Too Many Checkers, Too Little Time

***Abstract*—Conventional static bug finding techniques still suffer from scalability issues when used to check multiple types of vulnerabilities. We observe that this is mainly because they do not well utilize the homogeneous and the heterogeneous vulnerability properties. The former capture the commonality of different vulnerability types and, thus, can help avoid repeated computation. The latter allow developers to design vulnerability-specific optimization methods to boost the analysis efficiency. In this paper, we propose a theoretical framework, called CATAPULT, that is highly efficient to check a wide range of vulnerabilities at the same time. The high efficiency benefits from our novel observation that the data dependence relations in a program allow us to simultaneously make use of both the homogeneity and the heterogeneity. In order to facilitate the use of vulnerability properties and keep our framework extensible, we provide a domain-specific language to help developers write vulnerability checkers. Using the language, we have developed twenty checkers in CATAPULT. We have evaluated our approach using ten open-source software systems, nearly five million lines of code in total. The results demonstrate that CATAPULT is 20× and 11× faster on average than PINPOINT and CLANG STATIC ANALYZER, two representative techniques with industrial strength. The promising scaling effects are not achieved by sacrificing the effectiveness: we detected hundreds of real vulnerabilities, in which thirty-two have been confirmed by developers so far and three have been assigned CVE IDs due to their security impact.**

***Index Terms*—Static bug finding, sparse value-flow analysis, multiple checkers.**

I. INTRODUCTION

Numerous static analysis techniques have been proposed in the past decades to tackle a wide range of software vulnerabilities known as source-sink problems [1], [2]. Although some of these techniques are tailored for particular problems such as null pointer dereference[1] [4], [5] and memory leak[2] [1], [6]–[8], in practice, developers often use a framework-based approach [2], [9]–[12] that are capable of checking a variety of vulnerabilities. Our work is to address a relatively less investigated issue: when different vulnerability types are checked at the same time, the performance of the framework together with a collection of so-called vulnerability checkers, i.e., specific vulnerability detection methods, tends to have scalability issues with the growth of the number of checkers.

A common design of these frameworks consists of a core engine and a series of peripheral vulnerability checkers [13]–[15]. The core engine performs the main analysis task, exploring the control flow graph exhaustively by symbolic execution [16], data flow analysis [17], or abstract interpretation [18]. The checkers collect analysis results from the core engine for bug detection. These techniques are known to have performance problems [1], [6], [11], [19]. For example, it takes 6 to 11 hours for SATURN [9] and CALYSTO [10] to check only a single vulnerability type for programs of 685 thousand lines of code [10]. To run 35 default C/C++ checkers in an acceptable time budget, CLANG STATIC ANALYZER (CSA) [13] gives up much precision and recall, reporting more than 50% false positives and missing cross-file bugs. Even worse, since it is difficult to verify the absence of vulnerabilities on a path before the path is explored, the core engine cannot prune any control flow path to reduce the effort of program exploration, unless unsound assumptions are being made [13], [20]. Thus, the performance is hard to be improved notably in a sound manner.

Compared to the aforementioned techniques, sparse value-flow analysis (SVFA) [1], [2], [21], [22] does not blindly propagate values from a statement to all its successors on the control flow graph but directly propagates values to where they are used via data dependence. In addition, SVFA works in a demand-driven manner where we do not need to perform expensive computation on checker-unrelated data dependence. Owing to these features, SVFA is well-known to be more efficient when used to check a single source-sink problem [1], [2], [21], [22].

Nevertheless, existing SVFA designs are also not scalable in the face of multiple vulnerability checkers. For instance, it takes more than 70 hours for the most recent SVFA, PINPOINT [2], to run 20 checkers on a program of two million lines of code. On the one hand, as the number of checkers grows, there will be little checker-unrelated data dependence, and expensive computation eventually has to be carried out on all data dependence. As a result, the demand-driven feature of SVFA will not have any effect and the performance degrades. To illustrate, for the example in Figure 1, if we only check memory leak, we only need to traverse the data dependence graph from a [1], thereby avoiding unnecessary computation on the sub-graph rooted at the vertex *nil*. However, if we also check null pointer dereference, there will be no checker-unrelated data dependence and the whole graph needs to be traversed. On the other hand, to respect the principle of separation of concerns [23], existing SVFA techniques design and run different checkers separately [2], [21], [22], leading to a lot of repeated computation. In the example, we need to repeat exploring the graph from a for checking null pointer dereference and memory leak. This is because a will be null if the memory allocation fails, and a may cause memory leak if the memory allocation succeeds.

In this work, we propose a new design of SVFA that can mitigate the aforestated performance degradation and prevent the repeated computation by utilizing a variety of vulnerability properties. In the context of checking multiple vulnerability

[1]A null pointer dereference can be formulated as the value flow from a null pointer (source) to a dereference statement (sink) [3].

[2]A memory leak can be formulated as multiple value flows from a memory allocation statement (source) to memory release statements (sink) [1].

types, these properties can be classified into two categories, the homogeneous and the heterogeneous properties, both of which can be leveraged to improve scalability:

- **Homogeneity.** The homogeneous properties capture the common structure of different vulnerability types. Avoiding repetitive computation on the common information can significantly reduce the analysis workload.
- **Heterogeneity.** The heterogeneous properties (or checker specific properties) capture the specific structure of each vulnerability type. These properties can be leveraged to prune unnecessary program exploration in SVFA to mitigate the performance degradation.

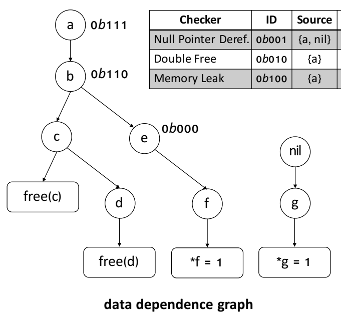
Our key insight is that the data dependence graph explored by SVFA allows us to simultaneously benefit from both the homogeneity and the heterogeneity. Homogeneity, or data dependence homogeneity (DD-homogeneity), means that different checkers abundantly share common data dependence chains (DDC) computed by the SVFA core engine. Such DD-homogeneity exists even between two checkers that check seemingly unrelated vulnerability types like memory leak and null pointer dereference. In our approach, to achieve high scalability, the DD-homogeneity is utilized at a fine-grain level where both complete DDCs and sub-DDCs can be reused across checkers. Such fine-grain DD-homogeneity is achieved via a theory in which a DDC is modeled as a context-free grammar. The grammar splits a DDC into sub-DDCs, each of which is labeled with a bit vector to record what checkers it contributes to. Such a few bits of memory usage enable the checkers to share common DDCs and guarantee that only compatible sub-DDCs that contribute to common checkers can be stitched together. In addition, the bit vector allows the core engine to be fully aware of the peripheral checkers. This makes it possible for the core engine to invoke queries on the checker-specific properties and leverage them to boost the performance.

As for the heterogeneity, we find that many checker-specific properties can be formulated as relations among program elements. Such relational heterogeneity is the key to reducing unnecessary work in SVFA. To benefit from the heterogeneity, we discover and employ a long-neglected feature of SVFA, which we refer to as "fast path-pruning". Generally speaking, it means that when tracking value flows along data dependence, we can stop immediately if a checker-specific property is violated. On the data dependence graph in Figure 1, the DDC from e can be pruned even we check both null pointer dereference and memory leak at once. This is because e depends on a non-null pointer b and cannot reach to any *free* statement. To facilitate such path pruning, we provide a domain-specific language for checker developers to specify various properties using extended first-order logic formulae. The core engine then automatically determines if the properties are violated with three different mechanisms.

We implement our framework, named CATAPULT,[3] on top of PINPOINT [2], a path-sensitive SVFA technique. Using our

[3]Our CATAPULT can throw one stone (SVFA core engine) to shoot multiple birds (vulnerability checkers).

```
int* a = malloc(sizeof(int));
if (!a) exit(1);
int* b = a;
if (...) {
    int* c = b;
    int* g = nil;
    if (...) { int* d = c; free(d); }
    else { free(c); *g = 1}
} else {
    int* e = b;
    int* f = e; *f = 1;
}
```



Checker	ID	Source	Sink
Null Pointer Deref.	0b001	{a, nil}	{*f=1,*g=1}
Double Free	0b010	{a}	{free(...)}
Memory Leak	0b100	{a}	{free(...)}

Fig. 1: Motivating example.

domain-specific language, we have developed twenty checkers, including all CSA's default checkers that can be formulated as source-sink problems [13]. In the evaluation, using ten popular industrial-sized software systems, we compare the scaling effects of CATAPULT with both the most recent SVFA, PINPOINT, and the state-of-the-art non-sparse bug-finding tool, CSA. We observe that CATAPULT can be 20× and 11× faster on average than PINPOINT and CSA, respectively. The sparse approach, PINPOINT, is even relatively slower than the non-sparse approach, CSA, because PINPOINT utilizes neither the homogeneity nor the heterogeneity while CSA utilizes the control-flow homogeneity (CF-homogeneity).[4] With a low false positive rate of 26.9%, CATAPULT detects many previously-unknown vulnerabilities, in which thirty-two have been confirmed by developers and three have been assigned CVE IDs due to their security impact.

