processors and 256GB of memory running Ubuntu 20.04. To have a fair comparison, we

Table 1. Statistics of evaluated SPEC CPU® 2017 and open-source programs. $|\mathbf{F}|$ denotes the number of functions, $|\overline{\mathbf{V}}|$ denotes the number of value-flow segments in their VFSGs for the NPD detection, and $|\overline{\mathbf{V}_{\text{src}}}|$ denotes the number of value-flow segments starting with a NULL value.

Source	ID	Program	KLoC	$\|\mathbf{F}\|$	$\|\overline{\mathbf{V}}\|$	$\|\overline{\mathbf{V}_{\text{src}}}\|$	Application Area
SPEC CPU® 2017	1	wrf	130	882	9,106	1,281	Weather Forecasting
	2	omnetpp	134	14,128	73,048	12,551	Discrete event simulation
	3	povray	170	2,782	32,089	9,255	Ray tracing
	4	cactuBSSN	257	3,893	134,572	19,623	Physics
	5	imagick	259	3,886	58,983	18,215	Image manipulation
	6	pop2	338	1,163	10,813	1,671	Ocean modeling
	7	perlbench	362	4,557	59,529	73,591	Perl interpreter
	8	cam4	407	953	9,861	1,548	Atmosphere modeling
	9	parest	427	58,045	441,717	16,947	Biomedical imaging
	10	xalancbmk	520	37,480	218,608	44,937	XML Converter
	11	gcc	1,304	39,809	271,924	266,418	GNU C Compiler
	12	blender	1,577	52,871	372,402	166,647	3D renderering
Open Source	13	postgres-14.6	1,149	21,652	408,577	96,888	Database system
	14	ffmpeg-5.1.2	1,346	18,725	434,519	53,139	Multimedia processing
	15	llvm-3.6.2	6,347	309,467	2,718,508	132,558	Low-level virtual machine

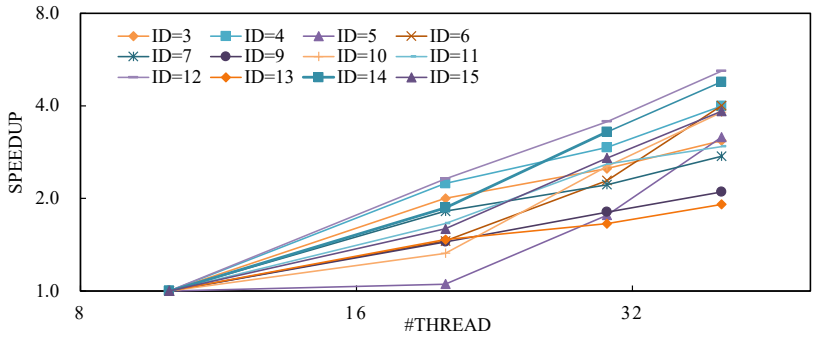


Fig. 5. Speedup of Octopus with different threads over a 10-thread setting on NPD detection. The x-axis and y-axis refer to the number of threads and the speedup at the logarithmic scale. We omit the projects wrf, omnetpp, and cam4 in Figure 5 as their analysis time is no more than ten seconds. All the stages finish before the thread pool is fully occupied, making it difficult to measure the speedup.

adopt the same design choices as Coyote to conduct experiments, as briefed in § 6.2. In order to mitigate the possible runtime performance difference brought by the multi-core CPU scheduling, we repeat 10 times and take their average time and memory consumption in the experiments introduced below.

7.2 RQ1: Speedup

To measure the speedup of our parallel algorithm, we choose the running time of the 10-thread setting T_{10} as the baseline and measure the running time over T_{10}. This is because if using one thread, the time for analyzing many projects would exceed the 12-hour timeout. Notably, the running time includes the time of realizable path

Table 2. Octopus (O)'s running time (sec), Coyote (C)'s running time (sec), and Octopus's speedup over Coyote (↗) on NPD detection. Similar to Figure 5, we ignore three projects wrf, omnetpp, and cam4 in the calculation of the average speedup.

