Precise and Scalable Static Bug Finding for Industrial-Sized Code

by

Qingkai Shi

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Department of Computer Science and Engineering

April 24 2020, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Qingkai Shi

April 24 2020

Precise and Scalable Static Bug Finding for Industrial-Sized Code

by

Qingkai Shi

This is to certify that I have examined the above PhD thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the PhD qualifying examination committee have been made.

Dr. Charles Zhang
(Thesis Supervisor)

Prof. Dit-Yan YEUNG
(Head)

April 24 2020, Hong Kong

To my family.

Acknowledgments

In the long journey of PhD study, with greatest fortunate, I have been helped, supported, and guided by many people. I am profoundly grateful to them.

I would like to thank my advisor, Dr. Charles Zhang, for the priceless guidance and help throughout these years. He taught me how to identify problems, solve problems, present ideas, write papers, manage time, and collaborate with others. It is his patience and commitment that changes me from an average man to a qualified PhD. I am very proud of working with him.

I would also like to thank the committee members of my PhD qualification examination, my thesis proposal defense, and my final thesis defense. They are Prof. Xiangyu Zhang from Purdue University, Prof. Weichuan Yu from the Department of Electronic and Computer Engineering, as well as professors from my major department, the Department of Computer Science and Engineering: Prof. Shing-Chi Cheung, Prof. Cunsheng Ding, Prof. Fangzhen Lin, Dr. Wei Wang, and Dr. Shuai Wang. Their valuable and insightful comments not only helped me a lot to improve the quality of my research and the thesis, but also inspired me to conduct better future research.

I am grateful for the help and friendship with which all friends and colleagues provide me. They have been my family in the Hong Kong University of Science and Technology, in no particular order: Jeff Huang, Xiao Xiao, Jinguo Zhou, Gang Fan, Rongxin Wu, Yepang Liu, Heqing Huang, Peisen Yao, Wensheng Tang, Yongchao Wang, Yiyuan Guo, Yushan Zhang, Chengpeng Wang, Yuandao Cai, Maryam Masoudian, Anshunkang Zhou, Hungchun Chui, Linjie Huang, Jiajun Gong, Lili Wei, Ming Wen, Yongqiang Tian, and Yuqing Quan.

Finally, my deepest gratitude goes to my family for their support and love, which have been a true inspiration and encouragement that pull me out of frustration and let me complete my PhD study.

My work is partially supported by the Hong Kong PhD fellowship scheme PF14-11387, a collaborative research grant from Microsoft Research, Asia, as well as Hong Kong GRF16214515, GRF16230716, GRF16206517, ITS/368/14FP, ITS/215/16FP, and ITS/440/18FP grants.

Contents

List of Figures

List of Tables

List of Abbreviations

CVE	Common Vulnerabilities and Exposures
FSVFG	Full-Sparse Value-Flow Graph
LoC	Lines of Code
SEG	Symbolic Expression Graph
SVFA	Sparse Value-Flow Analysis
SVFG	Sparse Value-Flow Graph

Abstract

Software bugs cost developers and software companies a great deal of time and money. Although previous work has reported many success stories for using static bug-finding tools, it is still difficult to find bugs hidden behind sophisticated pointer operations, deep calling contexts, and complex path conditions with a low false-positive rate, while sieving through millions of lines of code in just a few hours. In this thesis, we present our novel designs of sparse value-flow analysis to tackle a wide range of software bugs caused by improper value flows. The proposed approach has been commercialized and deployed in many of the global 500 companies. It also has reported hundreds of real bugs for open-source software systems.

The first problem addressed in the thesis is referred to as the pointer trap: a precise points-to analysis limits the scalability of sparse value-flow analysis and an imprecise one seriously undermines its precision. To solve the problem, we present Pinpoint, a holistic approach that decomposes the cost of high-precision points-to analysis by precisely discovering local data dependence and delaying the expensive inter-procedural analysis through memorization. Such memorization enables the on-demand slicing and, thus, improves the scalability with high precision. Experiments show that Pinpoint can check millions of lines of code in less than five hours with a false positive rate lower than 30%.

The second problem addressed in the thesis is known as the extensional scalability problem, which happens when we simultaneously check many value-flow properties with high precision. A major factor to this deficiency is that the core static analysis engine is oblivious of the mutual synergy among the properties being checked, thus inevitably losing many optimization opportunities. Our work is to leverage the inter-property awareness and to capture redundancies and inconsistencies when many properties are considered together. The evaluation results demonstrate that the approach, Catapult, is more than 8× faster than Pinpoint but consumes only 1/7 of the memory when checking twenty value-flow properties together.

