

# LIBALCHEMY: A Two-Layer Persistent Summary Design for Taming Third-Party Libraries in Static Bug-Finding Systems

Rongxin Wu

Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University  
Xiamen, China  
wurongxin@xmu.edu.cn

Yuxuan He

Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University  
Xiamen, China  
yuxuanhe@stu.xmu.edu.cn

Jiafeng Huang

Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University  
Xiamen, China  
thghjf@stu.xmu.edu.cn

Chengpeng Wang

The Hong Kong University of Science and Technology  
Hong Kong, China  
cwangch@cse.ust.hk

Wensheng Tang

The Hong Kong University of Science and Technology  
Hong Kong, China  
wtangae@cse.ust.hk

Qingkai Shi

State Key Laboratory for Novel Software Technology, Nanjing University  
Nanjing, China  
qingkaishi@nju.edu.cn

Xiao Xiao

Person  
Hang Zhou, China  
frogxx@gmail.com

Charles Zhang

The Hong Kong University of Science and Technology  
Hong Kong, China  
charlesz@cse.ust.hk

## ABSTRACT

Despite the benefits of using third-party libraries (TPLs), the misuse of TPL functions raises quality and security concerns. Using traditional static analysis to detect bugs caused by TPL function is non-trivial. One promising solution would be to automatically generate and persist the summaries of TPL functions offline and then reuse these summaries in compositional static analysis online. However, when dealing with millions of lines of TPL code, the summaries designed by existing studies suffer from an unresolved paradox. That is, a highly precise form of summary leads to an unaffordable space and time overhead, while an imprecise one seriously hurts its precision or recall.

To address the paradox, we propose a novel two-layer summary design. The first layer utilizes a line-sized program representation known as the program dependence graph to compactly encode path conditions, while the second layer encodes bug-type-specific properties. We implemented our idea as a tool called LIBALCHEMY and evaluated it on fifteen mature and extensively checked open-source projects. Experimental results show that LIBALCHEMY can check over ten million lines of code within ten hours. LIBALCHEMY has detected 55 true bugs with a high precision of 90.16%, eleven of which have been assigned CVE IDs. Compared to whole-program

analysis and the conventional design of path-sensitively precise summaries, LIBALCHEMY achieves an 18.56× and 12.77× speedup and saves 91.49% and 90.51% of memory usage, respectively.

## KEYWORDS

static bug-finding, function summary, third-party library

### ACM Reference Format:

Rongxin Wu, Yuxuan He, Jiafeng Huang, Chengpeng Wang, Wensheng Tang, Qingkai Shi, Xiao Xiao, and Charles Zhang. 2024. LIBALCHEMY: A Two-Layer Persistent Summary Design for Taming Third-Party Libraries in Static Bug-Finding Systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3597503.3639132>

## 1 INTRODUCTION

Third-party libraries (TPLs) enable developers to integrate pretested and reusable software components developed by other vendors, thus saving development costs. Despite the benefits, the usage of TPLs raises concerns about software quality and has become one of the most severe security threats (e.g., OWASP Top 10 in 2013 [1], 2017 [2], and 2021 [3]). Thus, the security issues of TPLs have attracted considerable attention in both industry and academia.

Typically, the usage of TPLs that are known to be vulnerable (e.g., published in the CVE database) receives more attention from developers due to the significant impact of standard-based vulnerability management. Various well-established techniques, such as software composition analysis [4–7] and code clone detection [8–11], are applicable to handle them. Meanwhile, developers may easily miss security bugs caused by the misuse of non-vulnerable TPL functions due to the need for a deeper understanding of the implementation details of TPL functions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ICSE '24, April 14–20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0217-4/24/04...\$15.00

<https://doi.org/10.1145/3597503.3639132>

For example, Figure 1 shows a null pointer dereference (NPD) bug caused by the misuse of TPL functions, simplified from a real-world bug [12]. In the application project, `opusfile-0.12`, the function `ogg_sync_buffer` in the library `libogg-1.3.4` is called at Line 4 and will return a null pointer under certain conditions. However, the application function `op_open1` does not check this return value and directly uses it as a parameter of the function `memcpy` in the library `glibc-2.31` (Figure 1 (b)) at Line 5, eventually leads to a null pointer dereference at Line 35. It is non-trivial for developers to find such bugs because it requires examining the implementation of the library code. Existing static TPL function misuse detection techniques [13–15], which extract the typical usage patterns of TPL functions from several samples and detect anomalous TPL function usages by comparing them with the typical patterns, would help to find such bugs. However, they would fail to distinguish the buggy and benign TPL function usages because they treat TPL functions as black boxes. Take the same code snippet in Figure 1(a) as an example. If the version of function `memcpy` invoked at Line 5 has evolved to the one in Figure 1 (c), it will not result in a null pointer dereference due to the null pointer guard at Line 40. However, the existing TPL function misuse detection techniques will still consider it as buggy. This indicates that it is essential to make static bug-finding techniques to understand the path conditions of TPL function usage (i.e., path sensitivity).

**Problems of State-of-the-Art Techniques.** Research on applying static analysis to find bugs in software systems has been continuously developed for decades [16–35]. Despite this tremendous research progress, we observe the difficulty of applying static bug detection techniques in the presence of numerous TPLs.

One possible solution is to perform whole-program analysis in both application and library code [23, 26, 27]. However, as shown by existing studies [16, 36], the size of dependency libraries can significantly dwarf the application code and thus raises the grand challenge for the scalability of analysis. For example, `vim`, a widely-used text editor with only 469K lines of code, transitively depends on 41 libraries with 4.25 million lines of code. When detecting bugs in such a scale using the state-of-the-art path-sensitive static analyzer, it exhausts 460G of memory after running for only 25 minutes in our evaluation (See § 6).

The second possible solution is applying aggressive or conservative approximations for the missing library behaviors [37]. For example, when performing taint analysis, aggressive approximation assumes that a tainted argument of a callee function will always flow into the return value of this callee function. The other way around, the conservative approximation takes the opposite assumption. If the approximation is inconsistent with the missing library behaviors, it will inevitably lead to false positives or negatives.

The third possible solution is to provide precise summaries of the missing library behaviors. Some industrial tools resort to hand-written summaries for commonly-used libraries (e.g., the GNU C library) [37]. At the same time, some research studies explore semi-automated techniques to reduce false positives by inferring specifications from the oracle or hand-written summaries for a set of TPL functions [18, 20, 38, 39]. Although the approaches mentioned above would be precise, constructing a human oracle is

time-consuming, labor-intensive, and error-prone when the number of TPL functions is large. An ideal solution is to automatically generate precise summaries of all TPL functions offline and reuse these summaries when static bug-finding systems working on application code encounter these TPL functions. Note that the concept of summaries is not novel and has been widely used in compositional or incremental static analysis. However, the design of summaries by the existing studies cannot fulfill the requirements of detecting bugs caused by TPL function misuse for the following three reasons. First, as pointed out by a prior study [16], summaries in a significant proportion of existing studies are designed and reused in the context of the same analysis process [24, 31, 33, 40–42], and there is no easy way to serialize and reuse them in multiple analysis runs. Second, another category of summaries is designed for caching results across runs of an analysis [43–47] but limits its capability to reuse results from the previous analysis run on the same program. Third, some summaries can be reused in multiple runs of analysis and across different program modules [16, 34, 35, 48, 49]. However, they suffer from the paradox between precision, the cost of time, and space overhead. Specifically, prior studies have demonstrated that path-sensitive precision is necessary for detecting many security bugs like memory-related errors [16, 34, 35], e.g., null pointer dereference and memory leak. Unfortunately, current path-sensitive summaries [48, 49] typically take the SMT constraints to represent path conditions, while this induces a high cost of time and space overhead. In our experiment, for instance, we observed that it takes over 16 hours and more than 450GB to persist the summaries for the libraries with over 1 MLoC.

**Our Solution.** We base our idea to resolve the paradox on the observation that, in an actual program, the explosive size of path conditions consumes enormous amounts of memory when performing path-sensitive analysis [50], and this implies a high space overhead if using the heavyweight form of path conditions, i.e., SMT constraints, let alone the high time overhead. Thus, in this work, we propose a two-layer summary design to mitigate the high cost while preserving the high precision.

In the first layer, we use a program dependence graph as the summary, a linear-sized program representation that essentially encodes the same program information as path conditions. The first-layer summary design can guarantee the low cost of persistence and the capability of faithfully recovering path conditions.

In the second layer, to avoid repetitively verifying bug-type-specific properties in TPL functions that have been analyzed, we use data-flow paths on PDG as the bug-type-specific summaries and design an efficient algorithm to recover path conditions. To mitigate the high cost of persisting the explosive size of candidate paths in TPL functions, we only collect the data-flow paths with specific start and end points for persistence. These persistent paths capture a function’s reachability relation among the parameter vertices, return vertices, and the bug-type-specific “source” and “sink” vertices at the intraprocedural level. Then, we propose an approach to recovering the interprocedural path conditions by on-demand stitching these intraprocedural data-flow paths. The second-layer summary design enables efficient path condition recovery with low space overhead.