ID	# Thread = 10			# Thread = 20			# Thread = 30			# Thread = 40		
	O	C	↗	O	C	↗	O	C	↗	O	C	↗
1	1	600	600.0	1	484	484.0	1	409	409.0	1	402	402.0
2	12	655	54.6	7	599	85.6	4	577	144.3	3	584	194.7
3	80	2,510	31.4	40	2,153	53.8	32	2,088	65.3	26	1,982	76.2
4	1,194	2,024	1.7	534	1,432	2.7	408	1,582	3.9	299	924	3.1
5	60	4,875	81.3	57	3,307	58.0	34	2,935	86.3	19	2,730	143.7
6	16	619	38.7	11	479	43.5	7	437	62.4	4	399	99.8
7	1,464	7,664	5.2	806	6,942	8.6	661	6,323	9.6	534	5,804	10.9
8	1	666	666.0	1	492	492.0	1	433	433.0	1	380	380.0
9	65	9,933	152.8	45	9,861	219.1	36	9,470	263.1	31	9,400	303.2
10	61	2,778	45.5	46	2,427	52.8	24	2,278	94.9	16	2,216	138.5
11	12,438	23,320	1.9	7,506	17,323	2.3	4,809	14,502	3.0	4,224	13,138	3.1
12	8,190	14,676	1.8	3,536	10,820	3.1	2,305	9,409	4.1	1,580	8,646	5.5
13	867	12,676	14.6	590	8,953	15.2	524	7,445	14.2	454	6,816	15.0
14	6,262	25,923	4.1	3,359	17,694	5.3	1,906	13,443	7.1	1,308	9,699	7.4
15	1,591	37,301	23.4	999	31,630	31.7	588	28,615	48.7	414	26,127	63.1
Avg			114.9			103.8			109.9			123.1

conditions and constraint solving. We then measure the running time of Octopus under different threads, namely 20-thread, 30-thread, and 40-thread settings. Furthermore, we compute the speedup ratio by $S = \frac{T_{10}}{T_p}$, where T_p is the running time under the setting with p thread. Finally, we obtain S under different available threads.

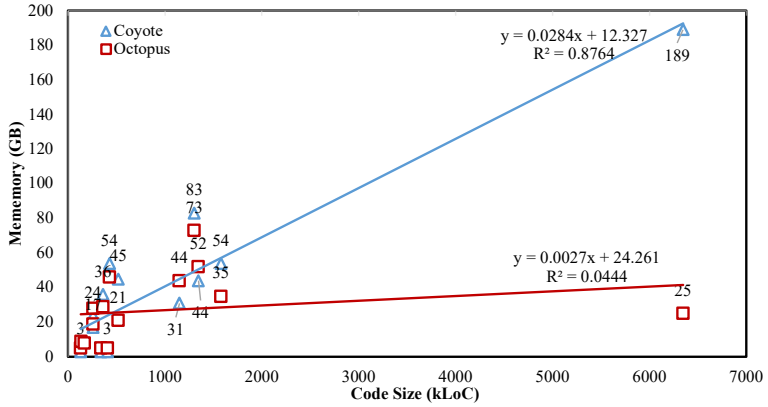
Figure 5 presents the details of the speedup under different settings. Obviously, the results show that the more threads allocated, the higher speedup will be gained. In particular, we achieve a 1.9-5.2× speedup under 40-thread settings compared to 10-thread settings. For example, Octopus gains 5.2× speedup [2] running 40 threads compared to that running ten threads when analyzing the project blender (ID=12) with 1.5 MLoC. The results also indicate that we can gain an almost-linear speedup when the number of threads grows.

7.3 RQ2: Scalability

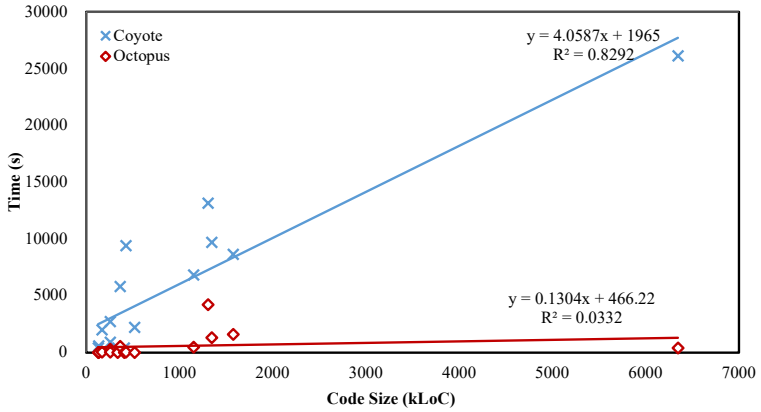
To compare the scalability of Octopus and Coyote for each project's analysis, we measure the running time and peak memory of both tools under different thread settings in relevance to their code size. Then we compute the speedup and memory consumption of Octopus over Coyote under each setting, quantifying the scalability improvement of Octopus. Similar to 7.2, we refer to the overall analysis time as the running time.

Table 2 shows the detailed running time statistics and its speedup ratios. Figure 6a displays the scalability improvement on a 40-thread setting in a straightforward manner. It is observed that, in general, Octopus improves scalability greatly when the code size increases. In particular, the speedup over Coyote under the 10-thread setting is 114.8× on average, and the speedup reaches 123.1× when using 40 processors. Moreover, the maximum speedup is 402.0× on the project wrf, a medium-sized program, under the 40-thread setting. Besides, the speedup over Coyote under the 40-thread setting reaches 67.8× when analyzing the open-source project llvm, which has

[2]The superlinear speedup is because, with more threads allocated, there is a higher possibility that solving intraprocedural path conditions can prune the feasibility of an interprocedural one. The implementation of the baseline Coyote and Octopus solve the same stage-solving strategy in constraint solving.



