Boosting Path-Sensitive Value Flow Analysis via Removal of Redundant Summaries

Yongchao Wang, Yuandao Cai, and Charles Zhang
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
Email: {ywanghz, ycaibb, charlesz}@cse.ust.hk

***Abstract*—Value flow analysis that tracks the flow of values via data dependence is a widely used technique for detecting a broad spectrum of software bugs. However, the scalability issue often deteriorates when high precision (i.e., path-sensitivity) is required, as the instantiation of function summaries becomes excessively time- and memory-intensive. The primary culprit, as we observe, is the existence of redundant computations resulting from blindly computing summaries for a function, irrespective of whether they are related to bugs being checked. To address this problem, we present the first approach that can effectively identify and eliminate redundant summaries, thereby reducing the size of collected summaries from callee functions without compromising soundness or efficiency. Our evaluation on large programs demonstrates that our identification algorithm can significantly reduce the time and memory overhead of the state-of-the-art value flow analysis by 45% and 27%, respectively. Furthermore, the identification algorithm demonstrates remarkable efficiency by identifying nearly 80% of redundant summaries while incurring a minimal additional overhead. In the largest *mysqld* project, the identification algorithm reduces the time by 8107 seconds (2.25 hours) with a mere 17.31 seconds of additional overhead, leading to a ratio of time savings to paid overhead (i.e., performance gain) of 468.48 ×. In total, our method attains an average performance gain of 632.1 ×.**

***Index Terms*—value flow analysis, inter-procedural analysis**

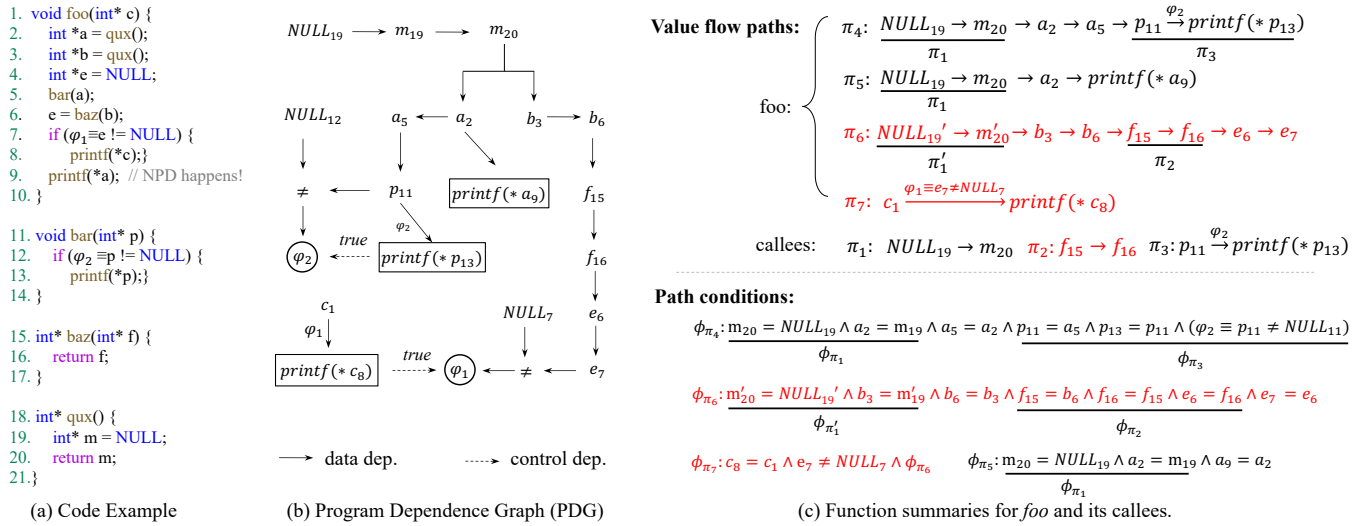
I. Introduction

Path-sensitive value-flow analysis [1]–[11] is highly effective in detecting a broad spectrum of software bugs, such as memory leaks in resource usage, null pointer dereference in memory safety, and the propagation of tainted data in security properties, by tracking the flow of values along data dependence relations. Essentially, detecting these bugs needs to collect feasible source-sink paths over a program dependence graph [1], [3]. For instance, detecting the null pointer dereference (NPD), considering the null value as the source and the pointer dereference statement as the sink. The process involves a two-step process: collecting paths that link a null value and a pointer dereference statement, and then verifying the satisfiability of the path conditions for those paths.

To scale the analysis to large-scale software systems with millions of lines of code, existing approaches [3]–[6], [12]–[14] employ a bottom-up strategy to gather feasible source-sink paths. Specifically, when analyzing a function, these approaches compute the intra-procedural value-flow paths and the corresponding path conditions as function summaries. The value-flow paths and conditions are referred to as summary paths and summary conditions, respectively. To ensure that only feasible summaries are collected, a constraint solver is invoked to verify the summary condition once the summary path is collected, despite being a computationally costly process. To avoid redundant path searching and analysis of callee functions, existing approaches clone the summaries of callees and reuse them continuously to supplement the more extended summaries collected within caller functions. The summary cloning and summary condition verification process continues until the highest function in the call graph (known as the root) is reached. At this point, the algorithm can directly identify source-sink paths by examining summary paths originating from sources and terminating at sinks. Since each summary path carries its corresponding path conditions, we can use the terms "summary" and "summary path" interchangeably without losing generality.

We use the buggy program shown in Fig. 1(a) to illustrate the existing bottom-up compositional value flow analysis. We use the symbol π to represent a value-flow path, while ϕ and φ represent path conditions. Moreover, the variable v_i denotes that variable v is either used or defined at Line i. Specifically, one of the function summaries for *foo*, represented as π_4 in Fig. 1(c), summarizes the propagation path of variable a_2. The variable a_2 receives the return value from the *qux* function in Line 2 and is subsequently passed to the *bar* function in Line 5. On one hand, the summary path π_4 is generated by combining the summaries π_1 and π_3, which are collected during the analysis of the *qux* and *bar* functions, respectively, before analyzing the *foo* function. On the other hand, the summary condition ϕ_{π_4} is obtained by instantiating edges and the corresponding guards along the summary path π_4 [3]. The guard φ_2 of the edge $p_{11} \rightarrow printf(*p_{13})$ is the constraint of $p_{11} \neq NULL_{12}$, which is instantiated when collecing the summary π_3 by traversing from vertex φ_2 on the program dependence graph, shown in Fig. 1(b). Once the summary condition ϕ_{π_4} is instantiated, the summary is verified by the constraint solver *Z3* [15] before it is stored. The summary conditions ϕ_{π_1}, ϕ_{π_2}, and ϕ_{π_3} are also verified when collected. If a summary condition is unsatisfiable(*unsat*), it is discarded to avoid an unfeasible summary being maintained. (We provide such a case in a longer version of this paper [16].) Moreover, the summary π_1 of the *qux* function is cloned twice (cloned one denoted as π_1') with different calling contexts (Line 2 and Line 3) to account for the different propagation paths between

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00195



(a) Code Example (b) Program Dependence Graph (PDG) (c) Function summaries for *foo* and its callees.

Fig. 1: Bottom-up analysis for the code shown in (a). The (b) shows the corresponding program dependence graph (PDG). (c) shows the partial function summaries collected during the bottom-up analysis. Redundant summaries are highlighted in red.

the summaries π_4 and π_6 (which summarizes the propagation path of variable b_3). This cloning mechanism eliminates the necessity of searching for and analyzing the *qux* function again, leading to improved efficiency.

The Explosive Summary Problem. However, the bottom-up approach still faces challenges in terms of analysis time and memory consumption. According to a report [3], a single analysis for a project with millions of lines of code can take several hours and require hundreds of gigabytes (GB) of memory. The main reason for this is the exponential growth in the size of summaries due to cloning for different calling contexts. Even worse, exploded summaries lead to frequent calls to the constraint solver, as each summary collection necessitates a call to the solver. For example, in the previous example, the summary π_1 is cloned and stored as two copies (π_1 and π_1') within the function *foo*. This leads to the collection of three new summaries π_4, π_5, and π_6, which eventually result in three calls to the constraint solver. When analyzing higher-level callers, the need for additional clones can cause significant performance issues.

Existing techniques to improve the performance have focused on achieving efficient summary path collection [17], [18] and the verification of summary conditions [6]. Specifically, Shi [17] and Tang [18] proposed a parallel algorithm to accelerate summary path collection, reducing the analysis time. Shi's recent work [6] introduced a unified representation of summary paths and summary conditions on the program dependence graph, enabling the direct verification of summary conditions on the PDG and eliminating the need for additional computation and storage of summary conditions.