For example, Figure 1 (b) and Figure 1 (c) demonstrate an unsafe and safe version of the library function `memcpy`, respectively. The only difference is that the safe version in Figure 1 (c) includes a branch condition at Line 40 that checks if the parameters `dst` and `src` could be null pointers, thereby avoiding the potential NPD at Line 42. A path-insensitive approach ignores such conditions and will generate the same summary for both versions, i.e., an NPD may happen at Lines 35 and 42 in Figure 1 (b) and Figure 1 (c), respectively. Clearly, such a summary is not precise enough for the safe version. Using this imprecise summary, a program that calls this safe version will be regarded as calling the original unsafe version. For example, even though we call the safe version at Line 5 in Figure 1 (a), due to the imprecise summary, a static analyzer will report a possible NPD at Line 5, which, however, is a false positive due to the branch condition at Line 40.

**Problems of a Naïve Approach.** Generally, a path-sensitive static analyzer is expected to analyze each path in a program and use path conditions to infer its path feasibility. Hence, a naïve approach to persisting summaries for TPL functions is to persist all possible bug-inducing paths and their path conditions. For example, Table 1 lists all paths in the TPL function `ogg_sync_buffer` that may return a null pointer to the client function and their path conditions. When analyzing the client in Figure 1 (a), we can reason whether the pointer `o` is null and check if it will lead to an NPD at Line 5. For instance, the first path in Table 1 means that the null pointer at Line 9 is returned to the caller function with the path condition shown in the second and third columns. When finding bugs in a caller function, we can check if this path condition is satisfiable under other conditions from the caller function.

While the naïve approach ensures the precision of path sensitivity, it may lead to critical performance issues due to two problems. First, a program may have an exponential number of paths (known as the path explosion problem). Enumerating and persisting all these paths could be in exponential complexity and, thus, impractical. Second, many of these paths share path conditions. Repetitively persisting these shared conditions is unnecessary and leads to a large space overhead. For example, in Table 1, where we use  $v_i$  to denote the value of the variable  $v$  at Line  $i$ , the first path and the second share the formula  $sz_{11} \leq MAX$  in their path conditions.

**The Approach in a Nutshell.** The basic idea to address the two problems mentioned above is to find a highly-compressed data structure that must satisfy three requirements to persist paths and path conditions with little overhead. First, the data structure should be linear with the program size to keep the persistence overhead acceptable, even for large-scale TPLs. Second, we should efficiently recover path conditions from the data structure to avoid repetitive analysis of TPLs. For instance, we can directly persist the source code of TPLs, which satisfies the first requirement. However, when analyzing a program using the TPLs, we must recompute the path conditions from the TPLs, which is costly. Third, since detecting different bugs requires different information from TPLs, the data structure should include bug-type-specific information.

We observe that the program dependence graph (PDG), a classic intermediate program representation, satisfies the requirements mentioned above and, thus, can serve as the highly-compressed data structure of TPL summaries. Figure 2 shows the PDG of the TPL

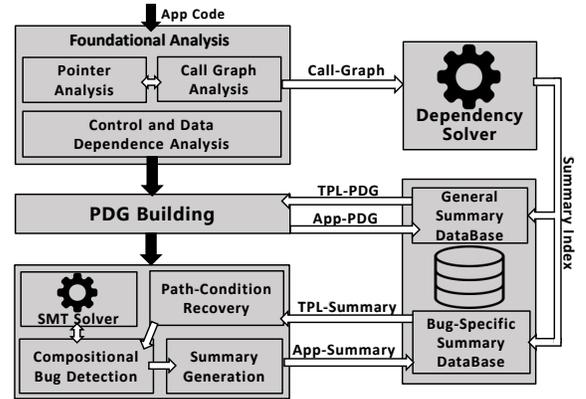


Figure 3: Architecture of LIBALCHEMY.

functions in Figure 1 (d), where the functions have been converted to SSA form. In the figure, we use  $v_i$  to denote the value of the variable or constant  $v$  at Line  $i$ . Basically, a PDG includes data-dependence edges and two kinds of control-dependence edges, as shown in the figure. A data-dependence edge may be labeled by a left-parenthesis ( $_i$  and right-parenthesis  $)_i$ , respectively, if the edge stands for a dependence relation at the entry and the exits of a callee function. The subscripts of the parentheses pair the entry (function call) and the exit (function return) at a call site. A true- and false-control-dependence edge means a program statement (i.e., the source of the edge) is reachable at runtime only if the condition it depends on (i.e., the destination of the edge) is evaluated to be true and false, respectively. Each node in the figure is either a statement or a  $\phi$  node representing a value that merges multiple values from different branches. For each  $\phi$  node, whenever one of its operands has a true-control-dependence edge to a condition, the other operand will have a false-control-dependence edge to the condition. The latter is omitted to keep the figure clear.

As an intermediate program representation, it is well-known that PDG is of linear size with the program and thus satisfies our first requirement. Also, we can recover the path condition via a simple graph traversal for a given data-flow path. For instance, for the first data-flow path in Table 1, i.e.,  $\textcircled{1} \leftrightarrow \textcircled{6} \leftrightarrow \textcircled{9} \leftrightarrow \textcircled{14}$ , where a numbered circle indicates a node of the PDG, we can collect its path condition from two aspects: (1) The values along the data-flow path have the same value, so we have the data-dependence condition  $0 = cRet_9 = cRet_{14} = cRet_{23}$ . (2) We have the condition  $\neg(sz_{11} \leq MAX) \wedge \neg(cRet_{14} \neq 0)$  along the control-dependence edges and the condition  $sz_8 = sz_{11}$  restricting the values in the control-dependence, so we can obtain the control-dependence condition by taking the conjunction of them. For the third requirement, bug-type-specific information can be easily recorded as data-flow paths in the figure. For instance, for checking the NPD bug we illustrated before, we can record the data-flow path  $\textcircled{1} \leftrightarrow \textcircled{6} \leftrightarrow \textcircled{9} \leftrightarrow \textcircled{14}$ , meaning that the TPL function may return a null pointer. Note that a data-flow path is much more compact than a conventional program path because it skips any unnecessary control flows in the code via data-flow relations. We detail this in the next section.

Figure 3 shows the architecture of our system. The input is the application code of which we would like to check the safety. Our static analysis first performs several fundamental analyses, such as pointer analysis and call graph analysis, to build the control and data-dependencies, yielding the PDG of the application code. During the call graph analysis, if a TPL function call is found, we will send a query request to our database. This enables the database to prepare any persisted summaries for further analysis. After building the PDGs of the application code, we connect them to the PDGs of TPL functions that the application depends on. As a result, we obtain a complete PDG depicting the control and data-dependencies from both the application code and the TPL functions. Finally, we check bugs using the complete PDG by querying data-flow paths and their path conditions. The path conditions are solved via an SMT solver. Since any application code could be used as a TPL in the future, we also persist the summaries of the application code into the database, which gradually enriches the database and empowers our bug-finding system.

### 3 PRELIMINARIES

This section presents several preliminaries of LIBALCHEMY. We first introduce the program dependence graph (§ 3.1) and then formally define feasible data-flow paths in the graph (§ 3.2). Lastly, we introduce static bug-finding (§ 3.3), which is formulated as a source-sink feasible path problem.

#### 3.1 Program Dependence Graph

To statically analyze a program, we utilize a classical graph representation, named program dependence graph (PDG), to depict the data flows in the program and encode their corresponding conditions.

**Definition 3.1.** (Program Dependence Graph) The program dependence graph of a program  $P$  is  $(V, E_d, E_c, \ell_d, \ell_c)$ , where

- $V$  contains the vertices that indicate the statements or, equivalently, the values defined by the statements.
- $v_1 \hookrightarrow v_2 \in E_d \subseteq V \times V$  is a data-dependence edge, indicating that  $v_2$  uses the value defined by  $v_1$ .
- $v_1 \rightarrow v_2 \in E_c \subseteq V \times V$  is a control-dependence edge, indicating that  $v_2$  is an if-statement and  $v_1$  is reachable when the if-condition of  $v_2$  is satisfied.
- $\ell_d$  maps each data-dependence edge to a parenthesis. The values of  $\ell_d(v_1 \hookrightarrow v_2)$  indicate the function calls and returns at different call sites, which are depicted by the left and right parentheses, respectively. For other edges,  $\ell_d(v_1 \hookrightarrow v_2) = \varepsilon$ .
- $\ell_c$  maps each control-dependence edge to *true* or *false*, indicating the true or false branches, respectively.

**Example 3.1.** Figure 2 shows the PDG of the program in Figure 1(d). The data-dependence edge from ⑬ to ⑱ shows that the value of `ret` defined at Line 26 is returned by `alloc`. The control-dependence edge from ⑦ to ⑪ indicates that the value of `cRet` after the if-statement is affected by whether `cRet → buf` is null at Line 12. The edges labeled with parentheses show two invocations of `alloc`.

Following existing studies [48, 52], we unroll the loops and recursive functions before the analysis, and thus, a PDG is essentially a directed acyclic graph (DAG). Additionally, the PDG resolves

indirect data-dependencies introduced by pointers. The construction of the PDG is not a focus of our work, and we utilize existing techniques [50, 51] as an off-the-shelf module in our analysis.

#### 3.2 Feasible Data-Flow Path

Given a PDG, we can reason how the data flows of interest are propagated by inspecting a particular class of paths, namely the *feasible data-flow paths*. Before defining them formally, we first introduce the concept of the *path condition*.