(a) Scalability in terms of analysis time



(b) Scalability in terms of analysis memory

Fig. 6. Scalability of Octopus and Coyote on a 40-thread setting for NPD detection.

over 6 MLoC. Such speedup is significant enough to support the efficient static analysis clients on the programs with million lines within a few minutes, promoting its potential for CI/CD integration [74]. In contrast, Coyote has to spend almost two hours analyzing the project postgres even with 40 threads, while it only takes around seven minutes for Octopus to finish the analysis under the same setting. We can observe that the scalability of Coyote is almost linear ($R = 0.8292$) to the code size while Octopus remains almost constant, demonstrating that Octopus is insensitive to the code size.

An interesting observation that Octopus's speedup over Coyote is not always increasing significantly. One example that the speedup is significantly increasing is when analyzing povray (ID=3), where Coyote requires 2,153, and 2,088 on 20, 30, and 40 threads, and the corresponding speedup does increase significantly in the order of 53.8×, 65.3×, 76.2×. However, sometimes the speedup over Coyote is not that obvious. For example, analyzing postgres (ID=13) at 20 and 40 threads only takes 590 and 454 seconds and yields a speedup of 15.2× and 15.0× to CoyoteİIt can be observed that the time spent in the 40-thread setting is not half of that in a 20-thread setting, but the number is rather close. This is because in the case of Octopus, as discussed in Theorem 4 in § 5.2, the

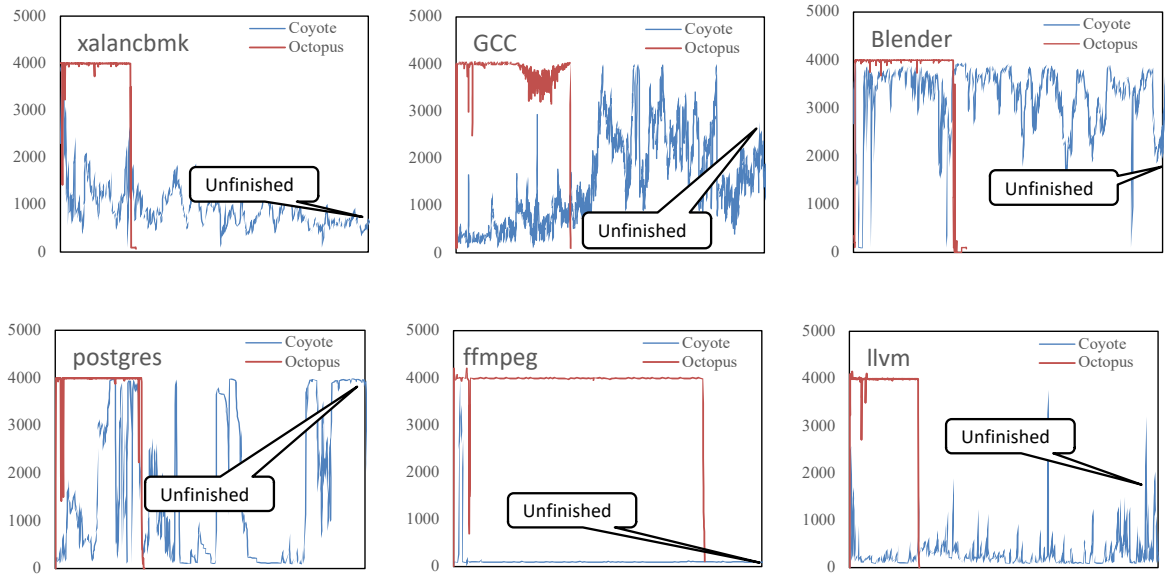


Fig. 7. CPU utilization rate of Octopus and Coyote on the NPD detection. The x-axis and y-axis stand for the time consumption and the CPU utilization rate (e.g., 4000% means fully occupying all forty cores), respectively. To show the difference clearly, the CPU utilization of Coyote is not completely presented.

potential speedup is $\Omega(\frac{|\overline{V_{\text{src}}}|\cdot(|\overline{V}|+|\overline{E}|)}{\overline{D}})$, indicating that when there is a chain-like VFSG that is typically owned by a small-scale program, the overall speedup can be limited to $\Omega(|\overline{V_{\text{src}}}|)$, which becomes the main obstacle to keeping a stable and linear speedup. Considering both cases exist in our experiments, the average speedup (table 2) is not increasing significantly. However, our evaluation has shown a significant speedup indicating higher scalability than Coyote.