In summary, our main contributions are three-fold:

- We propose a theoretical framework, CATAPULT, for checking multiple source-sink style vulnerabilities at the same time (Section III). To the best of our knowledge, CATAPULT is the first static bug finding framework that can simultaneously benefit from both the homogeneity and the heterogeneity of different vulnerability types.
- We implement CATAPULT as a prototype tool and designed a small language to facilitate the checker writing process. We have integrated twenty commonly-used checkers into our framework and provided three mechanisms to utilize checker-specific properties (Section IV).
- We evaluate the performance and the effectiveness of our approach. The results demonstrate that CATAPULT is more scalable than conventional approaches and, meanwhile, keeps high precision and high recall (Section V).

II. OVERVIEW

Assuming that we are checking null pointer dereference, memory leak, and double free, we now briefly explain our approach using the program and its data dependence graph in Figure 1.

[4]In terms of CF-homogeneity, we mean that the checkers share the common control flow information computed by the core engine, as discussed in the second paragraph of this section.

Homogeneity. In this example, checkers for the three vulnerability types share a lot of common data dependence: to check memory leak, we need to verify whether the DDCs from a to the two *free* statements cover all feasible program paths; to check double free, we need to verify whether the two DDCs can be feasible at the same time; and to check null pointer dereference, in addition to the DDC starting from the vertex *nil*, we need to verify whether the DDC from a to $*f$=1 can be feasible when a is a null pointer. We can observe that the checkers for memory leak and double free share the same DDCs, and the checker for null pointer dereference at least shares the data dependence from a to b with the other two.

Heterogeneity. Different vulnerability types have heterogeneous properties, which can be utilized to improve the performance of SVFA via its "fast path-pruning" feature.

Any value in a DDC for checking null pointer dereference must be a null pointer. In the example, we can query a simple analysis to know that b is not a null pointer because, otherwise, the program will exit at Line 2. Thus, when traversing the data dependence graph from a to check null pointer dereference, we can stop immediately at the vertex b.

Any value in a DDC for checking memory leak and double free must be reachable to a *free* statement [1], [6]. In this example, by querying a graph reachability indexing technique [24], it is easy to know that the vertex e is not reachable to any *free* statement on the data dependence graph. Thus, the DDC starting from e can be pruned for the two checkers.

Combining Homogeneity and Heterogeneity. In CATAPULT, we first assign the identities, $0b001$, $0b010$, and $0b100$, to the three checkers for null pointer dereference, double free, and memory leak, respectively. The vertex a on the data dependence graph is marked by the bit vector $0b111$, which is the bitwise disjunction of the three identities and implies that all the three checkers need to traverse the graph from it.

We then traverse the graph by a depth-first search. During the traversal, the bit vector makes the core engine fully aware of the peripheral checkers. Thus, we are able to query if the specific properties of each checker are violated. For example, when reaching the vertex b, because the current bit vector is $0b111$, we will invoke queries on the properties of all three checkers. In result, the lowest bit in the bit vector will flip because, as discussed before, the DDC from b will not contribute to checking null pointer dereference.

Similarly, when reaching the vertex e, because the current bit vector is $0b110$, we will only invoke queries on the specific properties of the memory-leak and the double-free checkers. Since e cannot reach any *free* statements, the bit vector is cleared. Thus, the depth-first search can be stopped immediately to avoid unnecessary program exploration.

For the DDCs we obtained by the graph traversal, we then solve their path conditions and checker-specific constraints to determine if some vulnerabilities exist at runtime.

Fine-Grain DD-Homogeneity and Relational Heterogeneity. The above example illustrates how we use the DD-homogeneity at a coarse-grain level, where we only reuse either a complete DDC between a source and a sink or a prefix of a complete DDC. To achieve high scalability, we argue that the DD-homogeneity should be utilized at a fine-grain level, where more sub-DDCs can be reused across different checkers. In addition, how to support the heterogeneous vulnerability properties in an extensible manner is still a problem. In the subsequent sections, we discuss these problems in detail.

III. APPROACH

The key design goal of our approach is to enable SVFA to benefit from both the homogeneity and the heterogeneity at the same time. In this section, we first introduce the notations used in the paper (Section III-A). With a context-free grammar model (Section III-B), we then explain how to make use of the fine-grain DD-homogeneity in two steps: reusing common sub-DDCs in a single vulnerability checker (Section III-C) and reusing common sub-DDCs across different checkers (Section III-D). Finally, we discuss the model of the heterogeneity (Section III-E) and present a domain-specific language to facilitate the checker writing process (Section III-F).

A. Preliminaries

Following the most recent SVFA technique, PINPOINT [2], we assume that a program consists of a series of pure functions. A pure function is a function where the return value only depends on its parameters, without observable side effects. With no loss of generality, we assume the code in each function is in SSA form, where every variable has only one definition [25]. The data dependence graph of such a program is defined as below.

Definition 1 (Data Dependence Graph)**.** A data dependence graph is a directed graph $\langle \Sigma, \rightsquigarrow \rangle$, where

- Σ is a set of vertices, each of which is denoted by $s|_l$, meaning that the variable l is defined or used in the statement s;
- $\rightsquigarrow$ is a set of edges, each of which represents a data dependence relation. $s_1|_{l_1} \rightsquigarrow s_2|_{l_2}$ means that $s_2|_{l_2}$ data depends on $s_1|_{l_1}$.

We say $\tau = \langle s_0|_{l_0}, s_1|_{l_1}, \cdots, s_n|_{l_n} \rangle$ is a data dependence chain (DDC) iff. the sequence represents a path on the data dependence graph. We use $\tau[i]$ to represent $s_i|_{l_i}$ if $0 \le i \le n$. Specifically, we use $\tau[-1]$ to represent the last element of τ.

A DDC τ_2 can be concatenated to another DDC τ_1, denoted as $\tau_1\tau_2$, iff. $\tau_1[-1] \rightsquigarrow \tau_2[0]$. The concatenation operation can be extended to two DDC sets, T_1 and T_2:

$$\mathrm{T}_1\mathrm{T}_2 \overset{\text{def}}{=} \{\tau_1\tau_2 : \tau_1 \in \mathrm{T}_1 \wedge \tau_2 \in \mathrm{T}_2 \wedge \tau_1[-1] \rightsquigarrow \tau_2[0]\}.$$

We use $\Sigma_{fp}, \Sigma_{ap}, \Sigma_{fr}$, Σ_{ar}, Σ_{src}, and Σ_{sink} to represent six special subsets of Σ:

- Σ_{fp} and Σ_{ap} represent the sets of formal and actual parameters, respectively.
- Σ_{fr} and Σ_{ar} represent the sets of formal and actual return values, respectively. We refer to the return value at a return statement as the formal return value and the return value at a call statement as the actual return value.

- Σ_{src} and Σ_{sink} represent the sets of source and sink vertices of a source-sink problem, respectively. They are specified for each checker by the checker developers.

B. A Context-Free Grammar Model

For each vulnerability type, we aim to find all target DDCs in the set $\mathrm{TG}(\Sigma_{src}, \Sigma_{sink})$, which is defined as below.

Definition 2 (Set of Target DDCs, $\mathrm{TG}(\Sigma_{src}, \Sigma_{sink})$)**.** Given a data dependence graph $\langle \Sigma, \rightsquigarrow \rangle$ and $\Sigma_{src}, \Sigma_{sink} \subseteq \Sigma$, a DDC $\tau \in \mathrm{TG}(\Sigma_{src}, \Sigma_{sink})$ iff. $\tau[0] \in \Sigma_{src} \wedge \tau[-1] \in \Sigma_{sink}$.

As shown in Figure 2, $\mathrm{TG}(\Sigma_{src}, \Sigma_{sink})$ can be modeled as a context-free grammar, which sets the foundation for our approach. Next, we introduce the terminal and non-terminal symbols in the grammar. The proofs of the productions are omitted due to the page limit.

A terminal symbol in the context-free grammar is a set of intra-procedural DDCs, which is defined below.