Our Approach. To tackle this problem, we propose the first approach that identifies and eliminates "useless" summaries while also reducing the size of collected summaries from callee functions. Our key observation is that certain summaries in the callee functions do not contribute to any source-sink path (even when their summary condition is satisfiable(*sat*)) and, thus, can be safely ignored without compromising the analysis's precision. For example, in Fig. 1(c), the summary π_6 obtained from the function *foo* does not lead to any bugs since there are no dereference operations on the inlined $NULL_{19}'$ from the callee function qux. As a result, it is unnecessary to compute the summary π_6 and solve its path condition ϕ_{π_6}. Additionally, we can further reduce unnecessary computations by avoiding the cloning of π_1, not collecting π_2 (and solving the summary condition) to eliminate the redundant π_6. Our experiments (as shown in Table I under the "#Redun" column) indicate that approximately 20% of redundant summaries are computed, solved, and maintained throughout the analysis process on average.

The benefit of our approach is twofold. First, our approach efficiently reduces time and memory usage by eliminating unnecessary computations of summaries, which can become exponentially large as the analysis progresses. Second, the high-cost invocation of a constraint solver to verify "useless" summaries is subsequently avoided. These two unique advantages make our approach more practical for efficiently analyzing large-scale software systems. Our approach is orthogonal and can be used in conjunction with other approaches, such as enhancing the summary representation by employing advanced data structures, i.e., graph structures, for representing and resolving summary conditions [6], and effectively collecting summaries in a parallel [18] or pipelined manner [17].

Challenges and Solutions. The challenge lies in identifying redundant summaries precisely and efficiently without hurting the precision of the analysis. Specifically, determining the contribution of a summary often relies on information from upper-layer functions that have not been analyzed yet.

Our key insight is that a useful summary should *be a compo-*

nent of paths or path conditions associated with a source-sink path of interest. Specifically, the summary should be reachable from at least one pair of sources and sinks or derived from the path conditions of the source-sink paths. In Fig. 1, the usability of summary π_2 is decided by evaluating its reachability with the source-sink pairs $(NULL_{19}, printf(*a_9))$ and $(NULL_{19}, printf(*p_{13}))$. That is, we assess the reachability of the source $NULL_{19}$ and the sink $printf(*a_9)$ (or $printf(*p_{13})$) using the head f_{15} and tail f_{16} of π_2, which are parameter and return of function baz. They are represented as $(NULL_{19}, f_{15})$, $(f_{16}, printf(*a_9))$, and $(f_{16}, printf(*p_{13}))$. Without considering how π_2 will be used in caller *foo*, we can still decide that π_2 is not reachable from the two mentioned source-sink pairs. Consequently, π_2 is deemed redundant in the long run and can be promptly discarded. Using this insight, we have devised a principled, sound, and efficient contribution identification algorithm powered by a novel concept, namely *contribution abstraction*, to identify the contributing summaries. We give more details in Section III.

Results. We have implemented the contribution identification (CI) algorithm based on the state-of-the-art value flow analysis, Fusion [6], and evaluated it on 17 real-world programs. The evaluation results show our CI algorithm can significantly reduce the time and memory overhead of the Fusion by 45% and 27%, respectively. Furthermore, the CI algorithm can efficiently identify almost 80% of redundant summaries while only incurring a minimal additional overhead. In the largest project, the *mysqld* case, CI helps Fusion save 8107 seconds (2.25 hours) with only 17.31 seconds of overhead, resulting in a ratio of time savings to paid overhead (performance gain) of 468.48 $\times$. Overall, CI achieves a substantial average performance gain of 632.1 $\times$.

To sum up, this paper makes three main contributions:

- We identify and address the redundant summary deficiency in the prior value flow analysis.
- We design the contribution identification algorithm to identify redundant summaries efficiently and effectively.
- On average, the contribution identification algorithm can substantially enhance the performance of value flow analysis, reducing time consumption by 45% and minimizing memory utilization by 27%.

II. Background and Preliminary

This section introduces the background of path-sensitive value flow analysis and basic notations throughout the paper.

A. Background

We assume that the target program is in the static single assignment (SSA) form [19], where each variable has only one definition and multiple definitions are merged using a ϕ-assignment, following many existing works [3], [7], [9], [12], [13], [20]. All elements within an array or a union structure are considered to be aliases. In our implementation, we have utilized the existing methods to resolve points-to relations [3].

Program Dependence Graph (PDG). Given the program P, PDG is constructed to characterize how a value flows from one program statement to another through edges labeled with path constraints. We follow the previous works [3], [7], [9] to construct the program dependence graph, where the definition, the use of all variables, and operators are modeled as vertices.

Definition 1. A program dependence graph for a function is a directed graph denoted as $G = (V, O, E_d, E_c)$:

- V is a set of vertices, each of which is denoted by $v@s_i$, meaning the variable v is defined or used at a statement s_i. We write $v@s_i$ as v_i for short as the program is in the SSA form. The guard vertices are denoted as $V_g \subseteq V$.
- O is a set of operator vertices (binary $\oplus$ or unary $\otimes$), each of which represents a symbolic expression.
- $E_d \subseteq (V \cup O) \times (V \cup O)$ is a set of directed edges. $(v_i, v_j) \in E_d$ means that the value v_i flows to value v_j. Edges are labeled with guards from $V_g \cup true$, which represent the constraints that qualify the value flow.
- $E_c \subseteq V \times V_b$ is a set of control dependence edges.

Example 1. In Fig. 1(b), the two value flows from c_1 to $printf(*c_8)$ and from p_{11} to $printf(*p_{13})$ are qualified by the branch expressions at Line 7 and Line 12. Therefore, the two edges $(c_1, print(*c_8))$ and $(p_{11}, printf(*p_{13}))$ are labeled by guards φ_1 and φ_2. The expression of these branches can be derived by searching the PDG from guard vertices φ_1 and φ_2, which are $e_7 \neq NULL_7$ and $p_{11} \neq NULL_{12}$, respectively.

To check a given value-flow path π [3], the path condition ϕ_π incorporates not only the value flows represented by the edges but also the value flows related to the instantiation of the guard vertices labeled on those edges. Note that these two categories of value flows are not identical. The value flows related to the instantiation of guard vertices can be interconnected with the value flows of other paths. Thus, the path condition ϕ_π often relates more paths beyond just π.

Example 2. Recall Fig. 1(c), the summary $\pi_7 : c_1 \xrightarrow{\varphi_1} printf(*c_8)$ summaries the value flow from c_1 to c_8 with the guard φ_1. The value flow represented by the edge is encoded as $c_8 = c_1 \wedge \varphi_1$. When instantiating the constraint represented by the guard φ_1, which states that $e_7 \neq NULL_7$, the value flow of e_7 is tracked. This results in the identification of the summary path of π_6, represented as $NULL'_{19} \rightarrow m'_{20} \rightarrow \cdots \rightarrow e_6 \rightarrow e_7$. Consequently, the path condition of π_6 is applied in this context. Taking all of this into consideration, the path condition of π_7 can be expressed as $c_8 = c_1 \wedge e_7 \neq NULL_7 \wedge \phi_{\pi_6}$. Summary condition ϕ_{π_7} involves more complex value flow than summary path π_7 due to instantiating the constraint represented by the guard φ_1.

Given a value-flow path π on the program dependence graph G, $\pi[i]$ represents $i-$th vertex $v_i@s_i$ on the path. Specifically, we use $\pi[-1]$ to denote the tail element of π. Given sets V_1 and V_2, which are subsets of the vertices V in the PDG, we use $\Pi(V_1, V_2)$ to represent the set of value-flow paths from a vertex in V_1 to another vertex in V_2.

The path conditions of a set of value-flow paths $\Pi(V_1, V_2)$ are represented as $\Phi(V_1, V_2, V_g)$, The additional V_g denotes

Algorithm 1: Path-Sensitive Value Flow Analysis

```
1  Procedure pathSensitveAnalysis(P)
2      Build call graph CG and PDG of P;
3      S(V_src, V_sink) ← ∅;
4      foreach f ∈ CG do
5          S^f(V_fp, V_fr) ← ∅; S^f(V_fp, V_sink) ← ∅; S^f(V_src, V_fr);
6      foreach f ∈ CG in bottom-up order do
7          foreach c ∈ callees of f do
8              collectCloneSolveSmry(S^f(V_fp, V_fr), c);
9              collectCloneSolveSmry(S^f(V_fp, V_sink), c);
10             collectCloneSolveSmry(S^f(V_src, V_fr), c);
11             collectSrcSinkPath(S(V_src, V_sink), c);
12     ∀(π, φ) ∈ S(V_src, V_sink), report π as a bug if φ is sat;
```

the set of guard vertices that necessitate instantiation during the construction of the path conditions.