Figure 6b also depicts the memory consumption of the two tools on a 40-thread setting. On average, Octopus consumes 33% less memory than Coyote because the redundancy can only happen in the intraprocedural segment generation, generally lower than the redundancy of Coyote as discussed in § 5.4. In particular, when analyzing the project llvm, we can save at most 164GB (= 189 − 25) memory when performing NPD detection, yielding an 86.7% memory reduction. However, there are cases where Octopus consumes more memory, because Octopus can occupy all threads and achieve the highest CPU utilization during the whole analysis, each thread must allocate enough memory for the computation. In contrast, Coyote cannot occupy all threads in many cases, e.g., when analyzing the project postgres, and thus can consume less memory.

To summarize, it is reasonable to draw the conclusion that our parallel algorithm is overwhelmingly more scalable than Coyote when analyzing real-world programs. The results also imply parallelizing the collection of realizable path conditions significantly improves the overall efficiency in the context- and path-sensitive value-flow analysis.

7.4 RQ3: CPU Utilization

To quantify the CPU utilization rate of Octopus and Coyote, we utilize Intel© VTune Profiler to measure the CPU time of analysis processes. Specifically, we display the CPU utilization of analyzing the three largest projects in SPEC CPU® 2017 benchmark and the three selected open-source programs.

Figure 7 shows the utilization against the running time of the two projects. We observe that Coyote cannot occupy all CPU cores during the analysis because, for many projects, the call graph is irregular and similar to a chain in shape, i.e., each function calls only a few functions. In such situations, there are few cases where the two analysis tasks do not depend on each other at the same time, yielding low parallelism. Even in the middle of the analysis, drop-offs in CPU utilization ratio are frequently observed. The parallelism will be extremely limited if the call graph is similar to a chain in shape, i.e., each function calls only a few functions. In this case, the independence of functions can hardly exist during the analysis lifecycle. Even in the middle of the analysis, drop-offs in CPU utilization ratio are frequently observed. In contrast, Octopus generates the value-flow segments of single functions simultaneously, of which each worker does not depend on the other. Also, the collection of realizable segment paths and the path conditions are also highly parallelized due to the independence of the search tasks. Although we observe drop-offs occasionally occur near the end of the analysis due to path condition solving, Octopus almost occupies all CPU cores during the whole analysis process.

7.5 Results of Taint Analysis

To show the generality of Octopus, we also conduct experiments on taint analysis for the experimental subjects. For simplicity, we only show the detailed performance statistics of Octopus of three open-source projects in Table 3, all of which have more than 1 MLoC. We surprisingly find that Octopus finishes taint analysis upon any project around or within one minute under each of the four settings. In contrast, Coyote has to spend over 3 hours analyzing the project ffmpeg with 40 threads, and the analysis exceeds the 12-hour time budget when analyzing llvm under all settings. The speedup ranges from 733.4× to 2,166.2× when analyzing the first two projects under different settings, demonstrating Octopus's superiority for the taint analysis.

According to Table 3, we also find that the speedup over Coyote of the taint analysis is much greater than that of the NPD detection. As shown in the columns $|\overline{\mathbf{V}_{\text{src}}}|$ of Table 3 and Table 1, the taint bugs are generally smaller in quantities compared with NPD bugs because they have fewer value-flow segments starting from the sources in the taint analysis than the NPD detection. For example, postgres gots only 1, 146 taint source segments compared to 12, 671 source segments for NPD detection, eventually yielding only 9 realizable and feasible segment paths. Moreover, Octopus achieves overwhelming speedup over Coyote because as exemplified in § 5.2, it can ignore the concatenation of segments not contributing to the taint bugs, thus avoiding redundant computation and achieving overwhelming speedups. However, Coyote needs to capture more value-flows due to the modeling of library functions in the taint analysis, dramatically limiting its performance such that none of the settings could finish under the 10-hour time limit when analyzing llvm. Octopus itself cannot achieve linear speedup because the analysis time is relatively short. The benefit of parallelization cannot be fully exposed since the thread pool is not fully occupied during the analysis. However, Coyote still has to explore unnecessary value-flow paths in a bottom-up manner, which cannot be fully parallelized, finally introducing high overhead in the taint analysis.

7.6 Comparison of Value-Flow Paths

Octopus works on the same precision as Coyote as discussed in § 6.2 and both have been applied with no-cache stragety [54]. Therefore, the reported number of feasible value-flow paths and their precision should be almost the same. For instance, Octopus reported 15,228, 8,158, and 78,389 feasible value-flow paths for NPD in the real-world project postgres, ffmpeg, and llvm. On the other hand, Coyote reported to have 15,666, 8,369, and

Table 3. The running time (sec) and speedup of taint analysis. "OOT" stands for out-of-time under the budget of 12 hours.