Definition 3 (Set of Intra-Procedural DDCs, $\mathrm{IP}(\Sigma_i, \Sigma_j)$)**.** Given a data dependence graph $\langle \Sigma, \rightsquigarrow \rangle$ and $\Sigma_i, \Sigma_j \subseteq \Sigma$, a DDC $\tau \in \mathrm{IP}(\Sigma_i, \Sigma_j)$ iff. $\tau[0] \in \Sigma_i \wedge \tau[-1] \in \Sigma_j$ and $\forall m \geq 0, n \geq 0$: $\tau[m]$ and $\tau[n]$ are in the same function.

As defined below, a same-level DDC starts and ends in the same function.[5]

Definition 4 (Set of Same-Level DDCs, $\mathrm{SL}(\Sigma_i, \Sigma_j)$)**.** Given a data dependence graph $\langle \Sigma, \rightsquigarrow \rangle$ and $\Sigma_i, \Sigma_j \subseteq \Sigma$, a DDC $\tau \in \mathrm{SL}(\Sigma_i, \Sigma_j)$ iff. $\tau[0] \in \Sigma_i \wedge \tau[-1] \in \Sigma_j$ and $\tau[0]$ is in the same function with $\tau[-1]$.

Example 1. In the following examples, we use s_i to represent the statement at Line i. The DDC $\langle s_1|_a, s_2|_a, s_5|_e, s_6|_e, s_2|_b, s_3|_b \rangle$ in Figure 3 is a same-level DDC because $s_1|_a$ and $s_3|_b$ are in the same function.

An output DDC, which is defined below, indicates that a checker-specific source escapes to its caller functions or upper-level caller functions.

Definition 5 (Set of Output DDCs, $\mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})$)**.** Given a data dependence graph $\langle \Sigma, \rightsquigarrow \rangle$, a DDC $\tau \in \mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})$ iff. $\tau[0] \in \Sigma_{src} \wedge \tau[-1] \in \Sigma_{fr}$ and $\tau[-1]$ is in the same function with $\tau[0]$ or in the (upper-level) callers of $\tau[0]$'s function.

Example 2. In Figure 3, suppose $s_5|_e \in \Sigma_{src}$, then both $\langle s_5|_e, s_6|_e \rangle$ and $\langle s_5|_e, s_6|_e, s_2|_b, s_3|_b \rangle$ are in $\mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})$, because the source flows to $s_6|_e$ and $s_3|_b$, in which $s_6|_e \in \Sigma_{fr}$ is in the same function with $s_5|_e$, and $s_3|_b \in \Sigma_{fr}$ is in the caller function.

An input DDC, as defined below, indicates that a formal parameter of a function f may flow to a sink in f or f's callees. A source in f's caller functions may propagate to the sink through the formal parameter.

[5]For simplicity, when we say $s_1|_{l_1}$ and $s_2|_{l_2}$ are in the same function, we mean that they are in the same function under the same calling context.

$$
\begin{aligned}
\mathrm{SL}(\Sigma_i, \Sigma_j) &\rightarrow \mathrm{IP}(\Sigma_i, \Sigma_j) && (1)\\
\mathrm{SL}(\Sigma_i, \Sigma_j) &\rightarrow \mathrm{IP}(\Sigma_i, \Sigma_{ap})\mathrm{SL}(\Sigma_{fp}, \Sigma_{fr})\mathrm{IP}(\Sigma_{ar}, \Sigma_j) && (2)\\
\mathrm{OUT}(\Sigma_{src}, \Sigma_{fr}) &\rightarrow \mathrm{SL}(\Sigma_{src}, \Sigma_{fr}) && (3)\\
\mathrm{OUT}(\Sigma_{src}, \Sigma_{fr}) &\rightarrow \mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})\mathrm{SL}(\Sigma_{ar}, \Sigma_{fr}) && (4)\\
\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink}) &\rightarrow \mathrm{SL}(\Sigma_{fp}, \Sigma_{sink}) && (5)\\
\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink}) &\rightarrow \mathrm{SL}(\Sigma_{fp}, \Sigma_{ap})\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink}) && (6)\\
\mathrm{TG}(\Sigma_{src}, \Sigma_{sink}) &\rightarrow \mathrm{SL}(\Sigma_{src}, \Sigma_{sink}) && (7)\\
\mathrm{TG}(\Sigma_{src}, \Sigma_{sink}) &\rightarrow \mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})\mathrm{SL}(\Sigma_{ar}, \Sigma_{sink}) && (8)\\
\mathrm{TG}(\Sigma_{src}, \Sigma_{sink}) &\rightarrow \mathrm{SL}(\Sigma_{src}, \Sigma_{ap})\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink}) && (9)\\
\mathrm{TG}(\Sigma_{src}, \Sigma_{sink}) &\rightarrow \mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})\mathrm{SL}(\Sigma_{ar}, \Sigma_{ap})\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink}) && (10)
\end{aligned}
$$

Fig. 2: A context-free grammar to produce the target DDC set.

```
1.  int* foo (int a) {              5.  int* bar (int e) {
2.      int b = bar(a);             6.      return e;
3.      return b;                   7.  }
4.  }
```

Fig. 3: Code for explaining symbols in the grammar. We use s_i to denote the statement at Line i. (Examples 1, 2, and 3)

Definition 6 (Set of Input DDCs, $\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink})$)**.** Given a data dependence graph $\langle \Sigma, \rightsquigarrow \rangle$, a DDC $\tau \in \mathrm{IN}(\Sigma_{fp}, \Sigma_{sink})$ iff. $\tau[0] \in \Sigma_{fp} \wedge \tau[-1] \in \Sigma_{sink}$ and $\tau[-1]$ is in the same function with $\tau[0]$ or in the (lower-level) callees of $\tau[0]$'s function.

Example 3. In Figure 3, suppose $s_6|_e \in \Sigma_{sink}$, then both $\langle s_5|_e, s_6|_e \rangle$ and $\langle s_1|_a, s_2|_a, s_5|_e, s_6|_e \rangle$ are in $\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink})$, because $s_5|_e \in \Sigma_{fp}$ is in the same function with the sink, and $s_1|_a \in \Sigma_{fp}$ is in the callee function.

C. Checking a Single Vulnerability Type

The context-free grammar implies that there is a Turing machine that can enumerate all the ways in which we can compute the target DDC set for a single vulnerability type [26]. For example, $\mathrm{IP}(\Sigma_{src}, \Sigma_{ap})\mathrm{IP}(\Sigma_{fp}, \Sigma_{sink})$ is such a way in which $\mathrm{TG}(\Sigma_{src}, \Sigma_{sink})$ can be produced by the set-concatenation operation defined in Section III-A.

Instead of giving a sophisticated formal representation of the Turing machine, we use Example 4 to illustrate the approach to computing the target DDC set. Our approach is a bottom-up compositional analysis, in which callees are analyzed before callers and the behaviors of a function are summarized as function summaries. In this manner, when analyzing a function, for each call statement in the function, we can reuse the callee's summaries to infer its semantics without analyzing the callee again. According to the context-free grammar, it is sufficient to generate three kinds of function summaries for DDCs in $\mathrm{SL}(\Sigma_{fp}, \Sigma_{fr})$, $\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink})$, and $\mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})$. The sufficiency is described as the following theorem and the proof is omitted due to the page limit.