**Definition 3.2.** (Path Condition) Consider a data-flow path  $p : u_1 \hookrightarrow \dots \hookrightarrow u_k$  in the PDG  $G = (V, E_d, E_c, \ell_d, \ell_c)$  of a program  $P$ , where  $u_i \hookrightarrow u_{i+1} \in E_d$  for  $1 \leq i \leq (k - 1)$ . The path condition  $C(p)$  is the constraint over  $V$  that holds if and only if the data flows along  $p$  happen in a concrete execution.

As shown in Definition 3.1, a PDG encodes the data-dependence and control-dependence simultaneously. Based on the PDG, we can track the data-flow paths of interest along the data-dependence edges and recover the condition of the data-flow path based on two kinds of dependencies. Furthermore, we can define the notion of *feasible data-flow paths* as follows.

**Definition 3.3.** (Feasible Data-Flow Path) Given a PDG  $G = (V, E_d, E_c, \ell_d, \ell_c)$ , a data-flow path  $p : u_1 \hookrightarrow \dots \hookrightarrow u_k$  is feasible if and only if its path condition  $C(p)$  is satisfiable and its label string  $\mathcal{L}(p)$  belongs the extended Dyck-CFL [53].

**Example 3.2.** Consider the data-flow path ① $\hookrightarrow$ ⑥ $\hookrightarrow$ ⑨ $\hookrightarrow$ ⑭, which corresponds to the first path in Table 1. We get the path condition by taking the conjunction with two conditions in the first row of Table 1. Obviously, the path condition is satisfiable, so the data-flow path is feasible.

#### 3.3 Static Bug-Finding

Given a program, we can leverage its PDG to detect various bugs statically. Before providing concrete examples, we first introduce the *source-sink feasibility* problem, which further enables us to formulate the detection of various types of bugs.

**Definition 3.4.** (Source-Sink Feasibility) Given  $v_{src}, v_{sink} \in V$ , the source-sink feasibility from  $v_{src}$  to  $v_{sink}$  determines whether a feasible data-flow path exists from  $v_{src}$  to  $v_{sink}$ .

The source-sink feasibility serves as an expressive formulation of static bug-finding. Many bug detectors can be formalized as the instantiations of source-sink feasibility queries. Formally, their bug specifications are defined as follows.

**Definition 3.5.** (Bug Specification) A bug specification  $\tau$  is a triple  $(\sigma_{src}, \sigma_{sink}, \delta_{reach})$ . Here, the predicates  $\sigma_{src}$  and  $\sigma_{sink}$  depict the properties the sources and sinks should satisfy, respectively. When the indicator  $\delta_{reach}$  is true, a bug occurs if a source can reach a sink via a feasible data-flow path. Otherwise, a bug occurs if there exists no feasible path from a source to a sink.

It is worth noting that while Definition 3.5 does not cover all possible bug types, it has covered a wide range of them, which are often referred to as the *source-not-sink* bug types and *source-must-sink* bugs [54]. In what follows, we use NPD and ML as two instances to demonstrate the scope of our static bug-finding system.

**Example 3.3.** (Null Pointer Dereference) We define  $\sigma_{src}(v)$  to decide whether  $v$  is  $a = \text{NULL}$ , where  $a$  is a pointer expression.  $\sigma_{sink}(v)$  is true if and only if  $v$  accesses the memory object pointed by a pointer expression. Besides,  $\delta_{reach}$  is true. To detect NPD, we need to examine whether a feasible data-flow path connects a source to a sink. Thus, NPD is a source-not-sink bug type.

**Example 3.4.** (Memory Leak)  $\sigma_{src}(v)$  is true if and only if  $v$  is  $a = \text{malloc}()$ , where  $a$  is an arbitrary pointer expression.  $\sigma_{sink}(v)$  is true if and only if  $v$  is  $\text{free}(a)$ . Meanwhile,  $\delta_{reach}$  is set to false. If a source cannot reach a sink in the PDG, a potential memory leak may happen. Hence, ML is a source-must-sink bug type.

According to Definition 3.3, determining the source-sink feasibility involves examining the satisfiability of the path condition and checking whether the label string belongs to the extended Dyck-CFL, which ensures the path and context sensitivity, respectively. However, achieving such precision is far from trivial, especially in the presence of TPLs. Even a small-size application can rely on a large number of TPLs, increasing the total size of the analyzed code. Besides, redundant reasoning of commonly-used TPLs wastes enormous analysis resources, such as time and memory. To tame TPLs in static bug-finding, we demonstrate a system named LIBALCHEMY to persist the summaries of TPLs, which can significantly improve the scalability of static bug-finding.

## 4 LIBALCHEMY SYSTEM DESIGN

This section presents the design of LIBALCHEMY. To reuse the analysis results of TPLs, we propose a multi-layer summary design (§ 4.1), which supports us in collecting bug-related data-flow paths in TPLs and recovering path conditions **on demand** (§ 4.2). Lastly, we demonstrate the overall algorithm of static bug-finding with our persistence design (§ 4.3).

### 4.1 Bug-Type-Specific Summary

As shown in § 3.3, the static bug-finding explores feasible paths from a specific form of source and sink. In the presence of the TPLs, such feasible paths are the concatenations of specific data-flow paths in the TPLs and applications. Based on this intuition, we define the *bug-type-specific summary* formally.

**Definition 4.1.** (Bug-Type-Specific Summary) Given a bug specification and the PDG of a program, the bug-type-specific summary  $S$  maps a pair of vertices  $(u_1, u_2)$  to a set of data-flow paths, where  $(u_1, u_2) \in (V_{src}^f \cup V_{arg}^f \cup V_{out}^f) \times (V_{sink}^f \cup V_{in}^f \cup V_{ret}^f)$ . Here,  $V_{src}^f$  and  $V_{sink}^f$  contain the sources and sinks in the function  $f$ , respectively, while  $V_{arg}^f$ ,  $V_{ret}^f$ ,  $V_{in}^f$ , and  $V_{out}^f$  contain the arguments, returns, inputs, and outputs of  $f$ , respectively.

Intuitively, the bug-type-specific summary abstracts the semantics of TPL functions with respect to the bug specification. Although a lot of nodes and paths may exist in the PDG of a TPL, only a few critical data-flow paths really matter in the bug-finding. From a high-level perspective, the bug-type-specific summary provides a compact semantic signature of the TPL, guiding the static bug finder to delve into the TPL functions in the analysis of the application.

**Example 4.1.** For the NPD detection, we can obtain a bug-type-specific summary summarizing the data-flow path  $\textcircled{5} \leftrightarrow \textcircled{6} \leftrightarrow$

$\textcircled{9} \leftrightarrow \textcircled{14}$ . The path shows the propagation of the *null* value, causing an NPD bug in the program as shown in Figure 1.

Note that several summaries are shared in the detection of different bug types. Specifically, if  $(u_1, u_2) \in (V_{arg}^f \times V_{in}^f) \cup (V_{arg}^f \times V_{ret}^f) \cup (V_{out}^f \times V_{in}^f) \cup (V_{out}^f \times V_{ret}^f)$ , we can safely reuse them as the summarized data-flow paths do not change. Thus, we reuse the four kinds of summaries and avoid generating and storing duplicates for different bug types.

### 4.2 TPL Summary Persistence

The bug-type-specific summaries depict all possible ways of forming a data-flow path that potentially triggers a bug. To avoid analyzing a TPL redundantly, we need to preserve the bug-type-specific summary of the TPL for each bug type. In this section, we first present how to design the summary for a TPL (§ 4.2.1) and then propose an algorithm to achieve the summary persistence (§ 4.2.2).

**4.2.1 Two-layer Summary.** To achieve path sensitivity, we can explicitly persist the path condition of each summarized data-flow path. However, deep calling contexts can dramatically increase the sizes of path conditions, which introduces massive overhead in analyzing applications heavily using TPLs. Also, not all the summarized data-flow paths can form a path indicating a bug, so exhaustively persisting path conditions introduces unnecessary overhead.

Interestingly, we notice that the path conditions and the PDG are synergic, i.e., the path conditions have been sparsely encoded on the PDG already. Meanwhile, the PDG can be utilized to recover the path conditions for the bug-type-specific summaries of different bug types and, thus, serves as a sharing representation of path conditions. Therefore, we propose the two-layer summary for a TPL to avoid unnecessary storage.

**Definition 4.2.** (Two-layer Summary) Given a program  $P$  and a bug specification, the two-layer summary of  $P$  is  $(G, S)$ , where  $G$  is the PDG of the program  $P$  as its *general summary*, and  $S$  is the bug-type-specific summary of the program  $P$ .

**Example 4.2.** For the TPL libogg-1.3.4 in Figure 1(d), we obtain the PDG in Figure 2 as its general summary. For NPD detection, we construct the bug-type-specific summaries of function alloc:

$$S(\textcircled{15}, \textcircled{20}) = \{\textcircled{15} \leftrightarrow \textcircled{18} \leftrightarrow \textcircled{20}\}$$

$$S(\textcircled{16}, \textcircled{20}) = \{\textcircled{16} \leftrightarrow \textcircled{18} \leftrightarrow \textcircled{20}\}$$

Similarly, we can obtain the bug-type-specific summaries of function ogg\_sync\_buffer as follows:

$$S(\textcircled{3}, \textcircled{12}) = \{\textcircled{3} \leftrightarrow \textcircled{12}\}$$

$$S(\textcircled{1}, \textcircled{14}) = \{\textcircled{1} \leftrightarrow \textcircled{6} \leftrightarrow \textcircled{9} \leftrightarrow \textcircled{14}\}$$

$$S(\textcircled{2}, \textcircled{14}) = \{\textcircled{2} \leftrightarrow \textcircled{7} \leftrightarrow \textcircled{9} \leftrightarrow \textcircled{14}\}$$

$$S(\textcircled{5}, \textcircled{14}) = \{\textcircled{5} \leftrightarrow \textcircled{6} \leftrightarrow \textcircled{9} \leftrightarrow \textcircled{14}\}$$

Then we can recover the path condition for each data-flow path, for example, the path in Example 3.2.