ID	$\lvert\overline{V}\rvert$	$\lvert\overline{V_{src}}\rvert$	$\lvert\overline{\mathcal{A}}\rvert$	# Thread = 10			# Thread = 20			# Thread = 30			# Thread = 40		
				O	C	↗	O	C	↗	O	C	↗	O	C	↗
13	62,815	1,146	9	9	10,592	1,176.9	9	8,000	888.9	8	6,706	838.3	8	5,867	733.4
14	52,065	1,412	203	13	28,160	2,166.2	13	18,612	1,431.7	12	14,561	1,213.4	13	12,231	940.8
15	2,037,891	346	9	75	OOT	N/A	68	OOT	N/A	63	OOT	N/A	57	OOT	N/A

79,281 in postgres, ffmpeg, and llvm [3]. After re-running a few times, the number changes again delicately. By profiling both tools, we identify that the minor difference observed in these projects is because of the solver timeouts that may or may not be triggered by different runs in a parallel environment. If constraint solving reaches a timeout, we conservatively treat it as an unsat result and reject the value-flow path. Therefore, the number will vary from one run to another.

7.7 Threats to Validity

We analyze the threats to validity of Octopus from the following three aspects:

Construct Validity. The construct validity mainly comes from the choice of metrics for the performance evaluation. We currently select CPU usage to measure the runtime efficiency for the analysis in parallel. It is also possible that other issues can cause high CPU usage. For example, CPU spin time can also cause high CPU usage but does not make actual work. Nevertheless, we have shown that all our Algorithms 2-4 is unlikely to have data races or long wait of dependent subtasks. Therefore, it is implausible that our approach can suffer from CPU-time-related construct validity.

Internal Validity. The internal validity mainly comes from the selection of checkers and their corresponding implementation used for evaluation. We currently select NPD detection and taint analysis as the two checkers for our evaluation. The overall performance and potential speedup might vary from one checker to another. However, the checker we used is the representative checker from a prior work [55] and has a standard specification (Section 6.1). Besides, we follow the SOTA path-sensitive analysis [54] to collect the relevant paths first and then recover their path conditions. If implemented with an online pruning strategy, i.e. solving the constraints to prune infeasible paths during the summarization, the performance might also vary from one program or checker to another. Nevertheless, the SOTA [54] has shown its efficiency in analyzing real-world applications. Under such settings, we can continuously improve the scalability of path-sensitive program analysis.

External Validity. The external validity mainly comes from the subject selection and the running environment. The subjects are representatives of real-world programs from SPEC®CPU 2017 and open-source repositories. The performance can still vary from analyzing one subject to analyzing another. Besides, the running environment is a multi-core CPU with a 2.2GHz main frequency for each physical core. A more or less powerful CPU might also affect the evaluation results. Nevertheless, both our theoretical analysis and empirical data have shown that it is unlikely that our approach is less scalable than the traditional approach. In addition, more significantly increasing speedups over Coyote may be observed if a more efficient thread pool implementation is integrated to mitigate the thread initialization/deinitialization overhead.

[3]The number of feasible value-flow paths is not equivalent to the number of bugs, because one bug can be triggered via multiple value-flow paths via different value-flow nodes on the value-flow graph in the SSA form. Therefore, we compare the feasible paths to demonstrate that Octopus can generate the almost same results as the baseline.

7.8 Future Work

Beyond our parallel analysis, we notice our insight into the realizability decomposition can further benefit the value-flow analysis from two other aspects.

- First, we can hopefully achieve nearly real-time incremental analysis. Unlike the bottom-up analysis, our value-flow segment graph generation is unaware of the call graph. Therefore, the incremental analysis based on our infrastructure only needs to analyze impacted functions when software evolves, which can further advance towards the goal of deploying static analyzers in the CI/CD workflow [74].
- Second, our realizability decomposition makes it possible to divide the whole analysis into almost arbitrary tasks upon the cloud clusters. The standardized, containerized, and virtualized clusters with lower hardware resources become the most common environment for the best elasticity in a typical industrial setting [34]. As demonstrated by the answer to RQ1, the almost linear speedup of Octopus demonstrates its practical potential of achieving better performance with a large number of physical processors.

8 RELATED WORK

There is a vast body of studies closely relevant to our work, covering various areas of static program analyses. In what follows, we discuss the works of each area in detail.