Theorem 1 (Summary Sufficiency)**.** Any target DDC can be written as the concatenation of (1) a function's intra-procedural DDCs, and (2) DDCs in $\mathrm{SL}(\Sigma_{fp}, \Sigma_{fr})$, $\mathrm{IN}(\Sigma_{fp}, \Sigma_{sink})$, and $\mathrm{OUT}(\Sigma_{src}, \Sigma_{fr})$ from its callees.

```
1.  void* xmalloc () {
2.      void* p = malloc(...);
3.      return p;
4.  }
5.  void xfree (void* u) {
6.      free(u);
7.  }
8.  void main () {
9.      void* a = xmalloc();
10.     xfree(a);
11.     if (...)
12.         free(a);
13.     return;
14. }
```

Fig. 4: Code for illustrating our algorithm. We use s_i to denote the statement at Line i. (Example 4)

(a) Steps for producing the summary for xmalloc.		
Step	**DDC**	**Rule**
1	$\underbrace{\langle s_2\vert_p\rangle}_{\text{IP}(\Sigma_{src},\Sigma_{src})}$	Definition (3)
2	$\underbrace{\langle s_2\vert_p, s_3\vert_p\rangle}_{\text{IP}(\Sigma_{src},\Sigma_{fr})}$	Definition (3)
3	$\underbrace{\langle s_2\vert_p, s_3\vert_p\rangle}_{\text{SL}(\Sigma_{src},\Sigma_{fr})}$	Production (1)
4	$\underbrace{\langle s_2\vert_p, s_3\vert_p\rangle}_{\text{OUT}(\Sigma_{src},\Sigma_{fr})}$	Production (3)
(b) Steps for producing the target DDC.		
Step	**DDC**	**Rule**
1	$\underbrace{\langle s_2\vert_p, s_3\vert_p\rangle}_{\text{OUT}(\Sigma_{src},\Sigma_{fr})}$	Definition (5)
2	$\underbrace{\langle s_2\vert_p, s_3\vert_p\rangle}_{\text{OUT}(\Sigma_{src},\Sigma_{fr})}\underbrace{\langle s_9\vert_a, s_{10}\vert_a\rangle}_{\text{IP}(\Sigma_{ar},\Sigma_{ap})}$	Definition (3)
3	$\underbrace{\langle s_2\vert_p, s_3\vert_p\rangle}_{\text{OUT}(\Sigma_{src},\Sigma_{fr})}\underbrace{\langle s_9\vert_a, s_{10}\vert_a\rangle}_{\text{SL}(\Sigma_{ar},\Sigma_{ap})}$	Production (1)
4	$\underbrace{\langle s_2\vert_p, s_3\vert_p\rangle}_{\text{OUT}(\Sigma_{src},\Sigma_{fr})}\underbrace{\langle s_9\vert_a, s_{10}\vert_a\rangle}_{\text{SL}(\Sigma_{ar},\Sigma_{ap})}\underbrace{\langle s_5\vert_u, s_6\vert_u\rangle}_{\text{IN}(\Sigma_{fp},\Sigma_{sink})}$	Definition (6)
5	$\underbrace{\langle s_2\vert_p, s_3\vert_p, s_9\vert_a, s_{10}\vert_a, s_5\vert_u, s_6\vert_u\rangle}_{\text{TG}(\Sigma_{src},\Sigma_{sink})}$	Production (10)

Fig. 5: Graph traversal guided by the grammar. (Example 4)

The following example illustrates how the context-free grammar guides the generation of the target DDCs.

Example 4. For the program in Figure 4, we perform a bottom-up static analysis, which analyzes the functions `xmalloc` and `xfree` before the function `main`. This analysis aims to find DDCs for checking memory leak. As the previous work [1], [6], $\Sigma_{src} = \{s_2|_p\}$ and $\Sigma_{sink} = \{s_6|_u, s_{12}|_a\}$. Now, we illustrate how the context-free grammar guides us to get the target DDCs, $\langle s_2|_p, s_3|_p, s_9|_a, s_{10}|_a, s_5|_u, s_6|_u\rangle$ and $\langle s_2|_p, s_3|_p, s_9|_a, s_{12}|_a\rangle$.

1) The bottom function `xmalloc` is analyzed by a depth-first search on the data dependence graph from the source vertex, $s_2|_p$. The search steps are illustrated in Figure 5(a), where each line means that the current DDC belongs to a DDC set according to a rule. The result implies that we need to generate a summary for the DDC $\langle s_2|_p, s_3|_p\rangle$, because it belongs to OUT$(\Sigma_{src}, \Sigma_{fr})$.
2) The bottom function `xfree` is analyzed by a depth-first search from the formal parameter, $s_5|_u$. With a similar process, we generate a summary for the DDC $\langle s_5|_u, s_6|_u\rangle \in$ IN$(\Sigma_{fp}, \Sigma_{sink})$.
3) The function `main` is analyzed. At Line 9, it calls the function `xmalloc`, which has a summary $\langle s_2|_p, s_3|_p\rangle \in$ OUT$(\Sigma_{src}, \Sigma_{fr})$. Thus, Productions (8) and (10), which start with OUT$(\Sigma_{src}, \Sigma_{fr})$, can be used to generate the target DDCs. The steps in Figure 5(b) illustrate how we produce the first target DDC. With a similar process, we can find the second target DDC.

D. *Checking Multiple Vulnerability Types*

As discussed before, the context-free grammar implies that TG$(\Sigma_{src}, \Sigma_{sink})$ can be computed by concatenating different DDC sets. As different concatenation methods will lead to different results, we now extend the original concatenation methods as follows to check multiple vulnerability types. It is worth noting that this extension not only enables different checkers to share common (sub-) DDCs, but also allows the SVFA core engine to be aware of the checkers. Such checker-awareness makes it possible to benefit from heterogeneous vulnerability properties as illustrated in Section II.

First, a unique integer in $\mathbb{ID} = \{2^k : k \geq 0\}$ is assigned to each vulnerability type as its identity. Also, an integer $\alpha = \alpha_0|\alpha_1|\cdots|\alpha_n$ is assigned to each DDC τ, denoted as τ^α. Here, $\alpha_k \in \mathbb{ID}$ and $|$ is the *bitwise or* operator. τ^α indicates what vulnerability types the DDC τ is built for. The following example explains the notation τ^α.

Example 5. Suppose we check three vulnerability types, which are associated with the following source-sink pairs, respectively: $(\Sigma^1_{src}, \Sigma^1_{sink}), (\Sigma^2_{src}, \Sigma^2_{sink})$, and $(\Sigma^3_{src}, \Sigma^3_{sink})$. The three vulnerability types are assigned identities $0b001$, $0b010$, and $0b100$, which are the binary representations of 2^0, 2^1, and 2^2, respectively. Then τ^{0b011} means that τ can be used to build DDCs in both TG$(\Sigma^1_{src}, \Sigma^1_{sink})$ and TG$(\Sigma^2_{src}, \Sigma^2_{sink})$ because $0b011 = 0b001 \mid 0b010$.

Then, we redefine the concatenation operation of two DDCs and that of two DDC sets as follows:

$$\tau_1^\alpha \tau_2^\beta \overset{\textbf{def}}{=} (\tau_1\tau_2)^{\alpha\&\beta},$$

$$\mathrm{T}_1\mathrm{T}_2 \overset{\textbf{def}}{=} \{(\tau_1\tau_2)^{\alpha\&\beta} : \tau_1^\alpha \in \mathrm{T}_1 \wedge \tau_2^\beta \in \mathrm{T}_2 \wedge \alpha\&\beta \neq 0 \wedge \tau_1[-1] \rightsquigarrow \tau_2[0]\},$$

Intuitively, the extended concatenation means that if two DDCs contribute to two sets of checkers, respectively, then their concatenation only contributes to the checkers in the intersection of the two sets. We provide an example as below.

Example 6. Following Example 5, suppose we have two DDCs, τ_1^{0b011} and τ_2^{0b101}, then their concatenation $(\tau_1\tau_2)^{0b011\&0b101} = (\tau_1\tau_2)^{0b001}$ means that $\tau_1\tau_2$ can only be used to assemble DDCs in TG$(\Sigma^1_{src}, \Sigma^1_{sink})$.

Using the code in Figure 4, we illustrate how to check multiple vulnerability types with the new concatenation operations.

Example 7. Suppose we need to check null pointer dereference, double free, and memory leak in the code shown in Figure 4. As illustrated in Section II, all three vulnerability types regard the return value of `malloc`, i.e., $s_2|_p$, as the source. The sinks of memory leak and double free are instructions calling `free()` and the sink of null pointer dereference is pointer-dereference instructions. The steps for checking them are listed as below.

1) We assign $0b001$, $0b010$, and $0b100$ to null pointer dereference, double free, and memory leak as their type identities, respectively.
2) The bottom functions `xmalloc` and `xfree` are analyzed in the same way as in Example 4, except that each of the DDCs is marked by a bit vector. That is, we generate $\langle s_2|_p, s_3|_p\rangle^{0b111} \in \text{OUT}(\Sigma_{src}, \Sigma_{fr})$ and $\langle s_5|_u, s_6|_u\rangle^{0b110} \in \text{IN}(\Sigma_{fp}, \Sigma_{sink})$ as the summaries of `xmalloc` and `xfree`, respectively. The fist DDC is marked by $0b111$ because all the three vulnerability types regard $s_2|_p$ as the source. The second is marked by $0b110$ because null pointer dereference does not regard $s_6|_u$ as the sink while the other two types do.
3) The function `main` is analyzed in the same way as in Example 4 except that the new concatenation operation is adopted. For example, the first target DDC can be computed as:

$$\langle s_2|_p, s_3|_p\rangle^{0b111}\langle s_9|_a, s_{10}|_a\rangle^{0b111}\langle s_5|_u, s_6|_u\rangle^{0b110} = \langle s_2|_p, s_3|_p, s_9|_a, s_{10}|_a, s_5|_u, s_6|_u\rangle^{0b110}$$

Because the target DDC is marked by $0b110$, it contributes to checking both memory leak and double free.