The two-layer summary design does not actually introduce large memory overhead. First, the PDG is a sparse graph. The number of vertices is linear to the program size, while the number of edges is almost linear to the number of vertices. The cost of persisting an entire PDG is affordable for a large TPL. Second, we only track

**Algorithm 1:** Persisting TPL summaries

---

```

1 Procedure persistSmry( $Libs, \sigma_{src}, \sigma_{sink}, \tau = (\sigma_{src}, \sigma_{sink}, \delta_{reach})$ ):
2    $Libs \leftarrow \text{TopologicalLibSort}(Libs)$ ;
3   for  $Lib$  in  $Libs$  do
4      $Lib \leftarrow \text{TopologicalFuncSort}(Lib)$ ;
5      $G = (V, E_d, E_c, \ell_d, \ell_c) \leftarrow \text{constructPDG}(Lib)$ ;
6      $\text{dumpPDG}(Lib, G)$ ;
7    $S \leftarrow \perp$ ;
8   for  $Lib \in Libs, f \in Lib$  do
9      $paths \leftarrow \text{pathFrom}(V_{src}^f \cup V_{arg}^f \cup V_{out}^f)$ ;
10    foreach  $p : u_1 \hookrightarrow \dots \hookrightarrow u_n \in paths$ 
11      if  $u_n \in V_{in}$  then
12         $u'_n \leftarrow \text{getConnectedArg}(u_n)$ ;
13        if  $S(u'_n, \cdot) \neq \emptyset$  then
14           $S(u_1, u_n) \leftarrow S(u_1, u_n) \cup \{p\}$ ;
15        if  $u_1 \in V_{out}$  then
16           $u'_1 \leftarrow \text{getConnectedRet}(u_1)$ ;
17          if  $S(\cdot, u'_1) \neq \emptyset$  then
18             $S(u_1, u_n) \leftarrow S(u_1, u_n) \cup \{p\}$ ;
19     $\text{dumpBugSmry}(\sigma_{src}, \sigma_{sink}, S)$ ;

```

---

and store the data-flow paths with specific start and end points. Although the branches in a single function can introduce multiple paths, we do not suffer from the path explosion problem. The key reason is that we first create disjunctions of intraprocedural path conditions and then concatenate them as the condition of an interprocedural path. Our evaluation will also demonstrate the low memory overhead of our two-layer design.

**4.2.2 Persisting TPL Summaries.** To persist the two-layer summary for a given TPL, we need to construct the PDG for the TPL first and then collect the bug-type-specific summary. Besides, we notice that a TPL may depend on other TPLs, i.e., its function may invoke other TPL functions. Following many static analyses in generating function summaries, we utilize the dependency relation among the TPLs and process them bottom-up. The two-layer summary of a TPL is generated once all the dependent TPLs are processed.

Algorithm 1 shows the details of persisting two-layer summaries for TPLs of an application. Initially, it takes as input a TPL list and a bug specification. We sort all the TPLs in the list according to their dependency relation (Line 2). We sort each TPL's functions based on the callee-caller relationship (Line 4). Following existing techniques of program dependence analysis [51], we can generate the PDG of each TPL (Line 5) and further store the PDG into the disk for persistence (Line 6). Then we process each TPL according to the sorting order and identify bug-type-specific summaries by examining data-flow paths in each function (Lines 8–18). For the data-flow path  $p$  ending at an input node, we examine the connected argument node  $u'_n$  (Lines 11–12), which is located in the PDG of the TPL that  $Lib$  relies on. If the argument node is the start point of a summarized path, we persist the path  $p$  (Line 13–14) as an interprocedural data-flow path that can be formed by stitching  $p$  with the summarized path. We process the case where  $u_1 \in V_{out}$  in a similar way (Line 15–18).

Intuitively, Algorithm 1 generalizes existing bottom-up interprocedural analysis [40, 55] to the function summary generation. The invoked TPL functions are processed before the ones invoking them. Notably, we only store the data-flow paths of a caller that may form an interprocedural data-flow path by concatenating them

**Algorithm 2:** Recovering path conditions

---

```

1 Procedure getPathCond( $G = (V, E_d, E_c, \ell_d, \ell_c), p$ ):
2    $\varphi \leftarrow true; \psi \leftarrow true$ ;
3   foreach  $v_i$  in  $p$  do
4      $\varphi \leftarrow \varphi \wedge \text{getDataDepCond}(v_i, E_d)$ ;
5      $\psi \leftarrow \psi \wedge \text{getCtrlDepCond}(v_i, E_d, E_c)$ ;
6   return  $\varphi \wedge \psi$ 
7 Procedure getDataDepCond( $v_i, E_d$ ):
8    $\varphi \leftarrow \text{expr}(v_i)$ ;
9   foreach  $v_j \hookrightarrow v_i \in E_d$  do
10     $\varphi \leftarrow \varphi \wedge \text{getDataDepCond}(v_j, E_d)$ ;
11  return  $\varphi$ 
12 Procedure getCtrlDepCond( $v_i, E_d, E_c$ ):
13   $\varphi \leftarrow true; \psi \leftarrow false$ ;
14  foreach  $v_j \rightarrow v_i \in E_c$  do
15     $\psi_{tmp} \leftarrow \text{getCtrlDepCond}(v_j, E_d, E_c)$ ;
16     $\psi \leftarrow \psi \vee ((\neg \ell_c(v_j \rightarrow v_i) \oplus \text{expr}(v_j)) \wedge \psi_{tmp})$ ;
17   $\varphi \leftarrow \varphi \wedge \text{getDataDepCond}(v_j, E_d)$ ;
18  return  $\psi \wedge \varphi$ 

```

---

with the paths of its callee. This effectively avoids collecting and persisting any unnecessary bug-type-specific summaries.

**4.3 Static Bug-Finding with Persistence**

Based on two-layer persistent summaries, we can not only concatenate the summarized data-flow paths in the TPLs with the data-flow paths in the application code but also recover the path conditions from the PDGs of the TPLs. This section details the procedure of recovering path conditions (§ 4.3.1) and statically finding bugs with persistent summaries (§ 4.3.2).

**4.3.1 Recovering Path Conditions.** As shown in Definitions 3.1 and 3.2, a PDG provides the necessary ingredient to track each data-dependence edge in the path and construct path conditions. Such a process can be achieved by graph traversal formulated by Algorithm 2. Basically, we take the conjunction of the data-dependence condition and the control-dependence condition of each node in the path  $p$  (Lines 2–6). To construct the data-dependence condition of a node  $v_i$ , we create a constraint based on the value definition induced by  $v_i$  (Line 8). Then we transitively collect the data-dependence conditions of the parents of  $v_i$  (Line 10), based on which we take the conjunction as the data-dependence condition of  $v_i$ . To construct the control-dependence condition of  $v_i$ , we transitively collect the control-dependence conditions of the nodes that are control-dependent on  $v_i$  (Line 15). We then create the disjunction for case analysis (Line 16) and append the data-dependence condition (Line 17) to form the control-dependence condition of  $v_i$  (Line 18). Because a PDG is loop-free, the functions `getDataDepCond` and `getCtrlDepCond` must terminate upon the PDG. Moreover, the path condition recovery can be achieved in linear time to the graph size for a given data-flow path, which permits us to perform path-sensitive analysis efficiently.

**Example 4.3.** Consider the data-flow path  $\textcircled{1} \hookrightarrow \textcircled{6} \hookrightarrow \textcircled{9} \hookrightarrow \textcircled{14}$  in Figure 2. By traversing the PDG along the data-dependence edge, we can construct the data-dependence condition  $0 = cRet_9 = cRet_{14} = cRet_{23}$ . Similarly, we can construct the control-dependence condition  $\neg(sz_{11} \leq MAX) \wedge \neg(cRet_{14} \neq 0) \wedge sz_{11} = sz_8$ , where  $sz_{11} = sz_8$  is induced by the data-dependence edge  $\textcircled{4} \hookrightarrow \textcircled{8}$ . Finally, we can obtain the path condition by taking their conjunction.

**Algorithm 3:** Bug-finding with persistent summary

```

1 Procedure findBug( $P, Libs, \tau = (\sigma_{src}, \sigma_{sink}, true)$ ):
2   persistSmry ( $Libs, \sigma_{src}, \sigma_{sink}$ );
3    $G \leftarrow \text{constructPDG}(P)$ ;
4    $V_{src}, V_{sink} \leftarrow \text{getSrcSink}(G, \sigma_{src}, \sigma_{sink})$ ;
5   foreach  $p \in \text{pathFrom}(V_{src})$  do
6     search( $p, V_{sink}$ );
7 Procedure search( $p, V_{sink}$ ):
8    $v_1 \leftrightarrow v_2 \leftrightarrow \dots \leftrightarrow v_n \leftarrow p$ ;
9   if  $v_n \in V_{in} \cup V_{ret}$  then
10     $v'_n \leftarrow \text{getConnectedArgOrOut}(v_n)$ ;
11     $P' \leftarrow \text{loadBugSmryAt}(v'_n)$ ;
12    foreach  $p' \in P'$  do
13      search(connect( $p, p'$ ),  $V_{sink}$ );
14  if  $v_n \in V_{sink}$  and  $\mathcal{L}(p) \in \text{extended Dyck-CFL}$  then
15     $G \leftarrow \text{loadPDG}(p)$ ;
16    if getPathCond( $G, p$ ) = SAT then
17      reportBug( $p$ );

```

4.3.2 *Finding Bugs with Persistent Summaries.* Leveraging path condition recovery, we can reuse persistent TPL summaries in static bug-finding. The key idea is to scan program paths from sources to sinks and load previously computed persistent bug-type-specific summaries on demand, which achieves full sensitivity with low time and memory costs.