Bottom Up Value Flow Analysis. Given the PDG and bug-specific sources V_{src} and sinks V_{sink}, the path-sensitive value flow analysis is to collect $\Pi(V_{\text{src}}, V_{\text{sink}})$ and $\Phi(V_{\text{src}}, V_{\text{sink}}, V_{\text{g}})$. To determine the presence of bugs, each path π in $\Pi(V_{\text{src}}, V_{\text{sink}})$ is evaluated by checking its path condition ϕ_π using a constraint solver. If the path condition ϕ_π is determined to be *sat*, the path π is reported as a detected bug. To scale up the collection of $\Pi(V_{\text{src}}, V_{\text{sink}})$ and $\Phi(V_{\text{src}}, V_{\text{sink}}, V_{\text{g}})$, the existing path-sensitive approaches [3], [5], [6], [13], [17], [21] use a compositional manner that analyses each function on a call graph from the bottom. Note that existing bottom-up analyses [3], [5], [13], [17], [21] first compute the Strongly Connected Components (SCC) of the call graph to make it acyclic. Then, the path-sensitive methods compute symbolic summaries for each function. These summaries are subsequently instantiated in the callers' different contexts, allowing them to be reused to merge various source-sink paths and their corresponding path conditions, thereby eliminating the redundant re-analysis of each function.

However, the alternative top-down approaches [22]–[25] analyze functions in a call graph from top to bottom, producing summaries for specific program contexts that cannot be reused for all source-sink paths. Thus, they require analyzing the same function multiple times for different calling contexts, sacrificing path sensitivity. As a result, these approaches can only determine the reachability of a source-sink pair without providing the connecting paths and path conditions.

To construct complete source-sink paths, bottom-up approaches gather three types of summaries.

Definition 2 (Function Summary)**.** A summary for the function f is represented by a tuple $s = (\pi, \phi_\pi)$, where summary path π captures a value flow path after the callee functions' summaries are cloned. The summary condition ϕ_π encodes value flows of π and value flows instantiated from guard vertices that are labeled on π.

- Transfer summary $S(V_{\text{fp}}, V_{\text{fr}})$ summarizes value flow paths from the function's formal parameters V_{fp} to the function's formal return V_{fr}.
- Input summary $S(V_{\text{fp}}, V_{\text{sink}})$ summarizes value flow paths from the function's formal parameters to a sink within the function or its callee functions.
- Output summary $S(V_{\text{src}}, V_{\text{fr}})$ summarizes value flow paths from sources to the function's formal return. The sources are found within the function or callee functions.

Existing work [3], [5] shows that collecting these different categories of value flow paths is sound for bug detection. Corresponding to V_{fp} and V_{fr} that represent the sets of formal parameters and formal return, V_{ap} and V_{ar} represent the sets of actual parameters and actual return. We denote total summaries that are collected from function f as $S^f(V_h, V_t) = (\Pi^f, \Phi^f)$. V_h and V_t are the head vertices and tail vertices of the summary path that could be collected in Algorithm 1. Specifically, V_h are the head vertices come from V_{fp}, V_{ar} and V_{src}, and V_t are the tail vertices come from $V_{\text{ap}}, V_{\text{fr}}, V_{\text{sink}}$, and V_{g}.

Algorithm 1 presents the existing bottom-up summary collection [3], [5], [6], [13], [17], [21] of $S(V_{\text{src}}, V_{\text{sink}})$ for the given program P. In general, it is accomplished by two helper functions: `collectCloneSolveSmry` and `collectSrcSinkPath`. The first function, `collectCloneSolveSmry`, is responsible for collecting summaries by cloning the callee's summaries and then solving the summary condition, filtering out the *unsat* ones. The second function, `collectSrcSinkPath`, collects the source-sink paths that can be discovered after collecting the summaries in the current function. The overall Algorithm 1 begins by constructing the CG and PDG for the program P. It then initializes a global set $S(V_{\text{src}}, V_{\text{sink}})$ to maintain all the source-sink paths, as well as three summary sets for each function to maintain the three types of summaries. The algorithm proceeds to process each function in a bottom-up fashion, collecting three types of function summaries (in Lines 8, 9, and 10) and source-sink paths (in Line 11), assisted by the two helper functions. Finally, the algorithm reports bugs in Line 12 by solving the path condition after all functions have been analyzed.

The `collectCloneSolveSmry` helper collects summaries directly from the current function if the value flow path does not pass through a function call. Otherwise, it collects summaries by concatenating the inlined summaries from the called function c. For example, in Fig. 1, the summaries π_1 π_2, and π_3 are collected directly from their respective functions, while others are collected by concatenating callee summaries.

A single summary is collected in two steps: collecting the summary path Π and instantiating the summary condition Φ. Instantiating the summary condition often requires additional summary paths. As demonstrated in Example 2, instantiating a summary condition often involves additional value flow starting from the guard vertex labeled on the summary path. Once a condition is instantiated, it is solved by a constraint solver. If a summary's condition is *unsat*, the summary is discarded because an infeasible summary implies that the resulting source-sink path is also infeasible. The helper function puts the feasible summaries into three types of summary sets,

which would be used in the caller functions in the incoming analysis. At that time, the summaries are inlined, which helps the caller form long summaries and possible source-sink paths.

Example 3. Recall Fig. 1(c). Functions *qux*, *baz*, and *bar* are called by the function *foo*. Thus, *qux*, *baz*, and *bar* are analyzed first, while *foo* is analyzed later. In function *qux*, only the output summary s_1 is generated between the source $V_{\texttt{src}} = \{NULL_{19}\}$ and the formal return $V_{\texttt{fr}} = \{m_{20}\}$, and then ϕ_{π_1} is verified. Thus, *qux* has $S^{\texttt{qux}}(V_{\texttt{src}}, V_{\texttt{fr}}) = \{s_1 = (\pi_1, \phi_{\pi_1})\}$, with the other three sets being empty. In function *baz*, only the transfer summary s_2 is collected between the formal parameter $V_{\texttt{fp}} = \{f_{15}\}$ and the formal return $V_{\texttt{fr}} = \{f_{16}\}$, with the condition verified. Thus, *baz* has $S^{\texttt{baz}}(V_{\texttt{fp}}, V_{\texttt{fr}}) = \{s_2 = (\pi_2, \phi_{\pi_2})\}$, with the other three sets being empty. In function *bar*, the input summary s_3 is collected between the formal parameter $V_{\texttt{fp}} = \{p_{11}\}$ and the sink $V_{\texttt{sink}} = \{printf(*a_9)\}$, with condition verified. Thus, *bar* owns $S^{\texttt{bar}}(V_{\texttt{fp}}, V_{\texttt{sink}}) = \{s_3 = (\pi_3, \phi_{\pi_3})\}$, with the other three sets being empty. Since there is a conditional edge in π_3, its summary condition ϕ_{π_3} involves the instantiation of the constraint represented by the guard φ_2, which is $p_{11} \neq NULL_{11}$. This involves the value flow path of p_{11}, which coincides with its summary path.

Example 4. When analyzing the top layer function *foo*, the summaries are cloned from the bottom layer functions accordingly. When collecting the transfer summaries starting from formal parameter $V_{\texttt{fp}} = \{c_1\}$, which reaches a sink $printf(*c_8)$ but cannot reach any actual input in $V_{ap} = \{a_5, b_6\}$, no transfer summaries are gathered as a result.

When collecting the input summaries, the summary s_7 is generated with a summary path collected from the formal parameter $V_{\texttt{fp}} = \{c_1\}$ to the sink $V_{\texttt{sink}} = \{printf(*c_8)\}$, without cloning callee summaries. However, the summary condition ϕ_{π_7} necessitates instantiating the guard vertex φ_1, where $V_g = \{\varphi_1\}$. To this end, the value flow starting from φ_1 is tracked. When reaching the actual output e_6 of the function call to *bar* in line 6, the summary s_2 is cloned from the callee *bar*. Moving forward, when reaching the actual output b_3 of the function call to *qux* in line 3, the summary s_1 is cloned, thus forming the summary s_6. Finanly ϕ_{π_7} is verified.

When collecting the output summaries, because the $V_{\texttt{src}}$ in the current function is empty, the output summaries of the callee function are cloned, introducing additional sources. Thus, in line 2, the summary s_1 is inlined again, which is the output summary of the function *qux*. By combining the summary s_1, the value path from the output a_2 to a_5 is traced and involves the call to the function *bar* in line 5. Since a_5 is passed to *bar*, the input summary or transfer summary that starts at the corresponding formal parameter is inlined. In our case, the input summary s_3 of the function *bar* is inlined. After concatenating with the input summary, the path reaches a sink, forming a complete source-sink path, but does not reach any formal return. Thus, no output summary is collected.