Comparing Example 4 with Example 7, it is easy to find that except for the bitwise operations, although the latter checks three vulnerability types while the former only checks one, their whole processes are the same. Intuitively speaking, although we have three vulnerability types to check in the latter example, they share the same source and we only traverse the data dependence graph once from the source rather than repeating the traversal for each of the three vulnerability types. Thus, the time costs of the two examples are also similar. In an extreme case, if the three vulnerability types do not share any (sub-) DDCs, the time cost will be similar to that of running the checkers individually. This is because we have to traverse the data dependence graph from each of the sources separately. To sum up, we have the following theorem to guarantee the performance improvement over the method of running checkers individually.

Theorem 2 (Bound of the Performance Improvement)**.** Suppose we have N vulnerability types as our targets. If we find DDCs for them individually, the checking processes take time $t_1, t_2, \cdots, t_N$, respectively. Then if we check them together and ignore the negligible overhead of bitwise operations, the time cost must be in the interval $[\max_{i=1}^{N}\{t_i\}, \text{sum}_{i=1}^{N}\{t_i\}]$.

E. Relational Heterogeneity

The context-free grammar model uses a bit vector to make the SVFA core engine fully aware of what checkers a DDC contributes to. Thus, the core engine can proactively invoke queries on the specific properties of the checkers indicated by the bit vector. In this manner, we can enjoy the "fast path-pruning" feature of SVFA to prune unnecessary DDCs in time.

Given a data dependence graph $\langle \Sigma, \rightsquigarrow \rangle$, we find that a checker-specific property, P, usually can be modeled as a relation among the vertices on the graph, i.e., $P \subseteq \Sigma^n (n \geq 2)$. We define three commonly used relations as below, but note that our approach is extensible for any relational property.

- $\mapsto$ and $\not\mapsto$ represent the points-to relations: $a \mapsto b$ means that the pointer a may point to an object b, and $a \not\mapsto b$ means the opposite. For example, when checking the use of uninitialized memory, we can specify the relation $* \mapsto$ *undef*, meaning that any pointer on the DDC for the checker may point to uninitialized memory.
- $\rightarrow$ and $\not\rightarrow$ represent the reachability relations: $a \rightarrow b$ means that a is reachable to b on the data dependence graph, and $a \not\rightarrow b$ means the opposite. For some vulnerability types, developers may require the DDCs must or must not go through a specific vertex on the data dependence graph [27]. In such a scenario, the two relations are indispensable.
- Relations used in a conventional first-order logic formula, e.g., $>$, $<$, $=$, and so on. For example, we can specify the relation $* > 2$ to check file-descriptor leak for C/C++ programs. This property means that any file descriptor greater than 2 should be released.[6]

We provide three mechanisms to support the queries on these relational checker-specific properties:

- We adopt a flow- and context-insensitive unification-based pointer analysis to solve the points-to relations [28], [29]. The analysis, albeit imprecise, is ultra-fast and can soundly answer points-to queries with $O(1)$ time-complexity.
- We perform a graph reachability indexing method called PathTree [24] on the global data dependence graph, so that we can quickly determine if a vertex is reachable to the other with $O(\log^2 |\Sigma|)$ time-complexity. In our experience, this method reduces a lot of search space.
- We invoke an SMT solver [30] to solve the DDC's path conditions and the checker-specific conditions. In CATAPULT, we do not wait until a DDC is completely built and then invoke the SMT solver. This is because the constraint might be very complex then. Instead, when performing the depth-first search on the data dependence graph, if the next vertex to visit is in a different function, we will invoke the SMT solver to test the DDC's feasibility for each checker. Intuitively, this can avoid unnecessary inter-procedural computations if the constraint-solving results imply that the DDC will not be feasible after it enters into another function.

F. A Domain-Specific Language

To facilitate the usage of the relational heterogeneity, we design a domain-specific language to help developers write checkers. We do not seek to argue that our language is better

[6]A negative descriptor is invalid, and the descriptors 0, 1, and 2 represent the standard input, output, and error channels, respectively. We do not need to release them.

Checker	C ::= D; P; V
Variable Def.	D ::= S; O;
Specification	S ::= **source** = E; **sink** = E
Others	O ::= *var* = E; O | ε
Expression	E ::= $*i* **in** *regexp* | E **or** E
Properties	P ::= *formula*; P | ε
Vulnerability	V ::= **vuln** = { DDCs=(DDC1, DDC2, ...) | *cond* };
var	::= a string representing a variable name
regexp	::= a regular expression of program statements
i	::= a number standing for the *i*th operand; 0 means the return value
formula	::= a first-order logic formula with extended relational operators, e.g., ↦, ⟶, etc.
cond	::= conditions on which the tuple of DDCs, i.e., (DDC1, DDC2, ...), can cause a vulnerability

Fig. 6: A brief grammar of the domain-specific language.

Example: A Simple Memory-Leak Checker

```
1.  source = $0 in .*=malloc(.*); // A source is a heap pointer
2.  sink = $1 in free(.*); // A sink is where the pointer is "freed"
3.  arith = $0 in .*=.*[+-\*/].*; // The ret value of an arithmetic operation
4.  * != 0; // A pointer equal to 0 cannot cause memory leak
5.  arith !⟶ *; // Do not go through an arithmetic operation before it is "freed"
6.  source ⟶ *; * ⟶ sink; // Two default reachability relations
7.  // A memory leak occurs only when DDCs starting with the same source do
8.  // not cover all possible program paths. (d1, ...): a tuple with unknown length;
9.  // or/not: logic operators; sat: satisfiable; pc: path condition
10. vuln = { d=(d1, ...) | forall(di in d: di[0]==d1[0]) and sat(not(or(di in d: pc(di)))) };
```

A Program Example and Its Data Dependence Graph

```
1.  // allocate an integer array
2.  int* a=malloc(sizeof(int)*100);
3.  if (...) { int* b=a+1; *b=1; free(b); }
4.  else if (...) { int* c = a; int* d = c; *d=1; }
5.  else if (...) { free(a); }
6.  else { free(a); }
```

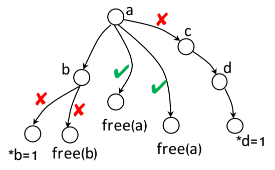


Fig. 7: Checker examples. The reserved key words are bold.

than the existing ones [27], [31]–[33] as discussed in Section VI. However, the language is suitable and extensible for our new SVFA design, as it can succinctly describe various relational checker-specific properties. Thus, the language allows CATAPULT to be extensible for more checkers and, meanwhile, remain highly efficient when running multiple checkers.

Due to the page limit, we only show a brief grammar of the language in Figure 6 and a checker example in Figure 7. As shown by the grammar, a checker written in the language consists of three parts: D, P, and V. The first part, D, defines the variables that will be used. The variable **source** and the variable **sink** must be defined, as they describe the basic specification of a source-sink style vulnerability. The second part, P, specifies the checker-specific properties using the variables defined before. Since a vulnerability, such as double free and memory leak, needs to be checked using multiple DDCs between the source and the sink vertices, the third part, V, describes the conditions on which a tuple of discovered DDCs can cause a vulnerability.

Example 8. In Figure 7, we illustrate a memory-leak checker written in the language. It specifies the property $* \neq 0$ because a pointer equal to zero cannot cause any memory leak. This constraint will be used together with the path conditions of the DDCs at the time of constraint solving. We also specify a property *arith* $\not\rightarrow *$, because a pointer cannot be successfully "freed" after an arithmetic operation in common cases. As shown in the program example and its data dependence graph, during the graph traversal, the two left-most DDCs are pruned based on the property *arith* $\not\rightarrow *$ and the right-most DDC is pruned based on the property $* \rightarrow$ *sink*. For this checker, we need to test the combination of the discovered DDCs between sources and sinks to determine if a memory leak can happen. A memory leak happens only when DDCs starting with the same memory allocation site, e.g., the two DDCs from a to *free*(a) on the data dependence graph, do not cover all feasible program paths. This requirement is described at Line 10.

IV. IMPLEMENTATION

CATAPULT is implemented on top of PINPOINT [2], the most recent SVFA technique that resolves pointer relations and provides path-sensitive data dependence graph. Like PINPOINT, we first compile C/C++ programs to LLVM bitcode [34], on which our analysis is performed. In our analysis, we use Z3 [30], an SMT solver, to solve constraints.