Without the loss of generality, we consider NPD-style bug detection as an example to demonstrate bug-finding with persistent summaries, where a bug occurs when a source reaches a sink via a feasible path. We formulate the technical design in Algorithm 3. It first prepares the summaries for all the TPLs and constructs the PDG of an application  $P$  (Line 2–3). After identifying sources and sinks based on the bug specification (Line 4), it traverses from the desired source and finds a path ending with the desired sinks (Line 5–6). During the path search, if the path encounters a call or return statement that requires the TPL summaries to connect the data-flow path onwards, it loads the summaries persisted at this statement (Lines 9–11) to continue the search (Line 13). Remarkably, the function `getConnectArgOrOut` fetches the argument or output  $v'_n$  according to the input or return  $v_n$ , respectively, while the function `loadBugSmryAt` collects all the bug-type-specific summaries starting from  $v'_n$ . When the path finally reaches a sink, it validates whether the label string of the path belongs to the extended Dyck-CFL [53] (Line 14). If so, it loads the persisted PDG to recover the path condition using Algorithm 2 and then decides the satisfiability of the path condition with a solver (Line 15–16). For NPD detection, a feasible data-flow path indicates a potential NPD (Line 17).

Notably, Algorithm 3 is general enough to support the detection of other kinds of bugs, such as the ML in Example 3.4. By altering the condition of reporting a bug (Line 16–17), we can easily extend Algorithm 3 to detect the bugs caused by the sources that can not reach any sink. As long as we generate the summaries for the TPLs, we can delve into the TPL functions when analyzing the application code with low overhead.

## 5 IMPLEMENTATION

We implemented LIBALCHEMY upon the state-of-the-art path-sensitive bug-finding system PINPOINT [51]. Following PINPOINT, LIBALCHEMY analyzes the program in LLVM IR after loop unrolling and invokes

**Table 2: Statistics of subjects. #Lines<sub>A</sub>, #Lines<sub>L</sub>, #Lines, and #TPL are application sizes, TPL sizes, total sizes (KLoC), and the numbers of transitively dependent TPLs, respectively.**

ID	Project	Version	#Lines <sub>A</sub>	#Lines <sub>L</sub>	#Lines	#TPL
1	libxi	1.8	8	1,210	1,218	13
2	opusfile	0.12	9	1,141	1,150	10
3	libshout	2.4.5	28	2,163	2,191	14
4	hwloc	2.7.0	57	2,856	2,913	23
5	tmux	3.2	68	1,727	1,795	12
6	libsndfile	1.0.31	75	1,311	1,386	12
7	transmission	3.0	94	5,655	5,749	42
8	dynamips	0.2.14	104	3,118	3,222	10
9	libsdl	1.2	154	1,446	1,600	6
10	openldap	2.5.16	331	3,566	3,897	10
11	vim	8.1.2269	469	4,253	4,722	41
12	vlc	3.0.9	546	8,202	8,748	35
13	openssl	1.1.1	616	1,829	2,445	8
14	binutils	2.34	1,782	1,087	2,869	9
15	mariadb	10.3	2,469	8,750	11,219	48
Total			<b>6,810</b>	<b>48,314</b>	<b>55,124</b>	

the SMT solver Z3 [56] to determine the satisfiability of a path condition. We provide more implementation details as follows, including TPL preparation and checker instantiation.

**TPL Preparation.** To collect the TPLs of an application, we adopt Ubuntu’s package manager `dpkg` [57] to obtain its dependencies. Specifically, we transitively call `dpkg` for each dependent package until all the dependencies are collected. After downloading the source code using `dpkg`, we compile it into the LLVM IR for further analysis. Though our implementation relies on the package manager `dpkg`, our design for collecting TPLs is general. It can be seamlessly extended to other library management platforms, such as `yum` package manager [58] from RedHat OSes.

**Checker Instantiation.** We have implemented checkers to detect four bug types: NPD, UAF, UUV, and ML. As shown in Definition 3.5, an NPD, UAF, and UUV are source-not-sink bug types, while an ML is a source-must-sink bug type. We select these four checkers because they can lead to severe memory errors, and developers typically give feedback promptly on them. We provide the bug specifications of the four bug types in configuration files, which LIBALCHEMY further parsed to generate bug-type-specific summaries in the bug detection. The four instantiations demonstrate the generality of our design in detecting value-flow bugs.

## 6 EVALUATION

In this section, we evaluate the effectiveness and scalability of LIBALCHEMY by answering the following research questions:

- **RQ1:** How effectively is LIBALCHEMY detecting security bugs caused by TPL function misuse?
- **RQ2:** How scalable is LIBALCHEMY when reusing the persistent TPL summaries?
- **RQ3:** How much time and space overhead does LIBALCHEMY incur when persisting TPL summaries?

### 6.1 Experimental Setup

**Subject Collection.** We selected 15 widely-used and well-known C/C++ open-source projects to evaluate LIBALCHEMY as shown in Table 2, where their source code and the source code of their dependent TPLs are all available in the Ubuntu sources. The selected projects have at least 100 stars, update frequently, and use TPLs as

**Table 3: Bug detection capability comparison. #Bug<sub>TPL</sub> represents the numbers of NPD, UAF, UUV, and ML bugs that are related to TPLs and detected by LIBALCHEMY. #F/#C/#T represents the numbers of fixed, confirmed, and reported bugs respectively. The mark “✓” indicates a baseline has the same detection capability as LIBALCHEMY within time and memory budget, while the mark “×” indicates the opposite.**

ID	LIBALCHEMY		LIBFREE	LIBWP	LIBCON	LIBPDG
	#Bug <sub>TPL</sub>	#F/#C/#T				
1	(2, 0, 0, 0)	1/1/2	×	✓	✓	✓
2	(2, 0, 0, 0)	1/1/2	×	✓	✓	✓
3	(2, 0, 0, 0)	1/2/2	×	×	✓	✓
4	(5, 0, 0, 0)	0/5/5	×	×	✓	✓
5	(2, 0, 0, 0)	1/1/2	×	×	×	✓
6	(3, 0, 0, 0)	0/3/3	×	✓	×	✓
7	(1, 0, 0, 0)	0/0/1	×	×	×	✓
8	(0, 0, 2, 0)	0/2/2	×	×	✓	✓
9	(0, 1, 0, 0)	1/1/1	×	×	×	✓
10	(6, 0, 0, 0)	6/6/6	×	×	✓	✓
11	(1, 0, 0, 0)	1/1/1	×	×	×	✓
12	(2, 0, 0, 0)	0/1/2	×	×	✓	✓
13	(21, 0, 0, 0)	21/21/21	×	×	✓	✓
14	(0, 0, 0, 7)	7/7/7	×	×	×	✓
15	(4, 0, 0, 0)	1/3/4	×	×	×	×
Total	(51, 1, 2, 7)	41/55/61				

their dependencies. The sizes of these projects themselves range from 8 KLoC to 2,469 KLoC. Since each project often depends on or transitively depends on many TPLs (6~48 TPLs for each project), the code sizes of their TPLs are significantly larger than the projects themselves, with a range from 1,087 KLoC to 8,750 KLoC. To set up the environment, we generate summaries for all TPLs into the database using LIBALCHEMY by offline analysis.

**Baselines.** We implement four settings based on the state-of-the-art static bug-finding tool PINPOINT [51] as the baselines:

- LIBFREE: Apply PINPOINT under the setting of neglecting the potential effects of TPLs.
- LIBWP: Apply PINPOINT to analyze both the application and its TPLs (i.e., whole-program analysis).
- LIBCON: Apply PINPOINT by using the path constraints of the bug-type-specific data-flow paths as the summaries, following conventional path-sensitive precise summaries [48].
- LIBPDG: Apply PINPOINT by using only the PDGs as the summaries. It should be noted that, the way of loading PDGs is same as LIBALCHEMY which is on-demand. We use this setting to evaluate the effects of the bug-type-specific summaries.

**Environment.** All experiments were run on an Ubuntu 20.04 server with an Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz 40-core processor and 512G of memory.

## 6.2 RQ1: Effectiveness of Bug Detection

To reduce the subjectivity of evaluation, we follow the practice of the existing work [51] and set a high bar for “true positive”: bugs confirmed by the developers of the evaluated subjects.