The `collectSrcSinkPath` helper is similar to the `collectCloneSolveSmry` but tries to collect the source-sink paths in each iteration and does not solve the path conditions. In the example of Fig. 1, the helper collects no source-sink paths from bottom-layer functions, as no such paths are formed, but it collects two source-sink paths, s_4 and s_5, from the upper-layer function *foo*. Eventually, ϕ_{π_4} and ϕ_{π_5} are solved when reporting bugs in Line 12 of Algorithm 1.

III. Overview

In this section, we illustrate the problem using the motivating example and briefly describe our key idea.

A. Explosive Summary Problem

To detect the bug, the bottom-up collection of function summaries outlined in Algorithm 1 can collect and maintain a superset of the function summaries that are actually required. As highlighted in red in Fig. 1(c), the summaries s_2, s_6, and s_7 do not contribute to the two source-sink paths, s_4 and s_5, for detecting the NPD bug, i.e., either as components of these paths or their associated path conditions. Specifically, non-contributing summaries can arise in two scenarios in Algorithm 1. First, when collecting three types of summaries, over-summarization can occur. For instance, when analyzing the function baz, there is no prior knowledge of which the specific summary contributes, resulting in the conservative collection of all summaries within it. Consequently, the non-contributing summary s_2 is collected as a transfer summary. Second, non-contributing summaries can be induced through the cloning of callee summaries. For example, the summary s_6 is a non-contributing summary generated by cloning and concatenating with the callee summary s_2. Additionally, given that there are no source-sink paths within the function boo that rely on s_2, and considering that s_6 is deemed non-contributory, the cloning of s_2 from function baz should be avoided during the analysis of *foo*. As summaries are maintained and cloned into higher-level functions, the size of S^f can exponentially increase due to the explosion of paths. More importantly, collecting non-contributing summaries introduces expensive constraint-solving. To sum up, collecting and cloning only the contributing summaries can significantly improve scalability. The challenge lies in identifying redundant summaries precisely and efficiently without hurting the precision of the analysis.

B. Removing Redundant Summaries

We first propose the key idea of assessing whether a summary is contributing or not by solving two graph reachability problems between heads and tails of the summary s with distinct reaching targets (source-sink pairs or guards) without computing any summaries. Based on the graph reachability abstraction, we design the contribution identification algorithm, which identifies a set of necessary head and tail vertices for identifying the contributing summaries. Summaries that are collected outside these necessary head and tail vertices are identified as non-contributing automatically. Next, we explain

how our graph reachability abstraction and algorithmic design overcome the above challenges.

Contributing Summmary. Our key observation is that whether a summary is contributing hinges on two aspects: its path contribution, where it must be a component of source-sink paths, and its condition contribution, where it must be involved in the path conditions associated with a source-sink path. Thus, we establish the definition of a contributing summary generated from a function based on its contribution to the source-sink paths $S(V_{\text{src}}, V_{\text{sink}})$.

Definition 3 (Contributing Summary)**.** A summary $s \in S^f$ is considered a contributing summary for function f if it satisfies at least one of the following criteria: (1) Path Contribution: The summary path π is used to connect at least one source-sink path, i.e., $\pi \in \Pi(V_{\text{src}}, V_{\text{sink}})$. (2) Condition Contribution: The summary path π is used to instantiate at least one guard vertex labeled along the source-sink paths, i.e., $\phi_\pi \in \Phi(V_{\text{src}}, V_{\text{sink}}, V_{\text{g}})$.

We use Fig. 1 as an example. The path π_1 is reachable from both source-sink pairs $(NULL_{19}, printf(*p_{13}))$ and $(NULL_{19}, printf(*a_9))$. In addition, the path π_3 is reachable from the source-sink pair $(NULL_{19}, printf(*p_{13}))$. Therefore, they are identified as contributing summaries; indeed, they are components of two source-sink paths, π_4 and π_5. Comparatively, the paths π_2, π_6, and π_7 are neither reachable by any pair of sources and sinks nor reachable by the guard vertex φ_2 labeled on the source-sink path π_4. Thus, these paths π_2, π_6, and π_7 are identified as non-contributing summaries. Additionally, despite the reachability of the summary π_6 from the guard vertex φ_1 labeled on π_7, more specifically, the tail of π_6, denoted as (e_7), being reachable from φ_1, the summary π_7 does not contribute to any source-sink path or conditions. Consequently, it becomes redundant for NPD detection.

In summary, the summary contribution is identified by assessing two reachabilities between specific heads and tails of the summary path with various targets:

1) Path contribution: If the summary $s = (\pi, \phi_\pi)$ has the path contribution, a source-sink pair $((src, sink) \in V_{\text{src}} \times V_{\text{sink}})$ exists, where src can reach both $\pi[0]$ and $\pi[-1]$, and $sink$ is reached by both $\pi[0]$ and $\pi[-1]$.
2) Condition contribution: If the summary $s = (\pi, \phi_\pi)$ has the condition contribution criteria, there exists a guard vertex g $\in V_{\text{g}}$ from $\Phi(V_{\text{src}}, V_{\text{sink}}, V_{\text{g}})$ that are reached by $\pi[0]$ and $\pi[-1]$.

With the above abstraction, the assessment of contributing summaries is reduced to two graph reachability problems. In Section IV, we give a sound, efficient, and effective contribution identification algorithm by applying the abstraction.

IV. Contribution Identification

In this section, we first present three technical designs of our contribution identification algorithm. We then give the details of the identification algorithm that identifies the necessary vertices for path and condition contribution. Finally, we establish the soundness of our approach, analyze the complexity of algorithms, and discuss the advanced graph reachability with consideration of the calling context.

Preserving the precision of the analysis. To ensure this, instead of directly utilizing the abstractions to identify non-contributing summaries, the identification algorithm uses the abstractions to soundly identify all necessary head vertices V_h and tail vertices V_t that are reached by source-sink pairs (path contribution) as well as guard vertices V_g that are labeled on source-sink paths (condition contribution). This means that contributing summaries can only be collected within necessary vertices, which we denote as V^N. Therefore, summaries contributing to source-sink paths can be collected within V^N as in the traditional methods. In contrast, summaries collected outside V^N are considered as the non-contributing summaries. The soundness proof is given in Section IV-C.

Efficient and effective identification. The identification process relies on graph reachability. More precise graph reachability results in fewer necessary vertices V^N being identified, allowing for recognizing more non-contributing summaries starting outside of V^N. However, using advanced reachability algorithms increases the identification overhead. The complexity of more advanced reachability algorithms often outweighs the precision gains they can provide. To strike a balance, i.e., spending minimal overhead while significantly boosting the efficiency of path-sensitive analysis, we select the classic breadth-first search (`bfs`) algorithm for implementing our abstractions. More discussion about this is in Section IV-D.

Resolving implicit contribution. The source of the implicit contribution comes from the condition contribution of a summary. With the abstractions, resolving the implicit contribution is transferred to gather the necessary guard vertices that are labeled on source-sink paths. The key is that the necessary guard vertices could be obtained from edge sets that are reachable from the necessary heads and tails for path contribution.

The contribution identification algorithm is outlined in Algorithm 2. At a high level, it identifies necessary vertices V^N in two parts for path and condition contribution, respectively, through three stages. First, it identifies the first part of necessary vertices V^N for path contribution using the procedure `identifyPathContrib` in Algorithm 2. Next, using these necessary heads and tails for path contribution, we collect the necessary guard vertices with the procedure `collectNecGuards` in Algorithm 3. Lastly, based on the necessary guard vertices and heads and tails that are not identified for path contribution, we further identify the second part of necessary vertices for condition contribution using the procedure `identifyCondContrib` in Algorithm 3.