8.1 Parallel Program Analysis

Parallel algorithms for static analysis is an active area of research. One typical line of the works concentrates on the parallelization of pointer analysis [7, 37, 40, 41, 61]. For example, Méndez-Lojo et al. [41] conducts the inclusion-based pointer analysis via graph rewrite rules to expose the amorphous data parallelism, which supports the parallel computation of points-to information. Su et al. [61] parallelizes a CFL-reachability-based pointer analysis via data sharing and query scheduling, which achieves field and context sensitivity efficiently. Another line of the related works targets a variety of parallel client analysis [49, 55, 66, 69, 79]. Particularly, Coyote [55] is the most closely-relevant work to Octopus. Similar to Octopus, Coyote decomposes the function summaries into more fine-grained ones and forms three kinds of intra-procedural value-flow summaries, which support inlining part of function summaries in advance even if the function summaries of callees are not fully prepared. However, Coyote is still restricted by the call graph structure, preventing it from achieving higher potential speedup. Wang et al. [69] and Zuo et al. [79] propose a disk-based parallel graph system to perform the flow and context-sensitive analysis, which reduces the consumption of time and memory simultaneously. In the future, it is also worth reducing the memory cost of our approach from the perspective of the system design, i.e., storing the value-flow segments in the disk and loading them on demand during the traversal, which can further promote the practical value of Octopus.

8.2 CFL-Reachability-based Analysis

CFL reachability has been broadly used in program analysis for a wide range of applications, such as context-sensitive data flow analysis [47], program slicing [48], shape analysis [46], type-based flow analysis [27, 42], and pointer analysis [58, 60, 70, 71, 76–78]. Most existing work labels the edges of specific graphs to enforce the CFL-reachable paths to depict desired properties. For example, several Anderson-style pointer analyses label the edges in the pointer assignment graph with the fields of structures and obtain the field-sensitive alias and points-to facts [60, 70, 76]. Besides, many context-sensitive analyses introduce the parentheses for the edges between callers and callees, where a CFL-reachable path corresponds to a possible control flow path in a concrete execution [47, 51, 67]. Octopus bears similarity with these context-sensitive analyses but considers path-sensitivity in the meanwhile. Besides, it poses the opportunity of scaling the analysis up by parallelization.

8.3 Compositional Analysis

There is a large amount of literature on compositional analyses, which reuses information within a procedure as summaries for better efficiency [1, 5, 39, 43, 47, 50, 53, 65]. According to the order of analyzing each function on a call graph, their summaries can be divided into two categories, namely top-down and bottom-up summaries. Top-down-based approaches [43, 47, 50] compute summaries on a specific program context, and reuse the compatible summaries when encountering the same context. Most program analyses nowadays prefer a bottom-up fashion [1, 5, 39, 53, 65]. They symbolize the effect of a function with bottom-up summaries and inline the summary of a callee into the summary of its caller, which avoids analyzing a single function redundantly. As indicated by existing studies [2], a bottom-up summary design provides better parallelism than top-down ones. However, the bottom-up style parallelism is restricted by the call graph structure [55], as a caller's summary cannot be fully determined until all of its callees' summaries are obtained. Our work achieves full parallelism by designing a fine-grained function summary, i.e., value-flow segments, and thus, can obtain better scalability with an affordable number of processors.

8.4 Value-Flow Analysis

The past few decades have witnessed the increasing popularity of value-flow analysis [26, 53, 62, 64]. It concentrates on the program dependencies to support various static analysis clients, such as the program optimization [26], program understanding [59], and bug detection [53, 63]. Particularly, many critical vulnerabilities, including NPD [52], memory leak [22], and taint bugs [4], can be effectively detected by finding realizable value-flow paths. Thus, Octopus can benefit a variety of value-flow analysis clients significantly, improving their efficiency in analyzing large-scale programs. Similar to Octopus, existing value-flow analyses have to establish a value-flow graph as the first stage [53, 64, 68] or resolve the value propagation induced by pointers during the graph traversal [23, 56], which demands the pointer analysis to propagate value flows. In our work, Octopus achieves the parallel collection of path conditions for realizable paths while still relying on a serial version of the GVFG construction. Unfortunately, the current flow-sensitive value-flow analysis [64] necessitates a flow-insensitive VFG in the preprocessing stage, which might introduce extra redundancy. In addition, it handles strong updates using on-demand graph traversal while simultaneously deleting edges on the VFG and propagating changes, making it difficult to search from different sources simultatenously following our approach. However, it would be quite a promising direction to design a parallel algorithm for constructing the value-flow graph as an extension such that the overall analysis is designed in a fully demand-driven but parallel manner, where the efficiency could even improve.

9 CONCLUSION

We have presented Octopus, a parallel approach to collect realizable path conditions to scale value-flow analysis. Based on realizability decomposition, Octopus decouples the intraprocedural path condition collection from the realizability reasoning. We have proved that Octopus can obtain a high potential speedup for each stage theoretically. The evaluation also evidences that Octopus achieves a full utilization of an arbitrary number of CPU cores and accelerates the SOTA parallel value-flow analysis by 123.1× on average for the NPD detection with 40 cores and the memory cost is averagely decreased. We believe Octopus not only enables faster whole-program analysis on workstations but innovates incremental and distributed analysis in the future.

ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. Eric Bodden and other anonymous reviewers for providing valuable feedback on earlier drafts, which helped to improve the quality of our presentation. We also thank Dr. Xiao Xiao for his insightful discussions.