As shown in Table 3, LIBALCHEMY detected 61 new bugs related to TPLs. Note that LIBALCHEMY also discovers 218 bugs that are irrelevant to TPLs. Since these bugs are not the focus of this work and can be detected by other baselines, we do not discuss them in

the following. We submitted all of 61 TPL-related bugs to developers for confirmation. It is shown that NPDs are the most common vulnerabilities (51), while the other three bug checkers report fewer bugs (1+2+7). Upon the submission, 55 of 61 bugs have been confirmed as true bugs [59], yielding a precision of 90.16% (55/61), while 41 of them have already been fixed. The other six bugs are false positives, including one pending review and five denied by the developers. Notably, some confirmed bugs are from high-quality systems such as MariaDB, vim, openssl, and binutils which have been constantly scanned by Coverity [60]. Meanwhile, eleven of the confirmed bugs are assigned with CVE IDs for their high impact on software security [59]. For example, we detected a MariaDB bug that is hidden for over eight years. This bug is serious enough to deserve its CVE ID: CVE-2022-47015.

Table 3 also demonstrates the bug detection capability of other baselines. LIBFREE neglects the side effects of TPLs, and thus cannot detect any TPL-related bugs. LIBWP, LIBCON, and LIBPDG consider the presence of TPLs, and thus theoretically have the same bug detection capability as LIBALCHEMY. However, due to the scalability issues (i.e., running out of time or memory budgets as shown in Table 4), LIBWP, LIBCON, and LIBPDG fail to analyze twelve, seven, and one projects, thus missing the bugs for these projects. These results demonstrate the superiority of LIBALCHEMY.

We also conduct a deep analysis on the five bug reports that the developers deny and summarize two reasons for the denial. First, three bug reports are denied because the triggering conditions of the bugs rarely happen in practice [61–63]. A typical example is that some TPL APIs return a null value only when the system runs out of memory (e.g., [62]). Second, the remaining denied bug reports are technically false positives due to the inherited limitations of the foundation framework [51], such as imprecise pointer analysis in containers, loop unrolling, etc.

## 6.3 RQ2: Scalability of Reusing Summaries

To measure the scalability, we compare the time and memory usage of LIBALCHEMY with LIBWP, LIBCON, LIBPDG, and LIBFREE, respectively. The timeout is set to 12 hours, the typical nightly-build duration. The memory budget is set to 512GB, the same as our server’s memory. The time speedup is calculated using the baselines’ analysis time divided by those of LIBALCHEMY. The memory reduction is calculated by using the difference between the memory usage of the baseline and LIBALCHEMY divided by the memory usage of the baselines. If the analysis encounters OOT or OOM, we conservatively set them to 12 hours and 512GB, respectively.

As shown in Table 4, LIBALCHEMY outperforms two soundness-equivalent baselines, LIBWP and LIBCON, achieving a time speedup of 18.56× and 12.77× on average, respectively. This is because our two-layer summary design enables efficient path condition recovery. Notably, LIBALCHEMY achieves a 5.17× speedup over LIBPDG, another soundness-equivalent baseline. This is because LIBALCHEMY eliminates infeasible paths and preserves all feasible paths when analyzing TPL code. This saves the time of searching paths for analysis of application with the acceptable cost of space overhead for persisting the second-layer summaries.

LIBALCHEMY saves 91.49% and 90.51% of memory usage compared with LIBWP and LIBCON, respectively. This is because LIBCON

**Table 4: Performance comparison. TS and MR represent the time speedup and memory reduction of LIBALCHEMY over the baselines. MR2 represents the memory reduction of the baseline LIBFREE over LIBALCHEMY. OOT and OOM indicate the analysis runs out of time or memory, respectively. Avg represents the geometric average.**

ID	LIBALCHEMY		LIBWP			LIBCON			LIBPDG			LIBFREE						
	Time	Mem	Time	TS	Mem	MR	Time	TS	Mem	MR	Time	TS	Mem	MR	Time	TS	Mem	MR2
1	3m	10.28G	11.12h	222.40	167.97G	93.88%	2.48h	49.60	152.34G	93.25%	2.26h	45.20	11.86G	13.32%	1m	0.33	3.18G	69.07%
2	9m	11.35G	7.73h	51.53	125.59G	90.96%	2.78h	18.53	95.69G	88.14%	2.90h	19.33	12.31G	7.80%	7m	0.78	4.50G	60.35%
3	11m	17.76G	OOT	65.45	268.71G	93.39%	5.31h	28.96	400.66G	95.57%	4.12h	22.47	18.99G	6.48%	7m	0.64	7.68G	56.76%
4	23m	22.06G	OOT	31.30	489.89G	95.50%	9.15h	23.87	479.90G	95.40%	1.71h	4.46	23.35G	5.52%	18m	0.78	7.28G	67.00%
5	48m	29.53G	OOT	15.00	253.15G	88.33%	OOT	15.00	OOM	94.23%	1.45h	1.81	30.04G	1.70%	38m	0.79	8.33G	71.79%
6	6m	12.50G	10.00h	100.00	291.64G	95.71%	OOT	120.00	OOM	97.56%	2.27h	22.70	13.41G	6.79%	5m	0.83	3.78G	69.79%
7	1.17h	20.19G	OOT	10.26	OOM	96.06%	OOT	10.26	OOM	96.06%	6.19h	5.29	25.40G	20.51%	58m	0.83	16.83G	16.64%
8	38m	20.26G	OOT	18.95	OOM	96.04%	11.61h	18.33	435.61G	93.35%	2.58h	4.07	24.37G	16.86%	34m	0.89	6.48G	68.02%
9	34m	14.37G	OOT	21.18	270.81G	94.69%	OOT	21.18	OOM	97.19%	1.81h	3.19	15.68G	8.35%	13m	0.38	4.19G	70.84%
10	41m	21.67G	OOT	17.56	OOM	95.77%	10.55h	15.44	306.87G	92.94%	1.51h	2.21	24.20G	10.45%	29m	0.71	8.63G	60.18%
11	5.85h	147.27G	OOT	2.05	460.85G	68.04%	OOT	2.05	OOM	71.24%	8.01h	1.37	153.97G	4.35%	25m	0.07	25.70G	82.55%
12	25m	21.78G	OOT	28.80	OOM	95.75%	4.85h	11.64	130.29G	83.23%	3.65h	8.76	23.20G	6.12%	27m	1.08	6.35G	70.84%
13	53m	28.45G	OOT	13.58	375.72G	92.43%	3.93h	4.45	143.26G	80.14%	2.63h	2.98	31.61G	10.00%	52m	0.98	23.06G	18.95%
14	2.85h	26.42G	OOT	4.21	389.13G	93.21%	OOT	4.21	OOM	94.84%	5.86h	2.06	32.80G	19.45%	10.58h	3.71	18.31G	30.70%
15	9.67h	65.90G	OOT	1.24	OOM	87.13%	OOT	1.24	OOM	87.13%	OOT	1.24	73.47G	10.30%	OOT	1.24	45.64G	30.74%
Avg				≥18.56		≥91.49%		≥12.77		≥90.51%		≥5.17		8.37%		≥0.71		50.91%

**Table 5: Time and space overhead of persisting summaries. Disk and Time stand for the disk space and time cost, respectively. One Checker includes NPD. Two Checkers include NPD and UAF. Four Checkers include NPD, UAF, UUV, and ML.**

ID	LIBCON						LIBALCHEMY							
	One Checker		Two Checkers		Four Checkers		First-Layer		One Checker		Two Checkers		Four Checkers	
	Disk	Time	Disk	Time	Disk	Time	Disk	Time	Disk	Time	Disk	Time	Disk	Time
1	469.99G	17.75h	755.73G	31.95h	1434.25G	50.70h	17.52G	20m	0.87G	2m	1.33G	3m	2.85G	6m
2	458.46G	17.25h	736.38G	31.05h	1384.70G	48.70h	17.10G	18m	0.84G	2m	1.28G	3m	2.75G	6m
3	491.38G	19.32h	906.67G	34.77h	1630.98G	56.35h	22.49G	22m	0.96G	3m	1.63G	5m	3.39G	9m
4	670.49G	16.40h	1295.02G	36.27h	2401.76G	67.11h	30.02G	24m	1.66G	5m	3.13G	8m	5.96G	16m
5	488.20G	18.28h	794.58G	32.91h	1512.85G	52.49h	17.78G	21m	0.90G	3m	1.41G	7m	3.04G	10m
6	468.30G	17.25h	757.78G	31.05h	1430.34G	48.70h	17.84G	21m	0.87G	2m	1.33G	3m	2.84G	6m
7	646.78G	25.28h	1282.78G	45.51h	2423.58G	78.07h	27.34G	25m	1.59G	6m	3.17G	9m	6.30G	17m
8	485.97G	19.30h	890.07G	34.73h	1611.24G	56.23h	22.12G	23m	0.99G	3m	1.64G	5m	3.50G	10m
9	537.91G	18.43h	897.66G	33.18h	1732.51G	53.10h	21.32G	20m	0.91G	3m	1.49G	7m	3.25G	10m
10	477.26G	17.82h	765.31G	32.07h	1447.48G	50.66h	17.44G	21m	0.87G	2m	1.34G	3m	2.90G	6m
11	688.58G	23.98h	1271.71G	43.16h	2472.52G	72.41h	27.93G	26m	1.52G	5m	2.73G	9m	5.59G	17m
12	525.70G	18.60h	983.9G	33.48h	1950.94G	53.8h	22.69G	24m	1.01G	3m	1.67G	6m	3.64G	15m
13	457.15G	17.236h	733.36G	31.01h	1379.69G	48.66h	17.07G	18m	0.83G	2m	1.26G	3m	2.73G	9m
14	476.48G	17.85h	765.83G	32.13h	1450.35G	50.8h	17.39G	19m	0.87G	3m	1.34G	5m	2.89G	8m
15	645.18G	21.12h	1263.13G	38.02h	2395.30G	62.61h	30.48G	25m	1.93G	4m	3.04G	7m	5.59G	12m

introduces additional deserialization overhead due to parsing the path conditions from the SMT-LIB2 format (the standard exporting format for mainstream SMT solvers), which is time-consuming [64]. LIBWP is generally not scalable as it can only successfully analyze three of 15 projects within the given time and memory budget. This is because exploring the data-flow paths among TPLs can easily lead to the path explosion problem. In contrast, LIBALCHEMY consumes no more than 70GB of memory to finish the analysis of the project MariaDB, a database system with more than 10 MLoC in terms of its application and dependent TPL. Compared with LIBPDG, LIBALCHEMY achieves a marginal improvement, i.e., 8.37% of memory reduction. Although LIBPDG requires searching more candidate paths in TPLs, it discards infeasible paths and only preserves feasible ones during searching.