A. Path Contribution Identification

Procedure `identifyPathContrib` in Algorithm 2 utilizes `bfs` to explore the graph separately from both the source and sink vertices. Since a vertex v only needs to be reachable by at least one source-sink pair, the `bfs` initiated from the source (sink) vertices maintain a shared visiting set called `srcVisited` (`sinkVisited`). This ensures that each vertex is visited only once during the `bfs` from the sources

Algorithm 2: Contribution Identification

```
1  Procedure identifyContrib(G)
2      V^N ← ∅; V^cand ← ∅;
3      identifyPathContrib(G,V^N,V^cand);
4      identifyCondContrib(G,V^N,V^cand);
5      return V^N;
6  Procedure identifyPathContrib(G,V^N,V^cand)
7      srcVisited ← ∅; sinkVisited ← ∅;
8      foreach v ∈ V_src do
9          bfs(srcVisited, v, G, forward);
10     foreach v ∈ V_sink do
11         bfs(sinkVisited, v, G, backward);
12     V^N ← srcVisited ∩ sinkVisited ∩ (V_t ∪ V_h − V_g);
13     V^cand ← srcVisited ∪ sinkVisited −V^N;
```

Algorithm 3: Condition Contribution Identification

```
1  Procedure identifyCondContrib(G,V^N,V^cand)
2      V_g^nec ← ∅; fwdVisited ← ∅; bwdVisited ← ∅;
3      gatherNecGuards(V^N, V_g^nec);
4      foreach v ∈ V^cand do
5          bfs(fwdVisited, v, G, forward);
6      foreach v ∈ V_g^nec do
7          bfs(bwdVisited, v, G, backward);
8      V^N ← V^N ∪ (fwdVisited ∩ bwdVisited) ∩ (V_t ∪ V_h);
9  Procedure gatherNecGuards(V^N, V_g^nec)
10     fwdEdges ← ∅; bwdEdges ← ∅;
11     foreach v ∈ V^N do
12         bfsEdge(fwdEdges, v, G, forward);
13     foreach v ∈ V^N do
14         bfsEdge(bwdEdges, v, G, backward);
15     foreach (u, v) ∈ bwdEdges ∩ fwdEdges do
16         V_g^nec ← V_g^nec ∪ L_d(u, v);
```

(sinks). After completing all `bfs`, the necessary vertices for path contribution can be obtained by computing the intersection between the vertices visited both by `srcVisited` and `sinkVisited` and $V_t \cup V_h - V_g$ in Line 12. As the guard vertices could not have the path contribution, the necessary vertices only come from $V_t \cup V_h - V_g$. Vertices that are visited by sources and sinks but not identified as necessary vertices for path contribution are called candidates, denoted as V^{cand}. These vertices may have condition contribution, which are computed on Line 13 and passed to the procedure `identifyCondContrib` on Line 4.

B. Condition Contribution Identification

The necessary guard vertices are labeled on the edges of the source-sink paths. Thus, necessary guard vertices can be collected using the necessary vertices for path contribution.

The necessary guards are collected and maintained in V_g^{nec} through procedure `gatherNecGuards`, using the `bfsEdge` traversal to gather the visited edges. Two shared edge sets, `fwdEdges` and `bwdEdges`, are utilized to keep track of the visited edges for forward and backward `bfsEdge` starting from the vertices in V^N, respectively. These two sets ensure that each edge is visited only once during the forward and backward `bfsEdge`. For each edge (u, v) encountered in `fwdEdges`, if the reverse edge (v, u) is also found in `bwdEdges`, the label $L_d(u, v)$ is added to V_g^{nec}.

After collecting the necessary guard vertices, the algorithm then proceeds to collect another part of the necessary vertices for condition contribution. If a vertex has already been identified for path contribution, separately identifying its condition contribution is unnecessary. Thus, the procedure `identifyCondContrib` explores the vertices from the candidates in the forward direction and from each necessary guard vertex in the backward direction. During the `bfs` iterations, the shared sets `fwdVisited` and `bwdVisited` are utilized to keep track of the visited vertices in each direction, respectively. By examining the head and tail vertices present in both `fwdVisited` and `bwdVisited` sets, another part of the necessary vertices can be identified.

Example 5. Procedure `identifyPathContrib`'s output is $\{NULL_{19}, m_{20}, a_2, a_5, \text{printf}(*a_9), p_{11}, \text{printf}(*p_{13})\}$, and the necessary guard vertice is $\{\varphi_2\}$. The vertices that have condition contribution are $\{p_{11}, a_2, a_5, m_{20}, NULL_{19}, \varphi_2\}$ and some of them are included in the path contribution. Thus, the output of the procedure `identifyCondContrib` is $\{\varphi_2\}$. Also, the set V^N to assess the contributing summary is $\{NULL_{19}, m_{20}, \text{printf}(*a_9), p_{11}, \text{printf}(*p_{13}), a_2, a_5, \varphi_2\}$.

C. Soundness

We propose the following theorem to establish the soundness of abstracting contributions in identifying non-contributing summaries for function f based on V^N.

Theorem 1 (Soundness)**.** Given the set V^N identified, for any function $f \in P$, if a summary $s = (\pi, \phi)$ is collected and neither $\pi[0]$ nor $\pi[-1]$ appears in V^N, it must be a non-contributing summary for function f. Canceling the corresponding operations does not affect $S(V_{\text{src}}, V_{\text{sink}})$.

See the proof in a longer version of this paper [16].

Example 6. In Fig. 1, collecting summary path π_6 and verifying its condition ϕ_{π_6} are prevented because it is determined that cloning π_1' to concatenate with $b_3 \to b_6$ would result in a non-contributing summary. It is because b_3 is not included in V^N. Similarly, the collection and verification of summaries s_2 and s_7 are removed because the head vertices f_{15} and c_1 of these summaries are not present in V^N.

D. Summary and Discussion

Based on the identification of V^N, as outlined in Theorem 1, Algorithm 1 can be improved and revised as Algorithm 4. During the process of collecting the three types of summaries, the algorithm incorporates a filtering step for the head, tail, and guard vertices using V^N. The filtering step guarantees that summaries located outside of V^N are not collected or cloned, effectively eliminating non-contributing summaries.

Algorithm 4: New Path-Sensitive Value Flow Analysis

```
Procedure newPathSensitveAnalysis(P)
    Build call graph CG and PDG of P;
    V^N ← identifyContrib(PDG);
    /* The initialize part is the same as the Algorithm 1 */
    foreach f ∈ CG in bottom-up order do
        /* Pruning the redundancies. */
        V_fp ← V_fp ∩ V^N; V_ap ← V_ap ∩ V^N;
        V_fr ← V_fr ∩ V^N; V_ar ← V_ar ∩ V^N; V_g ← V_g ∩ V^N;
        /* The summary collection part, which is the same
           as the Alogirhtm 1. */
    ∀(π, φ) ∈ S(V_src, V_sink), report π as a bug if φ is sat;
```

Verification of these summaries by a constraint solver is thus avoided. Next, we discuss the important points of the algorithm, including its complexity, precision, and efficiency.

Complexity. The procedure `identifyPathContrib` involves searching the PDG G at most twice, as do each of the other procedures, since each edge is visited only once, either in the forward or backward traversal direction. As a result, the overall complexity is linear with respect to the size of PDG.

Precision and Efficiency. The soundness of V^N ensures the correct collection of contributing summaries. However, there may still be vertices within V^N that lead to non-contributing summaries. The precision of V^N can be increased by minimizing the number of such vertices, thereby identifying more non-contributing summaries. The reason for the incorrect collection of some vertices in V^N is that Algorithm 2 utilizes a traditional reachability algorithm, specifically the `bfs`, to address the reachability problems without differentiating the calling context under which the summary could be used. However, certain non-contributing summaries can only be identified under specific contexts, i.e., via context-free-language (CFL) reachability [23], [26]–[30]. Consequently, non-contributing summaries that are reachable under traditional reachability algorithms but not under context-sensitive reachability algorithms could be further identified. The precision of V^N, therefore, depends on the approach used to solve reachability. The CFL reachability algorithm takes calling context into consideration by respectively labeling function calls and returns with matched parentheses $[_k$ and $]_k$ at line k. Thus, a vertex i is context-sensitively reachable from a vertex j if the label string of the path does not contain any mismatched parentheses.

Example 7. In Fig. 1 (b), the following edges are labeled with parentheses to fulfill the requirements of the CFL reachability algorithm: $m_{20} \xrightarrow{]_2} a_2$, $m_{20} \xrightarrow{]_3} b_3$, $a_5 \xrightarrow{[_5} p_{11}$, $b_6 \xrightarrow{[_6} f_{15}$, and $f_{16} \xrightarrow{]_6} e_6$. By replacing `bfs` with a CFL reachability algorithm in Algorithm 2, we would produce $V^{\mathrm{N}} = \{NULL_{19}, m_{20}, \mathrm{printf}(*a_9), p_{11}, \mathrm{printf}(*p_{13}), a_2, a_5, \varphi_2\}$, which is the same as when using `bfs`. In this case, using the CFL reachability algorithm does not improve the precision.

In practice, the precision of V^N that can be improved by CFL reachability is limited. In our evaluation, approximately 80% of non-contributing summaries were successfully identified using `bfs`. However, in terms of efficiency, CFL reachability often encounters a cubic complexity barrier [27]–[30]. As demonstrated by our experiment in Fig. 2, using a CFL-based algorithm to collect V^N significantly decreases overall performance. Thus, we use `bfs`.

TABLE I: $|S^{all}|$ is total size of collected summaries. `#Redun` is the number of redundant summaries that occur, with the percentage relative to $|S^{all}|$ shown in parentheses. `#Identified` is the number of identified summaries by CI, with the percentage relative to `#Redun` shown in parentheses. `#Src` and `#Sink` represent the number of sources and sinks.