REFERENCES

[1] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An overview of the saturn project. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 43–48. https://doi.org/10.1145/1251535.1251543

[2] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. 2012. Parallelizing top-down interprocedural analyses. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 217–228. https://doi.org/10.1145/2254064.2254091

[3] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 725–735. https://doi.org/10.1145/2884781.2884816

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[5] Domagoj Babić and Alan J. Hu. 2008. Calysto: Scalable and precise extended static checking. *Proceedings - International Conference on Software Engineering* (2008), 211–220. https://doi.org/10.1145/1368088.1368118

[6] David A. Bader and Kamesh Madduri. 2006. Designing multithreaded algorithms for Breadth-First Search and si-connectivity on the Cray MTA-2. *Proceedings of the International Conference on Parallel Processing* (2006), 523–530. https://doi.org/10.1109/ICPP.2006.34

[7] Thorsten Blaß and Michael Philippsen. 2019. GPU-Accelerated fixpoint algorithms for faster compiler analyses. *PervasiveHealth: Pervasive Computing Technologies for Healthcare* (2019), 122–134. https://doi.org/10.1145/3302516.3307352

[8] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (sep 1999), 720–748. https://doi.org/10.1145/324133.324234

[9] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[10] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 30:1–30:30. https://doi.org/10.1145/3158118

[11] 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 (PLDI '07)*. ACM, 480–491. https://doi.org/10.1145/1250734.1250789

[12] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable Taint Specification Inference with Big Code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 760–774. https://doi.org/10.1145/3314221.3314648

[13] Dmitry Chistikov, Rupak Majumdar, and Philipp Schepper. 2022. Subcubic certificates for CFL reachability. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.

[14] Douglas Comer. 2011. *Operating System Design: The Xinu Approach, Linksys Version* (1st ed.). Chapman & Hall/CRC.

[15] CWE-23. 2022. Common weakness enumeration. https://cwe.mitre.org/data/definitions/23.html

[16] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 25–35.

[17] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 57–68. https://doi.org/10.1145/543552.512538

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[19] Lisa Nguyen Quang Do and Eric Bodden. 2022. Explaining Static Analysis with Rule Graphs. *IEEE Transactions on Software Engineering* 48, 2 (2022), 678–690. https://doi.org/10.1109/TSE.2020.2999534

[20] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2020. Debugging Static Analysis. *IEEE Transactions on Software Engineering* 46, 7 (2020), 697–709. https://doi.org/10.1109/TSE.2018.2868349

[21] Manuel Fähndrich and K Rustan M Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. 302–312.

[22] 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 (ICSE '19)*. IEEE, 72–82. https://doi.org/10.1109/ICSE.2019.00025

[23] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 102:1–102:28. https://doi.org/10.1145/3133926

[24] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 271–283. https://doi.org/10.1145/3377811.3380323

[25] Susan Horwitz, Alan J. Demers, and Tim Teitelbaum. 1987. An Efficient General Iterative Algorithm for Dataflow Analysis. *Acta Informatica* 24, 6 (1987), 679–694. https://doi.org/10.1007/BF00282621

[26] Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand interprocedural dataflow analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 104–115. https://doi.org/10.1145/222124.222146

[27] John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, 207–218. https://doi.org/10.1145/996841.996867

[28] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391.

[29] Zhengmin Lai, You Peng, Shiyu Yang, Xuemin Lin, and Wenjie Zhang. 2021. Pefp: Efficient k-hop constrained st simple path enumeration on fpga. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1320–1331.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO '04)*. IEEE, 75:1–75:12. https://doi.org/10.1109/CGO.2004.1281665

[31] Charles E. Leiserson and Tao B. Schardl. 2010. A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers). In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 303–314. https://doi.org/10.1145/1810479.1810534

[32] 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)*. ACM, 343–353.

[33] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-Sensitive and Alias-Aware Typestate Analysis for Detecting OS Bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 859–872. https://doi.org/10.1145/3503222.3507770

[34] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. 2021. The Serverless Computing Survey: A Technical Primer for Design Architecture. *CoRR* abs/2112.12921 (2021). arXiv:2112.12921 https://arxiv.org/abs/2112.12921

[35] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møler, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[36] Linghui Luo, Eric Bodden, and Johannes Späth. 2019. A Qualitative Analysis of Android Taint-Analysis Results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 102–114. https://doi.org/10.1109/ASE.2019.00020

[37] Anders Alnor Mathiasen and Andreas Pavlogiannis. 2021. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 34. https://doi.org/10.1145/3434315 arXiv:2006.01491

[38] Stephen McCamant and Michael D. Ernst. 2008. Quantitative Information Flow as Network Flow Capacity. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 193–205. https://doi.org/10.1145/1375581.1375606

[39] Scott McPeak, Charles Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 - Proceedings*. 554–564. https://doi.org/10.1145/2491411.2501854