LIBFREE does not consider the presence of TPLs, thus infeasible to detect TPL-related bugs. Thus, it is not surprising that it is the most efficient ( $1/0.71=1.29\times$  speedup over LIBALCHEMY) and consumes the least memory (50.91% of memory reduction over LIBALCHEMY) among all the settings. However, there are still three exceptions when analyzing the project vlc, binutils, and MariaDB. For example, the speedup of LIBALCHEMY over LIBFREE is  $3.71\times$  when analyzing

binutils. By runtime profiling, we notice that LIBFREE will waste time exploring many infeasible program without exploring the path conditions from library functions paths.

#### 6.4 RQ3: Overhead of Persisting Summaries

To evaluate the time and space overhead of persisting summaries, we compared our summary design (i.e., LIBALCHEMY) with the conventional path-sensitive precise summary (i.e., LIBCON). Specifically, to understand the effects of compact encoding in our first-layer summary, we measure how the time and space cost of persisting summaries vary with the increasing number of bug-checkers.

Table 5 shows the comparison results. Taking one bug checker for example, LIBALCHEMY persists two-layer summaries, taking up 17.90GB to 32.41GB of disk space, while LIBCON persists single-layer summaries, taking up 457.15GB to 688.58GB of disk space. The space reduction of LIBALCHEMY over LIBCON is around 94.98% to 96.18%. The time overhead reduction of LIBALCHEMY over LIBCON is around 97.05% to 98.07%. This result indicates that our two-layer summary design significantly reduces the time and space overhead compared with conventional path-sensitive summary design.

Moreover, with an increasing number of bug checkers, the space and time overhead reductions of LIBALCHEMY over LIBCON are

much more significant. Taking four bug checkers as examples, LIBCON takes up 1,379.69GB to 2,472.52GB of disk space, while LIBALCHEMY only takes up 19.80GB to 36.07GB of disk space. The space overhead reduction of LIBALCHEMY over LIBCON is around 98.41% to 98.65%, which is much higher than when checking only one bug. Thus, the effects of compact encoding by our first-layer summary would be significantly amplified with the increasing number of bug checkers.

## 7 RELATED WORK

**Program Dependence Graph.** Approaches to constructing program dependence graphs (PDGs) have evolved rapidly in recent decades. Initially, PDG only represented a program’s data and control-dependence within a single function [65]. As an extension, system dependence graph (SDG) introduces the call and return edges to bridge the dependence across procedures in a system [66–68]. Recently, code property graphs have been proposed to combine semantics and structural representation of the program for detecting various bugs [69, 70]. However, these approaches cannot represent a precise program dependence due to their weak assumptions on alias relations. To address this problem, many existing approaches adopted a “staged design”, invoking a flow-sensitive pointer analysis algorithm to obtain a precise PDG before doing client analyses [71, 72]. However, existing studies show the stage design introduces scalability issues and prevents a PDG with higher precision from being constructed [51, 73]. Recently, a PDG equipped with a “holistic design” achieves path-sensitive precision in a scalable way [51]. Similarly, our work is built on such a state-of-the-art PDG representation, to facilitate scalable and precise bug detection.

**Function Summaries.** Existing studies on function summaries can be grouped into three categories. The first category is designed for caching the analysis results in the compositional analysis under the context of the same run of analysis [24, 31, 33, 40–42]. For example, the IFDS [40] and IDE frameworks [33] compute the low-level function summaries, which can be understood as adding a rapid edge from the function entry to the function exit and uses them to accelerate the reachability analysis in the same running process. However, such summaries can only be used if a function is called multiple times in the same analysis, and there is not any easy way to persist them and use them independently for different analyses. The second category is designed for incremental analysis [43–47]. Such summaries essentially need to capture the dependency among the analysis units so that only those impacted by the changes are reanalyzed. For example, Do et al. [46] built their incremental analysis on top of the IFDS framework, and used the layers of data-flow propagation along calling relations as the function summaries to prioritize the reanalysis task. Although such summaries can be used in different runs of analysis, they are limited in reusing results from the previous run of analysis on the same program module. The third category can be used for reusing analysis results in different runs and across different modules [16, 34, 35, 48, 49]. Stubdroid [16] uses component-level analysis to generate function summary based on FlowDroid [74], a flow-sensitive work for taint analysis. However, they are not path-sensitive and, thus, fall short of precision. To achieve the precision of path sensitivity, some studies use path

constraints as function summaries [48, 49], but our experiments have shown such approaches to be costly.

**Disk-based Static Analysis System.** The techniques used by disk-based static analysis systems generally fall into three categories. The first category stores the control-flow graph (CFG) onto graph databases and describes the client analysis in a CFG in graph queries [75, 76]. Particularly, Weiss et al. [75] built the incorrect error propagation rules as graph patterns and sped up their queries via graph indexing techniques. The second category adapts existing static analysis algorithms into graph systems such that existing algorithms can directly benefit from the scalability improvement from these systems [77–79]. However, both existing graph databases and graph systems do not persist a PDG and lack the capability of handling path sensitivity, preventing the higher requirement of a practical bug detection solution. The third category stores the transformation rules that summarize the taint-related flows and uses them to detect the taint-style bugs in a flow-sensitive manner. Our work also follows the summary-based approach but integrates PDG and data-flow paths as summaries to achieve both path sensitivity and high scalability.

## 8 CONCLUSION

We presented a two-layer summary design that enables the static bug finder to detect bugs caused by the misuse of TPL functions in an effective, scalable, and precise manner. We implemented our idea as a tool named LIBALCHEMY and evaluated it systematically. The evaluation results show that LIBALCHEMY is a promising industrial-strength static bug-finding system.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. Rongxin Wu is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001), Natural Science Foundation of China (62272400), and the grant from Huawei. Charles Zhang is supported by ITS/440/18FP grant from the Innovation and Technology Commission and the donations from Microsoft and Huawei. Chengpeng Wang is the corresponding author.

## REFERENCES

- [1] The Open Web Application Security Project. OWASP Top 10. [https://owasp.org/www-pdf-archive/OWASP\\_Top\\_10\\_-\\_2013.pdf](https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf), 2013. [Online; accessed Jul-2023].
- [2] The Open Web Application Security Project. OWASP Top 10. [https://owasp.org/www-project-top-ten/2017/Top\\_10](https://owasp.org/www-project-top-ten/2017/Top_10), 2017. [Online; accessed Jul-2023].
- [3] The Open Web Application Security Project. OWASP Top 10. <https://owasp.org/Top10/>, 2021. [Online; accessed 10-Jan-2022].
- [4] Synopsys Inc. Black Duck Software Composition Analysis. <https://www.blackduckssoftware.com/>, 2022. [Online; accessed 7-Dec-2022].
- [5] Micro Focus Inc. HP Fortify Software Composition Analysis. <https://www.microfocus.com/en-us/cyberres/application-security/software-composition-analysis>, 2022. [Online; accessed 7-Dec-2022].
- [6] Snyk Limited. Snyk Software Composition Analysis. <https://snyk.io/series/open-source-security/software-composition-analysis-sca/>, 2022. [Online; accessed 7-Dec-2022].
- [7] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. Centris: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 860–872. IEEE, 2021. <https://doi.org/10.1109/ICSE43902.2021.00083>.
- [8] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. Moverly: A precise approach for modified vulnerable code clone discovery from modified open-source software components. In *31st USENIX*