ID	Program	KLoC	$\lvert S^{all}\rvert$	#Redun	#Identified	#Src	#Sink
1	leela	21	47.2K	12.4K(**26%**)	9.3K(**75%**)	15	3.2K
2	nab	24	34.6K	5.2K(**15%**)	3.7K(**73%**)	88	3.6K
3	x264	96	94.2K	24K(**26%**)	19K(**79%**)	58	7.7K
4	wrf	130	51.1K	10.1K(**20%**)	8.4K(**83%**)	123	4.3K
5	omnetpp	134	405.6K	78.1K(**19%**)	66.1K(**85%**)	146	27.5K
6	povray	170	231.5K	45.8K(**20%**)	33.5K(**73%**)	24	14.5K
7	cactus	257	1.1M	307.7K(**29%**)	257.5K(**84%**)	38	49.8K
8	imagick	259	381.4K	30.3K(**8%**)	22.5K(**74%**)	154	12.2K
9	perlbmk	362	1.4M	217.8K(**15%**)	197.7K(**91%**)	232	40.9K
10	cam4	407	46.7K	10.1K(**22%**)	7.2K(**72%**)	53	3.4K
11	parest	427	3.6M	658.3K(**18%**)	555.9K(**84%**)	62	215.7K
12	xalanbmk	520	1.4M	294.4K(**21%**)	258.9K(**88%**)	23	77.7K
13	gcc	1304	3.5M	382.3K(**11%**)	287.4K(**75%**)	181	146.2K
14	blender	1577	3.2M	605.7K(**19%**)	505.6K(**83%**)	127	182.6K
15	libicu	537	1.1M	190.4K(**17%**)	165.8K(**87%**)	307	76.7K
16	ffmpeg	1346	2.1M	393.6K(**19%**)	244.7K(**62%**)	491	146.3K
17	mysqld	2030	3.8M	723.6K(**19%**)	519.2K(**72%**)	141	215.4K
avg			1.3M	234.7K(**19%**)	186K(**79%**)		

V. Evaluation

We have implemented the contribution identification (CI) algorithm in Algorithm 2 to identify the path and condition contribution in the state-of-the-art value flow analysis tool Fusion [6] for detecting NPD in C/C++ code. We denote Fusion with CI enabled as Light-Fusion, which serves as the performance-boosted client powered by our technique. We investigate the following three questions:

- (**RQ1**): How effective and efficient is CI in identifying the summary contribution?
- (**RQ2**): How much can CI boost the performance of existing value-flow analysis?
- (**RQ3**): Can CI enhance the path-sensitive analyzer's performance to be comparable with the top-down approach?

A. Experimental Setup

Baselines. Fusion collects the summaries in a parallel way, i.e., parallelizing functions located within the same level of the call graph, and uses the graph representation of summary conditions [6]. First, we compare Light-Fusion with Fusion. We did not compare to [17] [18] as their methodologies involve generating redundant summaries to enable parallelism, which contradicts the principles and objectives of our approach. In addition, we have developed a variant of Light-Fusion (denoted as CFL-Light-Fusion) that uses a more precise identification algorithm achieved through CFL reachability. Specifically, we adopted the open-source implementation of the state-of-the-art CFL reachability algorithm [31] provided by [32] and replaced the `bfs` used in Algorithm 2. Lastly, we compare the Light-Fusion to *PhASAR* [33], an open-sourced

TABLE II: The running time and memory usage of Fusion (F), Light-Fusion (L-F), and building of PDG are presented, along with CI's running time and performance gains (Gains).

ID	Memory(GB)			Time(s)				
	F	L-F	PDG	F	L-F	PDG	CI	Gains
1	18.42	13.13(**29%**)	1.1	185	61(**67%**)	6	0.11	1127.27
2	5.67	4.77(**16%**)	0.9	380	315(**17%**)	5	0.08	812.5
3	2.12	0.94(**56%**)	1.1	362	290(**20%**)	23	0.23	311.69
4	3.1	2.98(**4%**)	0.9	231	126(**45%**)	6	0.13	807.69
5	3.52	2.63(**25%**)	3.6	627	187(**70%**)	39	1.11	395.33
6	44.78	13.23(**70%**)	2.2	2087	869(**58%**)	35	0.67	1828.83
7	17.17	12.95(**25%**)	8.8	1435	524(**63%**)	423	3.34	272.67
8	79.98	74.89(**6%**)	7.9	3795	3485(**8%**)	118	2.23	139.14
9	102.7	77.9(**24%**)	19.7	8075	5641(**30%**)	312	8.67	280.9
10	2.52	2.08(**17%**)	11	409	343(**16%**)	65	0.14	471.43
11	18	7.32(**59%**)	32.6	10281	3450(**66%**)	497	14.22	480.48
12	32.06	18.83(**41%**)	10.6	5172	1689(**67%**)	114	4.91	709.95
13	215.11	178.11(**17%**)	35.7	19484	16569(**15%**)	515	15.75	185.07
14	336.37	234.88(**30%**)	21.9	10848	4454(**59%**)	536	12.11	527.91
15	131.92	83.77(**36%**)	12.4	8261	3660(**56%**)	124	4.01	1147.95
16	144.93	97.82(**33%**)	22.7	11018	4710(**57%**)	348	8.1	778.48
17	393.99	269.69(**32%**)	36.3	16157	8050(**50%**)	471	17.31	468.48
avg	91.31	66.65(**27%**)	13.5	5812.18	3201.35(**45%**)	213.94	5.48	632.1

implementation of top-down approaches [22]–[25]. To mitigate the impact of the execution environment, we ran experiments three times and calculated the average performance and performance gains.

Subjects. We have included all the large programs from the SPEC CPU@2017 benchmark [34] that consist of more than 10 KLoC. Additionally, we have selected three large programs, namely *libicu*, *ffmpeg*, and *mysqld*, which are widely-used software systems in their respective domains.

Sources and Sinks. For checking NPD, NULL pointers are selected as sources. The dereference operations are selected as sinks. Table I lists the evaluation subjects with statistics of sources and sinks.

Environment. All experiments were run on a server with eighty "Intel Xeon CPU E5-2698 v4@2.20GHz" processors and 512 GB of memory running Ubuntu-18.04. Each program is analyzed with a limit of 12 hours and 256 GB of memory. Fifteen threads are used to analyze the functions in the same layer when running both Fusion and Light-Fusion. The solver used to verify constraints is *Z3* [15].

B. RQ1: Effectiveness and Efficiency

To study the effectiveness of CI, we run two experiments for each benchmark. In the first experiment, we execute Fusion to produce the results $S(V_{\text{src}}, V_{\text{sink}})$ and collect all potential summaries that could be generated during analysis, denoted as S^{all}. Note that S^{all} contains the non-contributing summaries, which is a superset of $S(V_{\text{src}}, V_{\text{sink}})$. The summary size of each benchmark is recorded in the $|S^{all}|$ column of Table I. Second, we count non-contributing summaries from S^{all} based on the definition of contributing summaries (Definition 3). The relative number of non-contributing summaries is reported in the "#Redun" column of Table I.

In the second experiment, we run Light-Fusion using the same configuration as Fusion to produce $S(V_{\text{src}}, V_{\text{sink}})$. Light-Fusion, however, incorporates an additional identification algorithm, CI, to filter out non-contributing summaries before the summary collection and cloning. We count the number of identified non-contributing summaries and then compare it with the total number of non-contributing summaries in the first experiment to determine the identification ratio. The findings are listed in the "#Identified" column of Table I.

Throughout both experiments, the PDG of each benchmark is pre-built once and persisted to disk. Both Fusion and Light-Fusion read the same PDG as input when analyzing a benchmark. We monitor the execution time, the memory consumption, and the analysis results, $S(V_{\text{src}}, V_{\text{sink}})$. The performance data for the two runnings and the building of PDG are listed in Table II. In the second set, we specifically record the time consumed by CI, as the memory usage generally stays low, below 200 MB in 15 benchmarks, with the exceptions of *blender* at 265 MB and *mysqld* at 252 MB. The running time of CI is listed in the "CI" column of Table II.

Soundness. We compare the analysis results $S(V_{\text{src}}, V_{\text{sink}})$ across both sets and the unsafe sinks reported by Fusion and Light-Fusion in Section V-D. They remain the same, demonstrating the preservation of the precision of our approach.