The third problem addressed in the thesis is how to improve the parallelism of static bug finding over the conventional parallel designs. Conventionally, bottom-up program analysis has been easy to parallelize because functions without caller-callee relations can be analyzed independently. However, functions with caller-callee relations have to be analyzed sequentially because the analysis of a function depends on the analysis results of its callees. We present Cheetah, a framework of bottom-up analysis, in which the analysis task of a function is partitioned into multiple sub-

tasks. These sub-tasks are pipelined and run in parallel, even though the caller-callee relations exist. The evaluation results of Cheetah demonstrate significant speedup over a conventional parallel design.

Chapter 1

Introduction

1.1 Motivation

Software bugs cost developers and software companies a great deal of time and money. For instance, the Heartbleed bug[1] discovered in 2014 affected around 500,000 websites and hundreds of products from Cisco and Juniper. To tame such beasts hidden in software, developers from industry usually use static bug-finding tools to check possible bugs before a product is released.

Techniques following the design of conventional data-flow analysis and symbolic execution, such as the IFDS/IDE framework [93], Saturn [125], and Calysto [8], propagate data-flow facts to all program points following the control-flow paths. These "dense" analyses are known to have performance problems [89, 119, 21, 115]. For example, Babic and Hu [8] reported that it takes 6 to 11 hours for Saturn and Calysto to check the null-dereference bugs for programs of only 685 KLoC.

Sparse value-flow analysis (SVFA) mitigates this performance problem by tracking the flow of values via data dependence on sparse value-flow graphs (SVFG), thus eliminating the unnecessary value propagation [21, 115, 108, 114, 75]. Unfortunately, despite this tremendous progress, we still observe the difficulty of applying value-flow analysis at industrial scale — finding bugs hidden behind sophisticated pointer operations, deep calling contexts, and complex path conditions with low false positive rates, while sieving through millions of lines of code in just a few hours.

Figure 1.1 illustrates the workflow of SVFA, where the first step is to resolve pointer relations so that we can build data dependence hidden behind pointer

[1]Heartbleed Bug: `http://heartbleed.com`

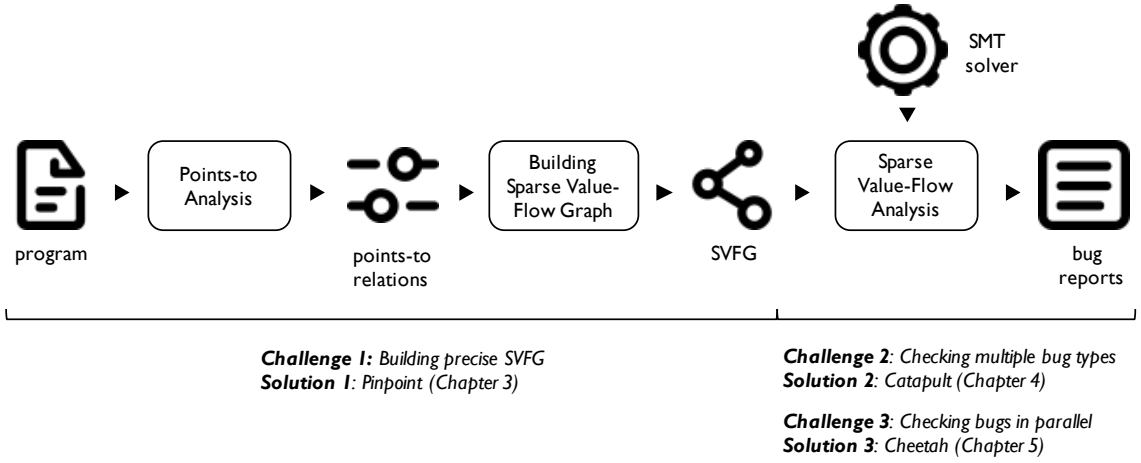

Figure 1.1: The workflow of sparse value-flow analysis.

operations. The second step is to build the SVFG based on the resolved data dependence relations, followed by a reachability analysis on the SVFG with the help of an SMT solver. We observe that, to make SVFA scalable and precise, we need to address the following problems in the state-of-the-art techniques.