- Security Symposium (USENIX Security 22)*, pages 3037–3053, 2022. <https://www.usenix.org/conference/usenixsecurity22/presentation/woo>.
- [9] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020. <https://www.usenix.org/conference/usenixsecurity20/presentation/xiao>.
  - [10] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707. IEEE, 2021. <https://doi.org/10.1109/ICSE43902.2021.00150>.
  - [11] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017. <https://doi.org/10.1109/SP.2017.62>.
  - [12] GitHub. A potential bug of NPD in opusfile. <https://github.com/xiphopusfile/issues/36>, 2022. [Online; accessed Jul-2023].
  - [13] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Daphne Yao, and Na Meng. Automatic detection of java cryptographic api misuses: Are we there yet. *IEEE Transactions on Software Engineering*, 2022. <https://doi.org/10.1109/TSE.2022.3150302>.
  - [14] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2018. <https://doi.org/10.1109/TSE.2018.2827384>.
  - [15] Peter Leo Gorski, Luigi Lo Iacono, Yasemin Acar, Sebastian Moeller, Christian Stransky, and Sascha Fahl. On the effect of security warnings on cryptographic api misuse. In *39th IEEE Symposium on Security and Privacy*, 2018. <https://www.usenix.org/conference/soups2018/presentation/gorski>.
  - [16] Steven Arzt and Eric Bodden. Stubbroid: Automatic inference of precise dataflow summaries for the android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 725–735. IEEE, 2016. <https://doi.org/10.1145/2884781.2884816>.
  - [17] Nomair A Naem and Ondřej Lhoták. Faster alias set analysis using summaries. In *International Conference on Compiler Construction*, pages 82–103. Springer, 2011. [https://doi.org/10.1007/978-3-642-19861-8\\_6](https://doi.org/10.1007/978-3-642-19861-8_6).
  - [18] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *Asian Symposium on Programming Languages and Systems*, pages 290–306. Springer, 2013. [https://doi.org/10.1007/978-3-319-03542-0\\_21](https://doi.org/10.1007/978-3-319-03542-0_21).
  - [19] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 553–566, 2015. <https://doi.org/10.1145/2676726.2676977>.
  - [20] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 789–801, 2016. <https://doi.org/10.1145/2914770.2837628>.
  - [21] Manu Sridharan, Shay Artzi, Marco Pistoi, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 1053–1068, 2011. <https://doi.org/10.1145/2048066.2048145>.
  - [22] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117, 2015. <https://doi.org/10.1145/2771783.2771803>.
  - [23] Li Li, Tegawendé F Bissyandé, Damien Oetou, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329, 2016. <https://doi.org/10.1145/2931037.2931044>.
  - [24] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240, 2012. <https://doi.org/10.1145/2382196.2382223>.
  - [25] IBM T.J. Watson Research Center. The T.J. Watson Libraries for Analysis (WALA). <https://github.com/wala/WALA>, 2022. [Online; accessed Nov-2022].
  - [26] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 93–104, 2014. <https://doi.org/10.1145/2666620.2666631>.
  - [27] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014. <https://doi.org/10.1145/2619091>.
  - [28] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. <https://doi.org/10.4230/LIPICs.ECOOP.2021.2>.
  - [29] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 198–209, 2020. <https://doi.org/10.1145/3377811.3380390>.
  - [30] Johannes Späth, Karim Ali, and Eric Bodden. Ide al: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017. <https://doi.org/10.1145/3133923>.
  - [31] Atanas Rountev and Barbara G Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *International Conference on Compiler Construction*, pages 20–36. Springer, 2001. [https://doi.org/10.1007/3-540-45306-7\\_3](https://doi.org/10.1007/3-540-45306-7_3).
  - [32] Atanas Rountev, Scott Kagan, and Thomas Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, pages 2–16. Springer, 2006. [https://doi.org/10.1007/11688839\\_2](https://doi.org/10.1007/11688839_2).
  - [33] Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, pages 53–68. Springer, 2008. [https://doi.org/10.1007/978-3-540-78791-4\\_4](https://doi.org/10.1007/978-3-540-78791-4_4).
  - [34] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–95, 2015. <https://doi.org/10.1145/2676726.2676997>.
  - [35] Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. Conditional dyck-cfl reachability analysis for complete and efficient library summarization. In *European Symposium on Programming*, pages 880–908. Springer, 2017. [https://doi.org/10.1007/978-3-662-54434-1\\_33](https://doi.org/10.1007/978-3-662-54434-1_33).
  - [36] John Toman and Dan Grossman. Taming the static analysis beast. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. <https://doi.org/10.4230/LIPICs.SNAPL.2017.18>.
  - [37] Synopsys Inc. Coverity Scan. <https://scan.coverity.com/>, 2022. [Online; accessed 7-Dec-2022].
  - [38] Ankush Das, Shuvendu K Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *International Conference on Computer Aided Verification*, pages 324–342. Springer, 2015. [https://doi.org/10.1007/978-3-319-21690-4\\_19](https://doi.org/10.1007/978-3-319-21690-4_19).
  - [39] Sam Blackshear and Shuvendu K Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 209–218, 2013. <https://doi.org/10.1145/2491956.2462188>.
  - [40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 49–61, 1995. <https://doi.org/10.1145/199448.199462>.
  - [41] Domagoj Babic and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 211–220. IEEE, 2008. <https://doi.org/10.1145/1368088.1368118>.
  - [42] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pages 567–577. ACM, 2011. <https://doi.org/10.1145/1993498.1993565>.
  - [43] Steven Arzt and Eric Bodden. Reviser: efficiently updating ide-/ifds-based dataflow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, pages 288–298, 2014. <https://doi.org/10.1145/2568225.2568243>.
  - [44] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 554–564, 2013. <https://doi.org/10.1145/2491411.2501854>.
  - [45] Lori L Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, 1989. <https://doi.org/10.1109/32.58766>.
  - [46] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '17*, pages 307–317. ACM, 2017. <https://doi.org/10.1145/3092703.3092705>.
  - [47] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient incremental static analysis using path abstraction. In *International Conference on Fundamental Approaches to Software Engineering*, pages 125–139. Springer, 2014. [https://doi.org/10.1007/978-3-642-54804-8\\_9](https://doi.org/10.1007/978-3-642-54804-8_9).
  - [48] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):16–es, 2007. <https://doi.org/10.1145/1232420.1232423>.
  - [49] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on*

- Principles of Programming Languages*, POPL '05, pages 351–363. ACM, 2005. <https://doi.org/10.1145/1040305.1040334>.
- [50] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 930–943, 2021. <https://doi.org/10.1145/3453483.3454086>.
- [51] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles 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*, pages 693–706, 2018. <https://doi.org/10.1145/3192366.3192418>.
- [52] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 480–491, 2007. <https://doi.org/10.1145/1250734.1250789>.
- [53] Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. Indexing the extended dyck-clf reachability for context-sensitive program analysis. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1438–1468, 2022. <https://doi.org/10.1145/3563339>.
- [54] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 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, pages 480–491. ACM, 2007. <https://doi.org/10.1145/1250734.1250789>.
- [55] Qingkai Shi and Charles Zhang. Pipelining bottom-up data flow analysis. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 835–847. ACM, 2020. <https://doi.org/10.1145/3377811.3380425>.
- [56] Leonardo De Moura and Nikolaj Bjørner. 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*, pages 337–340. Springer, 2008. [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24).
- [57] Linux manual page. dpkg(1). <https://man7.org/linux/man-pages/man1/dpkg.1.html>, 2022. [Online; accessed 12-Dec-2022].
- [58] Yum. Yum v3.4.3 documentation. <http://yum.baseurl.org/api/yum/>, 2022. [Online; accessed 12-Dec-2022].
- [59] All bug reports by libAlchemy. <https://github.com/ash1852/fusion-scan.github.io?tab=readme-ov-file>, 2022.
- [60] Synopsys. Coverity Scan. <https://scan.coverity.com/projects>, 2022. [Online; accessed Jul-2023].
- [61] Bug report denied by libXi. <https://gitlab.freedesktop.org/xorg/lib/libxi/-/issues/14>, 2022.
- [62] Bug report denied by transmission. <https://github.com/transmission/transmission/issues/3706>, 2022.
- [63] Bug report denied by MariaDB. <https://jira.mariadb.org/browse/MDEV-29891?filter=-2>.
- [64] David R Cok et al. The smt-lib2 language and tools: A tutorial. *Language c*, pages 2010–2011, 2011. <https://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf>.
- [65] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987. <https://doi.org/10.1145/24039.24041>.
- [66] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990. <https://doi.org/10.1145/77606.77608>.
- [67] Neil Walkinshaw, Marc Roper, and Murray Wood. The java system dependence graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64. IEEE, 2003. <https://doi.org/10.1109/SCAM.2003.1238031>.
- [68] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, 1999. <https://doi.org/10.1145/302405.302675>.
- [69] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014. <https://doi.org/10.1109/SP.2014.44>.
- [70] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015. <https://doi.org/10.1109/SP.2015.54>.
- [71] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction, CC '16*, pages 265–266. ACM, 2016. <https://doi.org/10.1145/2892208.2892235>.
- [72] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. *ACM SIGPLAN Notices*, 48(11):85–96, 2013. <https://doi.org/10.1145/2555670.2466483>.
- [73] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264, 2012. <https://doi.org/10.1145/2338965.2336784>.
- [74] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *AcM Sigplan Notices*, 49(6):259–269, 2014. <https://doi.org/10.1145/2666356.2594299>.
- [75] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. Database-backed program analysis for scalable error propagation. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 586–597. IEEE, 2015. <https://doi.org/10.1109/ICSE.2015.75>.
- [76] Dániel Lukács, Gergely Pongrácz, and Máté Tejfel. Are graph databases fast enough for static p4 code analysis? In *ICAI*, pages 213–223, 2020. <https://ceur-ws.org/Vol-2650/paper22.pdf>.
- [77] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News*, 45(1):389–404, 2017. <https://doi.org/10.1145/3093337.3037744>.
- [78] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qihong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019. <https://doi.org/10.1145/3302424.3303972>.
- [79] Zhiqiang Zuo, Yiyu Zhang, Qihong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Chianina: an evolving graph system for flow-and context-sensitive analyses of million lines of c code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 914–929, 2021. <https://doi.org/10.1145/3453483.3454085>.