Effectiveness. While using normal graph reachability (`bfs`) to collect necessary vertices V^N, CI soundly approximates the summary contribution without considering context sensitivity. Thus, some redundant summaries could be incorrectly classified as contributing ones. Column "#Identified" in Table I lists the number of redundant summaries that can be identified and the percentage relative to redundant summaries that occur for each benchmark. We observe that CI can precisely identify redundant summaries, reaching the high ratio from 62% (in *ffmpeg*) to 91% (in *perlbench*). On average, CI correctly identifies 79% of redundant summaries. One could reject the bogus V^N by reaching context sensitivity. However, handling context sensitivity may be more costly. We examine this aspect in Section V-C. The high rate of identification achieved by CI in most projects shows that there is limited room for improvement with a context-sensitive identification algorithm.

Efficiency. Table II presents the running performance of Fusion, Light-Fusion, CI, and the performance gains when applying CI. Light-Fusion, using our CI to prune redundant summaries, includes CI's running time in its performance metrics. For all benchmarks, CI's running time for collecting V^N remains below a minute. Specifically, using CI significantly improves the running time and memory of Light-Fusion, as evidenced by the data in the "Time" and "Memory" columns of the table. Without computing and maintaining the redundant summaries captured by CI, the average running time of Fusion is reduced from 5812.18 seconds to 3201.35 seconds, saving over 40 minutes (or 2,610.83 seconds). Also, the average memory usage for Fusion drops from 44.26 GB to 33.88 GB.

Breakdown. To investigate how much resource is spent collecting redundant summaries and how much resource is spent on calling the constraint solver to verify the conditions of redundant summaries, we break down the reduced performance in Table II. This is obtained by computing the difference in running time and memory between Fusion and Light-Fusion, as shown in Table II. Additionally, we present the number of solver calls avoided in Table III. Once the summary path is collected, the constraint solver is called; thus, the avoided calls of the constraint solver equal the number of

TABLE III: `#Solver` is the number of saved solver calls, which is equal to the number of identified redundant summaries (`#Identified` column in Table I). The total (`T`) reduction in performance, performance of calling the solver (`S`), with the percentage of each with respect to the total reduction in performance shown in parentheses.

ID	#Solver	Time(s)		Memory(GB)	
		T	S	T	S
1	9.3K	124	110(**89%**)	5.29	1.3(**25%**)
2	3.7K	65	59(**91%**)	0.9	0.1(**11%**)
3	19.0K	72	66(**92%**)	1.18	0.26(**22%**)
4	8.4K	105	95(**90%**)	0.12	0.01(**8%**)
5	66.2K	440	403(**92%**)	0.89	0.18(**20%**)
7	257.5K	911	840(**92%**)	4.22	1(**24%**)
8	22.5K	310	280(**90%**)	5.09	1.4(**28%**)
9	197.7K	2434	2222(**91%**)	24.8	5.6(**23%**)
10	7.2K	66	59(**89%**)	0.44	0.1(**23%**)
11	555.9K	6831	6403(**94%**)	10.68	3.3(**31%**)
12	258.9K	3483	3320(**95%**)	13.23	4.3(**33%**)
13	287.4K	2915	2817(**97%**)	37	8.9(**24%**)
14	505.6K	6394	6188(**97%**)	101.49	37.1(**37%**)
15	165.8K	4601	4403(**96%**)	48.15	12(**25%**)
16	244.7K	6308	6099(**97%**)	47.11	8.8(**19%**)
17	519.2K	8107	7803(**96%**)	124.3	26.8(**22%**)
avg	268.7K	2610	2487(**95%**)	24.7	6.66(**27%**)

redundant summaries identified, as shown in the data in the columns "#Identified" in Table I and "#Solver" in Table III. For the total reduced running time, almost 90% is spent on verifying the conditions of redundant summaries across the benchmarks. As the size of the program grows, the complexity of the summary conditions tends to increase as well. This results in longer summary paths and more time required to solve these conditions. Therefore, the proportion of time spent on the solver increases from 89% to 97%. On the other hand, for the total reduced memory, almost 73% is consumed by storing the collected summaries.

Performance Gains. It is noted that the running time of CI does not hurt the performance of Light-Fusion. The "Gains" column in the table illustrates the performance gains achieved by using CI, which is the ratio of the time saved by Light-Fusion to the overhead incurred by CI. In the *povray* (ID 6) case, CI achieves the highest performance gain of 1828.83 × by reducing 1218 seconds with only 0.67 seconds of overhead. In the *mysqld* (ID 17) case, CI reduces the most time, saving 8107 seconds (2.25 hours), which is nearly half of the original Fusion running time, with only 17.31 seconds of overhead. On average, CI achieves a performance gain of 632.1×.

C. RQ2: Performance Boosting

Fusion vs. Light-Fusion. The performance comparison between Fusion and Light-Fusion is listed in Table II. The percentage numbers in parentheses represent the extra performance requirements of Fusion against Light-Fusion. On average, both the time and memory could be reduced by 45% and by 27% by using CI. The time reduction is more significant than the memory reduction since analyzers allocate considerable CPU resources to summary collection and solving path conditions, which can be NP-hard [4], [35]. Thus, pruning redundant summaries can save significant time by conserving CPU resources. The memory reduction percentages are constrained by the number of redundant summaries in the benchmarks, as indicated by the "#Redun" column in Table I, which shows an average redundancy ratio of 19% closely corresponding to the average memory reduction percentages.

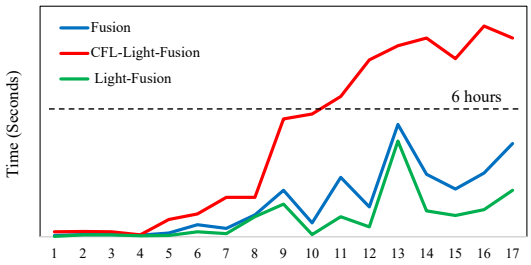


Fig. 2: Light-Fusion vs. its variants

Light-Fusion vs. CFL-Light-Fusion. To explore the impact of enhancing the CI precision, such as by using the CFL reachability [27]–[30] to reach context sensitivity and identify more redundant summaries compared to *bfs*, we replaced *bfs* in CI with the state-of-the-art CFL reachability algorithm [31]. The modified algorithm was executed as the CFL-Light-Fusion instance compared with Fusion and Light-Fusion. We monitored the running time on all three instances.

The results presented in Fig. 2 demonstrated that CFL-Light-Fusion did not yield performance improvements; instead, it significantly slowed down the process, particularly as the program size increased. Notably, CFL-Light-Fusion failed to analyze benchmarks beyond *cam4* (ID 10) within a six-hour timeframe. This decrease in performance can be attributed mainly to the cubic complexity of CFL-based approaches [27]–[30], which introduce substantial overhead that outweighs the precision benefits. As a result, trading efficiency for precision becomes impractical.

D. RQ3: Comparing to Top-down Approaches

To evaluate the performance improvements of the path-sensitive analyzer compared to top-down approaches [22]–[25], [36], we conducted a comparative analysis between Light-Fusion and *PhASAR* [33]. Using the latest release [37] of *PhASAR* at the time of writing, we configured it to analyze the same source-sink pairs as Light-Fusion to ensure a fair comparison.

Performance. The results shown in Fig. 3 show the running time and memory usage of Fusion, Light-Fusion, and *PhASAR*, which are represented by blue, green, and red bars, respectively. In general, both Fusion and Light-Fusion require more resources compared to *PhASAR*, as indicated by the taller blue and green bars compared to the red bars. However, the green bars (representing Light-Fusion) are closer in height to the red bars (*PhASAR*) than the blue bars (representing Fusion). In some benchmarks, the green bars are even lower than the red bars. For example, the running time of Light-Fusion for *cactus* (ID 7) and *xalanbmk* (ID 12), as well as the memory usage for *x264* (ID 3), are lower than those of *PhASAR*.

Next, we quantify the amount of running time and memory among Fusion, Light-Fusion, and *PhASAR*. The results show

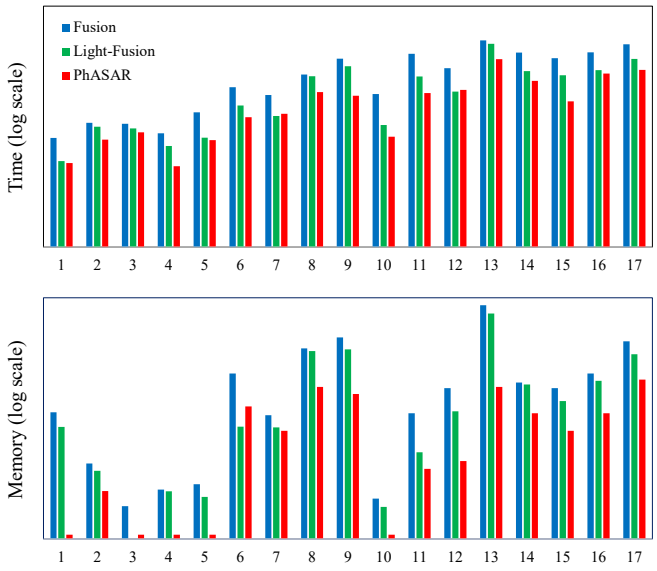


Fig. 3: Performance: Fusion vs. Light-Fusion vs. *PhASAR*.

that Fusion requires 3.35× the time and 9.75× the memory consumed by *PhASAR* on average. However, Light-Fusion can reduce the requirements to just 1.4× the time and 6.95× the memory. Additionally, Light-Fusion has performance comparable with *PhASAR* on benchmarks such as *leea* (ID 1), x264 (ID 3), *omnetpp* (ID 5), *cactus* (ID 7), *xalanbmk* (ID 12), and *ffmpeg* (ID 16), requiring only an additional 0.5× of *PhASAR*'s execution time. However, Fusion could only complete *x264* (ID 3) within the same threshold.