In CATAPULT, using our domain specific language, we have implemented twenty checkers according to the checker classification in CSA [13]. These checkers include all CSA's default checkers that can be formulated as source-sink problems. We briefly introduce these checkers in Table I. More details can be found on the homepage of CSA [13].

Our implementation is soundy [35], which means that it handles most language features in a sound manner while we also make some well-identified unsound choices following the previous work [1], [6], [9], [10]. In our implementation, we use class hierarchy analysis to resolve virtual functions [36]. We unroll each cycle twice on both the call graph and the control flow graph. We assume that distinct function parameters do not alias each other [3]. Following SATURN [9], a representative static bug detection tool, CATAPULT does not model inline assembly, function pointers, or library functions such as std::vector from the C++ standard template library.

V. EVALUATION

In this section, we present our systematic evaluation of CATAPULT's scaling effects by comparing it with two recent techniques for static bug finding. As a new design of SVFA, CATAPULT was first compared with PINPOINT [2], the most recent and only SVFA that can achieve path-sensitivity with high scalability. In addition, we also compared CATAPULT with a non-sparse approach, CSA [13], an industrial-strength bug detector. We also tried to compare CATAPULT with other tools like Saturn [9] and Calysto [10]. However, they are either unavailable or not runnable on the experimental environment we are able to set up. Although our main contribution is to

TABLE I: Checkers In CATAPULT

ID	Checkers	Brief Description
1	core.CallAndMessage	Check for uninitialized arguments and null function pointers
2	core.DivideByZero[†]	Check for division by zero
3	core.NonNullParamChecker	Check for null passed to function parameters marked with nonnull
4	core.NullDereference	Check for null pointer dereference
5	core.StackAddressEscape	Check that addresses of stack memory do not escape the function
6	core.UndefinedBinaryOperatorResult	Check for the undefined results of binary operations
7	core.VLASize (Variable-Length Array)	Check for declaration of VLA of undefined or zero size
8	core.uninitialized.ArraySubscript	Check for uninitialized values used as array subscripts
9	core.uninitialized.Assign	Check for assigning uninitialized values
10	core.uninitialized.Branch	Check for uninitialized values used as branch conditions
11	core.uninitialized.CapturedBlockVariable	Check for blocks that capture uninitialized values
12	core.uninitialized.UndefReturn	Check for uninitialized values being returned to callers
13	cplusplus.NewDelete	Check for C++ use-after-free
14	cplusplus.NewDeleteLeaks	Check for C++ memory leaks
15	unix.Malloc	Check for C memory leaks, double-free, and use-after-free
16	unix.MismatchedDeallocator	Check for mismatched deallocators, e.g., new and free()
17	unix.cstring.NullArg	Check for null pointers being passed to C string functions like strlen
18	alpha.core.CallAndMessageUnInitRefArg	Check for uninitialized arguments in function arguments and ObjC message expressions
19	alpha.unix.SimpleStream	Check for misuses of C stream APIs, e.g., an opened file is never closed
20	alpha.unix.Stream	Check stream handling functions, e.g., using a null file handle in fseek

[†] It is molded as a vulnerability with the same source and sink such as a in $c = b/a$.

TABLE II: Subjects for Evaluation

ID	Programs	Size (KLoC)
1	Shadowsocks	32
2	WebAssembly	75
3	Transmission	88
4	Redis	101
5	ImageMagick	358
6	Python	434
7	GlusterFS	481
8	LibICU	537
9	OpenSSL	791
10	MySQL	2,030

TABLE III: Comparison with PINPOINT

ID	PINPOINT (min)	CATAPULT[†] (min)	CATAPULT[‡] (min)
1	34.5	5.3 (7×)	2.3 (15×)
2	84.0	10.9 (8×)	3.5 (24×)
3	142.8	17.2 (8×)	14.3 (10×)
4	1025.4	82.7 (12×)	28.5 (36×)
5	3110.9	312.8 (10×)	125.1 (25×)
6	677.3	95.4 (7×)	26.5 (26×)
7	245.7	41.7 (6×)	24.5 (10×)
8	1272.7	194.3 (7×)	88.3 (14×)
9	1483.2	147.9(10×)	44.8 (33×)
10	4280.1	535.0 (8×)	266.9 (16×)
Total	12356.6	1443.2 (9×)	624.7 (20×)

[†] CATAPULT only utilizing the homogeneity.
[‡] CATAPULT utilizing both the homogeneity and the heterogeneity.

improve the scalability for multiple checkers, we also investigated CATAPULT's effectiveness of finding real vulnerabilities to confirm that the promising scaling effect of CATAPULT is not achieved by sacrificing its bug finding capability.

As shown in Table II, our evaluation subjects are ten popular open-source C/C++ projects. All of them are widely used as benchmarks in the literature. Their sizes range from 32 KLoC to 2 MLoC with nearly 5 MLoC in total.

All experiments were performed on a server with two Intel© Xeon© CPU E5-2698 v4 @ 2.20GHz (each has 20 cores) and 256GB RAM running Ubuntu-16.04.

A. *Comparing with* PINPOINT

We first compared CATAPULT with the most recent SVFA, PINPOINT [2], from two aspects. First, PINPOINT currently runs different checkers separately. Thus, in the first experiment, we ran checkers together in CATAPULT without utilizing various checker-specific properties. In this manner, we can study the significance of the homogeneity among different vulnerability types. Second, to further evaluate the significance of the heterogeneity, in the second experiment, CATAPULT was enabled to benefit from the checker-specific properties.

In the experiments, we compiled every project to a single LLVM bitcode file that links code from all compilation units. We then ran the aforestated two experiments with the twenty checkers listed in Table I. The comparison results are shown in Table III. The second column lists the total time cost of running the checkers one by one in PINPOINT. The third column demonstrates the time cost of CATAPULT, but without utilizing the checker-specific properties. The fourth column records the time cost of CATAPULT that makes full use of the homogeneity and the heterogeneity.

Comparing the results in the second and the third columns, CATAPULT is roughly 9× faster than PINPOINT when the checker-specific properties are not utilized. This performance improvement verifies our observation on the DD-homogeneity. Apparently, the degree of the performance improvement is related to the checker types: the more the common DDCs, the more the improvement we can achieve. Since CATAPULT includes all CSA's default checkers that can be modeled as source-sink problems, we argue that the improvement is common and significant in practice. Theorem 2 also guarantees that the performance of our approach cannot be worse than that of running checkers individually.

Despite the above-mentioned improvement, the performance of CATAPULT using only the homogeneity is still not quite satisfactory. For example, for MYSQL, a project with 2 million LoC, it still takes about 9 hours for CATAPULT to

TABLE IV: Comparison with CSA

ID	CSA (min)	CATAPULT[†] (min)	CATAPULT[‡] (min)
1	5.2	2.8 (2×)	1.0 (5×)
2	18.2	2.2 (8×)	1.3 (14×)
3	6.6	1.9 (3×)	1.1 (6×)
4	11.7	3.3 (4×)	1.9 (6×)
5	19.0	8.2 (2×)	3.8 (5×)
6	40.3	9.9 (4×)	6.3 (6×)
7	94.9	10.4 (9×)	1.1 (86×)
8	29.0	5.5 (5×)	2.9 (10×)
9	14.3	6.7 (2×)	1.8 (8×)
10	240.9	64.4 (4×)	21.2 (11×)
Total	480.1	115.3 (4×)	42.4 (11×)

† CATAPULT only utilizing the homogeneity.

‡ CATAPULT utilizing both the homogeneity and the heterogeneity.

TABLE V: Effectiveness

ID	CSA		CATAPULT/PINPOINT	
	# Reports	# FP	# Reports	# FP
1	13	9	9	0
2	12	4	10	2
3	30	17	24	2
4	48	30	39	5
5	692*	89	26	8
6	55	31	48	7
7	544*	67	59	22
8	345*	32	161	31
9	109	71	48	15
10	742*	42	245	88
FP Rate	58.7%		26.9%	

* We inspected one hundred randomly-sampled bug reports.

finish the analysis in the first experiment. This is very close to the upper bound of the acceptable time budget in industry, which is 10 hours [37], [38]. As shown by the results in the fourth column, the performance of CATAPULT is further improved when the heterogeneity is utilized, about 20× faster than PINPOINT on average. As a typical example, CATAPULT can finish checking MYSQL in 5 hours, which is aligned with the industrial requirement of checking millions of lines of code in 5 - 10 hours [37], [38].

To sum up, in order to build a scalable vulnerability detector, both the homogeneity and the heterogeneity of different vulnerability types should be well utilized.