[40] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, 107–116. https://doi.org/10.1145/2145816.2145831

[41] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *Proceedings of the 25th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '10)*. ACM, 428–443. https://doi.org/10.1145/1869459.1869495

[42] Ana Milanova. 2020. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29. https://doi.org/10.1145/3428246

[43] Brian R. Murphy and Monica S. Lam. 2000. Program analysis with partial transfer functions. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (2000), 94–103. https://doi.org/10.1145/328691.328703

[44] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2021. Efficient Hop-Constrained s-t Simple Path Enumeration. *The VLDB Journal* 30, 5 (sep 2021), 799–823. https://doi.org/10.1007/s00778-021-00674-5

[45] Gordon D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science* 1, 2 (1975), 125–159.

[46] Thomas Reps. 1995. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '95)*. ACM, 1–11. https://doi.org/10.1145/215465.215466

[47] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, 49–61. https://doi.org/10.1145/199448.199462

[48] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes* 19, 5 (1994), 11–20. https://doi.org/10.1145/195274.195287

[49] Jonathan Rodriguez and Ondřej Lhoták. 2011. Actor-based parallel dataflow analysis. In *Proceedings of the 20th International Conference on Compiler Construction (CC '11)*. Springer, 179–197. https://doi.org/10.1007/978-3-642-19861-8_11

[50] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

[51] 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.* OOPSLA (2022), 1–31. https://doi.org/10.1145/3563339

[52] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the extensional scalability problem for value-flow analysis frameworks. *Proceedings - International Conference on Software Engineering* (2020), 812–823. https://doi.org/10.1145/3377811.3380346 arXiv:1912.06878

[53] 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. *ACM SIGPLAN Notices* 53, 4 (2018), 693–706. https://doi.org/10.1145/3192366.3192418

[54] 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 2021)*. Association for Computing Machinery, New York, NY, USA, 930–943. https://doi.org/10.1145/3453483.3454086

[55] Qingkai Shi and Charles Zhang. 2020. Pipelining bottom-up data flow analysis. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 835–847. https://doi.org/10.1145/3377811.3380425

[56] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP '16)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 1–26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[57] SRI-CSL. 2023. Whole Program LLVM: wllvm ported to go. https://github.com/SRI-CSL/gllvm. [Online; accessed 19-Jan-2023].

[58] 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 (PLDI '06)*. ACM, 387–400. https://doi.org/10.1145/1133981.1134027

[59] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 112–122. https://doi.org/10.1145/1250734.1250748

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

[61] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel pointer analysis with CFL-reachability. In *Proceedings of the International Conference on Parallel Processing*, Vol. 2014-Novem. 451–460. https://doi.org/10.1109/ICPP.2014.54

[62] 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*. 460–473.

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

[64] Yulei Sui and Jingling Xue. 2018. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Transactions on Software Engineering* 46, 8 (2018), 812–835.

[65] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. *2012 International Symposium on Software Testing and Analysis, ISSTA 2012 - Proceedings* (2012), 254–264. https://doi.org/10.1145/04000800.2336784

[66] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* 35, 10 (2000), 264–280. https://doi.org/10.1145/354222.353189

[67] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, 83–95. https://doi.org/10.1145/2676726.2676997

[68] Chengpeng Wang, Wenyang Wang, Peisen Yao, Qingkai Shi, Jinguo Zhou, Xiao Xiao, and Charles Zhang. 2022. Anchor: Fast and Precise Value-Flow Analysis for Containers via Memory Orientation. *ACM Transactions on Software Engineering and Methodology* (2022). https://doi.org/10.1145/3565800

[69] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, 389–404. https://doi.org/10.1145/3037697.3037744

[70] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09)*. Springer, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

[71] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 155–165. https://doi.org/10.1145/2001420.2001440

[72] 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. *Proceedings - International Conference on Software Engineering* (2018), 327–337. http://doi.acm.org/10.1145/3180155.3180178

[73] Zhang Yang and Eric A. Hansen. 2006. Parallel breadth-first heuristic search on a shared-memory architecture. *AAAI Workshop - Technical Report* WS-06-08 (2006), 33–38.

[74] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. 2021. CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 471–482. https://doi.org/10.1109/ICSME52107.2021.00048

[75] Danfeng Zhang and Andrew C. Myers. 2014. Toward general diagnosis of static errors. *ACM SIGPLAN Notices* 49, 1 (2014), 569–581. https://doi.org/10.1145/2578855.2535870

[76] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 435–446. https://doi.org/10.1145/2491956.2462159

[77] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '14)*. ACM, 829–845. https://doi.org/10.1145/2660193.2660213

[78] 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)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464

[79] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: An evolving graph system for flow-and context-sensitive analyses of million lines of C code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 914–929. https://doi.org/10.1145/3453483.3454085