Analysis Results. We record the unsafe sinks reported by Fusion, Light-Fusion, and *PhASAR* for each benchmark. A sink is considered unsafe if there is a feasible path from a source to the sink. Note that top-down approaches, such as *PhASAR*, usually sacrifice path sensitivity for better scalability. As a result, they may misclassify safe sinks as unsafe ones.

First, the results show that the unsafe sinks reported by Light-Fusion are the same as those reported by Fusion, demonstrating our approach's precision-preserving nature. Next, we examine the unsafe sinks reported by Light-Fusion and *PhASAR*. Light-Fusion identifies 90% of the sinks, which are reported as unsafe by *PhASAR*, as actually being safe, indicating a high false-positive rate for *PhASAR*.

Additionally, nearly 92% of unsafe sinks identified by Light-Fusion are also identified by *PhASAR*. However, *PhASAR* fails to report some unsafe sinks identified by Light-Fusion due to not modeling the value flow of library functions after manual verification. We provide the complete comparison data in a longer version of this paper [16].

In conclusion, our method incorporates path-sensitive analyses without compromising precision and still achieves good performance. This aligns with common industrial requirements [38], [39] of maintaining both high performance and low false positive rates.

VI. Related Work

Compositional Analysis. Many compositional analyses aim to improve efficiency by reusing information within a procedure as summaries [3], [12], [17], [22], [25], [39], [40], which include top-down and bottom-up summaries. Most program analyses prefer a bottom-up approach [3], [12], [17], [39], [40]. In these approaches, a function's effect is represented using bottom-up summaries, and the summary of a callee is inlined into the summary of its caller. This avoids redundant analysis of individual functions. However, when collecting the summary of a callee, it is challenging to determine whether the summary will be useful to callers since the callers are typically analyzed afterward. Our work employs a lightweight analysis to pre-compute a set of vertices, which allows us to determine whether a summary should be computed and reduces unnecessary computations.

Value Flow Analysis. Cherem et al. [1] utilized value flow analysis to detect software bugs such as memory leaks. Subsequently, several works have aimed to refine the recall and precision of this analysis [3], [5]–[7], [9]. However, most of these analyses are not inter-procedurally path-sensitive, with the exceptions of Pinpoint [3] and Fusion [6]. Fusion is an optimization of the performance issues identified in Pinpoint, which were caused by the explosion of summaries and paths in inter-procedural analysis. Fusion addresses this problem by eliminating the storage of path conditions. However, it is worth noting that even with Fusion, there are still redundant summaries and paths being computed in function summaries, which cannot be fully optimized. While this work helps Fusion overcome redundancy issues related to function summaries.

VII. Conclusion

We present the contribution identification algorithm, which addresses the redundant summary deficiency in the prior value flow analysis. It identifies redundant summaries efficiently and effectively without compromising soundness or efficiency. Furthermore, it results in an average decrease in the time and memory overhead of state-of-the-art path-sensitive value flow analysis by 45% and 27%, respectively. In the end, it enables path-sensitive analyses without compromising precision and still achieves good performance, making it comparable to path-insensitive analyses.

VIII. Acknowledgment

We thank the anonymous reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. This work is funded by research donations from Huawei, TCL, and Tencent. Yuandao Cai is the corresponding author.

References

[1] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM, 2007, pp. 480–491.

[2] B. Livshits and M. S. Lam, "Tracking pointers with path and context sensitivity for bug detection in c programs," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '03. ACM, 2003, pp. 317–326.

[3] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '18. ACM, 2018, pp. 693–706.

[4] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: Scalable path-sensitive memory leak detection for millions of lines of code," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE, 2019, pp. 72–82.

[5] Q. Shi, R. Wu, G. Fan, and C. Zhang, "Conquering the extensional scalability problem for value-flow analysis frameworks," in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020, pp. 812–823.

[6] Q. Shi, P. Yao, R. Wu, and C. Zhang, "Path-sensitive sparse analysis without path conditions," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 930–943.

[7] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, 2014.

[8] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the 14th International Symposium on Code Generation and Optimization*, ser. CGO '16. IEEE, 2016, pp. 160–170.

[9] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC '16. ACM, 2016, pp. 265–266.

[10] C. Wang, W. Wang, P. Yao, Q. Shi, J. Zhou, X. Xiao, and C. Zhang, "Anchor: Fast and precise value-flow analysis for containers via memory orientation," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–39, 2023.

[11] C. Wang, P. Yao, W. Tang, Q. Shi, and C. Zhang, "Complexity-guided container replacement synthesis," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–31, 2022.

[12] D. Babic and A. J. Hu, "Calysto: Scalable and precise extended static checking," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. IEEE, 2008, pp. 211–220.

[13] Y. Xie and A. Aiken, "Scalable error detection using boolean satisfiability," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. ACM, 2005, pp. 351–363.

[14] R. Wu, Y. He, J. Huang, C. Wang, W. Tang, Q. Shi, X. Xiao, and C. Zhang, "Libalchemy: A two-layer persistent summary design for taming third-party libraries in static bug-finding systems," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[15] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '08. Springer, 2008, pp. 337–340.

[16] Y. Wang, Y. Cai, and C. Zhang, "Boosting path-sensitive value flow analysis via removal of redundant summaries," 2025. [Online]. Available: https://arxiv.org/abs/2502.04952

[17] Q. Shi and C. Zhang, "Pipelining bottom-up data flow analysis," in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020, pp. 835–847.

[18] W. Tang, D. Dong, S. Li, C. Wang, P. Yao, J. Zhou, and C. Zhang, "Octopus: Scaling value-flow analysis via parallel collection of realizable path conditions," *ACM Transactions on Software Engineering and Methodology*, 2023.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[20] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '08. USENIX, 2008, pp. 209–224.

[21] P. Cousot and R. Cousot, "Modular static program analysis," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. Springer, 2002, pp. 159–179.

[22] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. ACM, 1995, pp. 49–61.

[23] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up slicing," in *Proceedings of the 2nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. FSE '94. ACM, 1994, pp. 11–20.

[24] B. R. Murphy and M. S. Lam, "Program analysis with partial transfer functions," in *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, 1999, pp. 94–103.

[25] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science*, vol. 167, no. 1, pp. 131–170, 1996.

[26] T. Reps, "Shape analysis as a generalized path problem," in *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '95. ACM, 1995, pp. 1–11.

[27] J. Kodumal and A. Aiken, "The set constraint/cfl reachability connection in practice," in *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '04. ACM, 2004, pp. 207–218.

[28] S. Chaudhuri, "Subcubic algorithms for recursive state machines," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. ACM, 2008, pp. 159–169.

[29] M. Yannakakis, "Graph-theoretic methods in database theory," in *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '90. ACM, 1990, pp. 230–242.

[30] D. Melski and T. Reps, "Interconvertibility of a class of set constraints and context-free-language reachability," *Theoretical Computer Science*, vol. 248, no. 1-2, pp. 29–98, 2000.

[31] Y. Lei, Y. Sui, S. Ding, and Q. Zhang, "Taming transitive redundancy for context-free language reachability," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1556–1582, 2022.

[32] "POCR: Light-weight CFL-reachability solver," https://github.com/kisslune/POCR, 2022 Oct.

[33] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An interprocedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.

[34] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[35] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, 2023, pp. 143–152.

[36] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "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*, ser. PLDI '14. ACM, 2014, pp. 259–269.

[37] "Phasar Github Release 03/24/2024," https://github.com/secure-software-engineering/phasar/releases/tag/v2403, 2024 March 24.

[38] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[39] S. McPeak, C.-H. Gros, and M. K. Ramanathan, "Scalable and incremental software bug detection," in *Proceedings of the 14th European Software Engineering Conference Held Jointly with the 21st ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '13. ACM, 2013, pp. 554–564.

[40] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, *The Saturn Program Analysis System*. Stanford University, 2006.