B. Comparing with CSA

We also compared CATAPULT with CSA [13], which runs checkers together based on the CF-homogeneity. As discussed in Section I, techniques like CSA cannot benefit from the heterogeneity because the effort of program exploration cannot be reduced by pruning paths. As before, in the comparison, CATAPULT was run in two modes: with and without utilizing the heterogeneity.

For fair comparison, we chose CSA 5.0 [13] as our comparison target because of two reasons. First, it was the latest version of CSA at the time of our study. Second, it provides two methods of constraint solving: an imprecise range-based solver and a fully functional SMT solver, Z3 [30], that is also used in CATAPULT. This makes it possible to configure both of them to use Z3 in the experiments. In this manner, we can exclude the interference of the solvers when comparing their scalability. Like most commercial tools, CSA only supports compilation-unit level checking. For fairness, in our experiments, CATAPULT was also configured to check compilation units separately. Without loss of accuracy, for each project, we invoke fifteen threads to check different compilation units in parallel. Note that this is why the time cost shown in Table IV is much less than that in Table III.

As shown by the second and third columns of Table IV, If the heterogeneity is not utilized, the performance improved will be 4× on average. This 4× improvement demonstrates that the DD-homogeneity is more effective that the CF-homogeneity, because of the inherent sparseness of data dependence relations.

Comparing the results in the second and fourth columns of Table IV, we can observe that CATAPULT is up to 86× faster and 11× faster on average when both the homogeneity and the heterogeneity are utilized. This result confirms the significance and necessity of utilizing the heterogeneity via the "fast path-pruning" feature of SVFA.

C. Usefulness in Practice

In order to confirm that the promising scaling effects are not achieved by sacrificing the bug finding capability, we also manually inspected the warnings reported by CATAPULT, PINPOINT, and CSA. Table V presents the number of reported warnings as well as the number and the rate of false positives (FP). Since CATAPULT is built on top of PINPOINT and we do not compromise its precision and recall, their reported warnings are the same. For CSA, because it reported too many warnings, we could not afford the time to manually inspect all of them (validating each warning can take tens of minutes, one day, or even longer). Thus, we randomly sampled a hundred reports for the four projects marked with "*" in Table V. For the other projects, we manually inspected all CSA's reports since the number of warnings is much smaller.

We can observe from the results that, on average, CATAPULT's FP rate is 26.9%, which is the same as PINPOINT but much less than CSA's 58.7%. In other words, CATAPULT is able to satisfy the industrial requirement of less than a 30% FP rate [37], [38]. In terms of recall, CATAPULT reported 214 more real issues, which cover all real issues reported by CSA. In the reported issues of CATAPULT, at the time of writing, we have confirmed thirty-two previously-unknown vulnerabilities with the developers, including sixteen null pointer dereferences, nine use-after-free or double-free vulnerabilities, six resource leaks, and one stack-address-escape vulnerability. Three of them even have been assigned CVE IDs due to their severity and security impact. The CVE IDs are hidden because of the double-blind review regulation.

Together with the experiment results on scalability, we can conclude that the ideas behind CATAPULT have considerable practical value. In terms of all aspects, including not only the scaling effects, but also the precision and the recall, the tool itself is promising in providing industrial-strength capability of static bug finding.

VI. Related Work

Our work is related to several others in the literature. In what follows, we discuss the representative ones in three groups.

A. Static Bug Finding

Existing approaches to static bug finding track value flows in two manners, via the control flows or along the data dependence. The former are referred to as the "dense" approaches while the latter are known as the sparse approaches.

As discussed before, the core engine of "dense" approaches [9], [10], [39]–[48] cannot benefit from the heterogeneity of diverse vulnerability types to reduce the effort of program exploration. In addition, because they analyze a program statement by statement, many unrelated statements are analyzed. Several model checking techniques can check multiple vulnerability types at once but also observed scalability issues: checking multiple vulnerability types simultaneously will exhaust computing resources. To solve the problem, they have to make some tradeoffs: (1) isolating some checkers [49]–[52]; or (2) even making wrong assumptions [20]. Our approach is not a "dense" approach and is able to enjoy both the DD-homogeneity and the relational heterogeneity. It is also unnecessary for us to make the above tradeoffs.

SVFA techniques are known to be more efficient than "dense" approaches because they propagate values along data dependence instead of control flows [1]–[3], [6], [21]. Existing work either focuses on how to check a single vulnerability type such as null pointer dereference [3] and memory leak [1], [6] or proposes approaches to building more precise data dependence graphs [2], [6]. Thus, they actually study a fundamentally different problem.

Query languages are usually proposed to facilitate the use of static bug finding systems. The language METAL [31] uses the state machine as a fundamental abstraction and the language QL [32] works on a relational database. Differently, our language is designed based on the data dependence graph and it is unnecessary to work with a database. Yamaguchi et. al. [53], [54] and tools like PQL [33] and PIDGIN [27] use taint-style patterns to define taint-issue checkers. They are similar to us because they also focus on source-sink style vulnerabilities. However, their languages are not designed for relational heterogeneity and, thus, not concise or even not able to describe all checker-specific properties supported by us. We do not seek to argue that our language is novel or better than the ones above. However, the language is more suitable and extensible for our new design of SVFA: it can succinctly describe various relational checker-specific properties, which allows CATAPULT to be extensible for more checkers and, meanwhile, keep highly efficient when running multiple checkers.

B. Reusing Results to Improve Scalability

Reusing previously computed analysis results can significantly improve the scalability of program analysis. Researchers have proposed many approaches to reusing analysis results in different dimensions. First, because different programs can share common parts such as code libraries, lots of work has been proposed to extract knowledge from the common parts and reuse such knowledge to improve analysis efficiency [55]–[59]. However, this is a different problem, because they share the analysis results of the common code across different programs while we share the analysis results across various checkers for a single program.

Second, to improve the scalability of a single program analysis, compositional program analysis is an appealing approach as it can cache and reuse function summaries at different call sites [9], [10], [43], [60]. Our approach is also a compositional approach. The difference is that the summaries we produced are not only shared at different call sites for a single checker, but also reused across different checkers.

Third, since a lot of program analyses depend on low-level analysis such as points-to analysis and call graph analysis, reusing these fundamental results in different client analyses gradually becomes a common practice [55], [56], [61]. In comparison, our work proposes to share analysis results among multiple same-level analyses (i.e., the checkers).

C. Context-Free Grammar Based Modeling

Our theoretical framework describes our solution as an extended CFL-reachability problem that is originally introduced as a subclass of Datalog [62]. Recently, there have been many studies on Datalog engines [63]–[69], CFL-reachability algorithms [70], [71], and their applications. In static program analysis, the applications include data flow analysis [39], [72], pointer analysis [73]–[81], shape analysis [72], [82], program slicing [83], and type-based flow analysis [78], [84].

The aforementioned Datalog engines and CFL-reachability solvers can efficiently reply to queries on reachability relations, such as "is a source reachable to a sink?" and "what sinks can a source reach?". However, it is difficult for them to solve our problem, because such queries are not adequate for us. To check a source-sink problem, we have to find all paths from a source to a sink, each of which represents a cause of the vulnerability. In addition, when searching paths on the data dependence graph, we must ensure inter-procedurally path-sensitivity by solving complex path conditions and guarantee that the checker-specific properties are not violated. In comparison to the previous work, although our model is based on a context-free grammar, the role of the grammar is different: the grammar explicitly guides us to produce and compose function summaries for achieving the fine-grain DD-homogeneity.

VII. Conclusion

This paper presents CATAPULT, the first static bug finding system that can benefit from both the homogeneity and the heterogeneity of different vulnerability types. In our approach, the homogeneity is utilized at a fine-grain level and the heterogeneity is modeled as relations. As the number of vulnerability types increases quickly nowadays, we believe that the ideas behind CATAPULT as well as the approach itself will play important roles in the area of static bug finding.

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] 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, p. to appear.

[3] V. B. Livshits and M. S. Lam, “Tracking pointers with path and context sensitivity for bug detection in c programs,” *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 317–326, 2003.

[4] D. Hovemeyer and W. Pugh, “Finding more null pointer bugs, but not too many,” in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 9–14.

[5] D. Hovemeyer, J. Spacco, and W. Pugh, “Evaluating and tuning a static analysis to find null pointer bugs,” in *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1. ACM, 2005, pp. 13–19.

[6] 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.

[7] D. L. Heine and M. S. Lam, “A practical flow-sensitive and context-sensitive c and c++ memory leak detector,” in *ACM SIGPLAN Notices*, vol. 38, no. 5. ACM, 2003, pp. 168–181.

[8] M. Orlovich and R. Rugina, “Memory leak analysis by contradiction,” in *International Static Analysis Symposium*. Springer, 2006, pp. 405–424.

[9] 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.

[10] D. Babic and A. Hu, “Calysto: Scalable and precise extended static checking,” in *2008 ACM/IEEE 30th International Conference on Software Engineering*, ser. ICSE 2008. IEEE, May 2008, pp. 211–220.

[11] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, “Design and implementation of sparse global analyses for c-like languages,” in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 229–238.

[12] M. Das, S. Lerner, and M. Seigle, “Esp: Path-sensitive program verification in polynomial time,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI ’02. ACM, 2002, pp. 57–68.

[13] “Clang static analyzer.” https://clang-analyzer.llvm.org/, 2017.

[14] “Infer static analyzer.” http://fbinfer.com/, 2017.

[15] “Coverity scan.” https://scan.coverity.com/projects/, 2017.

[16] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[17] G. A. Kildall, “A unified approach to global program optimization,” in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1973, pp. 194–206.

[18] P. Cousot and R. Cousot, *Basic Concepts of Abstract Interpretation*. Kluwer Academic Publishers, 2004, pp. 359–366.

[19] M. N. Wegman and F. K. Zadeck, “Constant propagation with conditional branches,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991.

[20] E. Goldberg, M. Gdemann, D. Kroening, and R. Mukherjee, “Efficient verification of multi-property designs (the benefit of wrong assumptions),” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, 2018, pp. 43–48.

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

[22] “Static value-flow analysis in llvm – pointer analysis and program dependence analysis for c and c++ programs.” https://github.com/unsw-corg/SVF, 2017.

[23] E. W. Dijkstra, “On the role of scientific thought,” in *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 60–66.

[24] R. Jin, N. Ruan, Y. Xiang, and H. Wang, “Path-tree: An efficient reachability indexing scheme for large directed graphs,” *ACM Transactions on Database Systems (TODS)*, vol. 36, no. 1, p. 7, 2011.

[25] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “An efficient method of computing static single assignment form,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1989, pp. 25–35.

[26] J. E. Hopcroft, R. Motwani, and J. D. Ullman, “Automata theory, languages, and computation,” *International Edition*, vol. 24, 2006.

[27] A. Johnson, L. Waye, S. Moore, and S. Chong, “Exploring and enforcing security guarantees via program dependence graphs,” in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 291–302.

[28] “Canary: A unification-based alias analysis and some relative tools.” https://github.com/qingkaishi/canary, 2017.

[29] B. Steensgaard, “Points-to analysis in almost linear time,” in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996, pp. 32–41.

[30] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[31] S. Hallem, B. Chelf, Y. Xie, and D. Engler, “A system and language for building system-specific, static analyses,” in *ACM SIGPLAN Notices*, vol. 37, no. 5. ACM, 2002, pp. 69–82.

[32] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, “Ql: Object-oriented queries on relational data,” in *30th European Conference on Object-Oriented Programming*, 2016.

[33] M. Martin, B. Livshits, and M. S. Lam, “Finding application errors and security flaws using pql: a program query language,” *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 365–383, 2005.

[34] “Clang: a c language family frontend for llvm.” http://clang.llvm.org/, 2017.

[35] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, “In defense of soundiness: a manifesto,” *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[36] J. Dean, D. Grove, and C. Chambers, “Optimization of object-oriented programs using static class hierarchy analysis,” in *European Conference on Object-Oriented Programming*. Springer, 1995, pp. 77–101.

[37] 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.

[38] S. McPeak, C.-H. Gros, and M. K. Ramanathan, “Scalable and incremental software bug detection,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 554–564.

[39] 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*. ACM, 1995, pp. 49–61.

[40] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, “Modular verification of software components in c,” *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 388–402, 2004.

[41] I. Dillig, T. Dillig, and A. Aiken, “Sound, complete and scalable path-sensitive analysis,” in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 270–280.

[42] Y. Xie and A. Aiken, “Context-and path-sensitive memory leak detection,” in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 115–125.

[43] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, “Precise and compact modular procedure summaries for heap manipulating programs,” in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 567–577.

[44] C. Y. Cho, V. D’Silva, and D. Song, “Blitz: Compositional bounded model checking for real-world programs,” in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 136–146.

[45] T. Ball and S. K. Rajamani, “The slam project: Debugging system software via static analysis,” in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’02. ACM, 2002, pp. 1–3.

[46] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’02. ACM, 2002, pp. 58–70.

[47] E. Clarke, D. Kroening, and K. Yorav, “Behavioral consistency of c and verilog programs using bounded model checking,” in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 368–371.

[48] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “Predicate abstraction of ansi-c programs using sat,” *Formal Methods in System Design*, vol. 25, no. 2, pp. 105–127, 2004.

[49] S. Apel, D. Beyer, V. Mordan, V. Mutilin, and A. Stahlbauer, "On-the-fly decomposition of specifications in software model checking," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 349–361.

[50] G. Cabodi and S. Nocco, "Optimized model checking of multiple properties," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–4.

[51] P. Camurati, C. Loiacono, P. Pasini, D. Patti, and S. Quer, "To split or to group: from divide-and-conquer to sub-task sharing in verifying multiple properties," in *International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS), Lausanne, Switzerland*, 2014.

[52] V. O. Mordan and V. S. Mutilin, "Checking several requirements at once by cegar," *Programming and Computer Software*, vol. 42, no. 4, pp. 225–238, 2016.

[53] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 590–604.

[54] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 797–812.

[55] A. Rountev, M. Sharp, and G. Xu, "Ide dataflow analysis in the presence of large object-oriented libraries," in *International Conference on Compiler Construction*. Springer, 2008, pp. 53–68.

[56] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 378–400.

[57] S. Kulkarni, R. Mangal, X. Zhang, and M. Naik, "Accelerating program analyses by cross-program training," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016, pp. 359–377.

[58] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 58.

[59] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 177–187.

[60] P. Cousot and R. Cousot, "Modular static program analysis," in *International Conference on Compiler Construction*. Springer, 2002, pp. 159–179.

[61] X. Xiao, Q. Zhang, J. Zhou, and C. Zhang, "Persistent pointer information," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 463–474.

[62] M. Yannakakis, "Graph-theoretic methods in database theory," in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1990, pp. 230–242.

[63] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 196–206.

[64] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 422–430.

[65] K. Hoder, N. Bjørner, and L. De Moura, "μz–an efficient engine for fixed points with constraints," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 457–462.

[66] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005, pp. 97–118.

[67] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 243–262, 2009.

[68] "Logicblox - next generation analytics applications." https://developer.logicblox.com/, 2017.

[69] M. Madsen, M.-H. Yee, and O. Lhoták, "From datalog to flix: a declarative language for fixed points on lattices," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 194–208.

[70] N. Heintze and D. McAllester, "On the cubic bottleneck in subtyping and flow analysis," in *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*. IEEE, 1997, pp. 342–351.

[71] 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.

[72] T. Reps, "Program analysis via graph reachability," *Information and Software Technology*, vol. 40, no. 11-12, pp. 701 – 726, 1998.

[73] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, "Fast algorithms for dyck-cfl-reachability with applications to alias analysis," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 435–446.

[74] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su, "Efficient subcubic alias analysis for c," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 829–845.

[75] X. Zheng and R. Rugina, "Demand-driven alias analysis for c," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 197–208, 2008.

[76] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for java," in *ACM SIGPLAN Notices*, vol. 40, no. 10. ACM, 2005, pp. 59–76.

[77] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *ACM SIGPLAN Notices*, vol. 41, no. 6. ACM, 2006, pp. 387–400.

[78] P. Pratikakis, J. S. Foster, and M. Hicks, "Existential label flow inference via cfl reachability," in *International Static Analysis Symposium*. Springer, 2006, pp. 88–106.

[79] G. Xu, A. Rountev, and M. Sridharan, "Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 98–122.

[80] L. Shang, X. Xie, and J. Xue, "On-demand dynamic summary-based points-to analysis," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 264–274.

[81] D. Yan, G. Xu, and A. Rountev, "Demand-driven context-sensitive alias analysis for java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 155–165.

[82] 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*. ACM, 1995, pp. 1–11.

[83] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5, pp. 11–20, 1994.

[84] J. Rehof and M. Fähndrich, "Type-based flow analysis: from polymorphic subtyping to cfl-reachability," *ACM SIGPLAN Notices*, vol. 36, no. 3, pp. 54–66, 2001.