(1) The Pointer Trap. The first problem is to build precise data dependence through a points-to analysis. Existing techniques discover data dependence through an independent points-to analysis. However, since a highly precise points-to analysis is difficult to scale to millions of lines of code [60], these approaches often give up the flow- or the context-sensitivity in the points-to analysis and avoid using SMT solvers to determine path-feasibility, such as in the cases of Fastcheck [21] and Saber [115]. Choosing a scalable but imprecise points-to analysis blows up the SVFG with false edges, overloads SMT solvers, and generates many false warnings, which we refer to as the pointer trap. In practice, we observe that developers have very low tolerance to such compromises because forsaking any of the following goals — scalability, precision, the capability of finding bugs hidden behind deep calling contexts and intensive pointer operations — creates major obstacles of adoption.

(2) The Extensional Scalability Problem. Modern static analyzers often need to simultaneously check a few dozen or even hundreds of value-flow properties. For instance, Fortify,[2] a commercial static code analyzer, checks nearly ten thousand value-flow properties from hundreds of unique categories. We observe that checking these properties together with a high precision causes serious scalability issues, which we refer to as the extensional scalability problem. To the best of our knowledge, a very limited number of existing static analyses have studied how to statically check

[2]Fortify Static Analyzer: `https://microfocus.com/products/static-code-analysis-sast`

multiple program properties at once, despite that the problem is very important at an industrial setting. A major factor to this problem, as we observe, is that the core static analysis engine is oblivious of the mutual synergy, i.e., the overlaps and inconsistencies, among the properties being checked, thus inevitably losing many optimization opportunities.

(3) The Limit of Parallelization. In multi-core era, we often take advantage of parallelization to scale up a program analysis. Conventionally, since our approach works in a bottom-up manner (i.e., before analyzing a function, all its callee functions are analyzed and summarized as function summaries [125, 8, 106, 18, 124, 20, 37, 22, 38]), it has been easy to parallelize the analyses of functions that do not have caller-callee relations. However, functions with caller-callee relations have to be analyzed sequentially because the analysis of a caller function depends on the analysis results, i.e., function summaries, of its callee functions. Otherwise, when analyzing the caller function, we may miss some effects of the callees due to the incomplete function summaries. With regard to the limitation of parallelization, McPeak et al. [79] pointed out that the parallelism often drops off at runtime and, thus, the CPU resources are usually not well utilized.

1.2 Contribution

With the aim of achieving the industrial requirement of static bug finding, i.e., checking millions of lines of code in 5 to 10 hours with less than 30% false positives [79, 13], this thesis makes three major contributions to scaling up path-sensitive static bug finding. Specifically, we present the following works in the thesis to address the aforementioned problems.

(1) Escaping from the Pointer Trap. We advocate a novel holistic approach to SVFA, nameed Pinpoint. In the approach, instead of hiding points-to analyses behind points-to query interfaces, we create an analysis slice, including points-to queries, value flows, and path conditions, that is just sufficient for the checked properties. In this manner, we can escape from the pointer trap by precisely discovering local data dependence first and delaying the expensive inter-procedural data dependence analysis through symbolically memorizing the non-local data dependence relations and path conditions. At the bug detection step, only the relevant parts of these mementos are further "carved out" in a demand-driven way to go for a high precision.

Experiments show that Pinpoint can check two million lines of code within 1.5 hours. The overall false positive rate is also very low, ranging from 14.3% to 23.6%.

(2) Conquering the Extensional Scalability Problem. We advocate an inter-property-aware design, namely Catapult, so that the core static analysis engine can exploit the mutual synergy among different properties for optimization. To check a value-flow property, users of our framework need to explicitly declare a simple property specification, which picks out source and sink values, respectively, as well as the predicate over these values for the satisfaction of the property. Surprisingly, given a set of properties specified in our property model, our static analyzer can automatically understand the overlaps and inconsistencies of the properties to check. Based on the understanding, before analyzing a program, we can make dedicated analysis plans so that, at runtime, the analyzer can share the analysis results on path-reachability and path-feasibility among different properties to reduce redundant graph traversals and unnecessary invocations of the SMT solver. The experimental results demonstrate that Catapult is more than 8× faster than Pinpoint but consumes only 1/7 of the memory when checking twenty common value-flow properties together.

(3) Breaking the Limit of Parallelization. To break the limit of parallelization caused by the calling dependence, we present Cheetah, a framework of bottom-up data flow analysis that breaks limits of function boundaries, so that functions with calling dependence can be analyzed in parallel without additional synchronizations. Our key insight is that many analysis tasks of a function only depend on partial analysis results of its callee functions. Our basic idea is to partition the analysis task of a function into multiple sub-tasks, so that we can pipeline the sub-tasks to generate function summaries. We formally prove the correctness of our approach and apply it to a null analysis and a taint analysis to show its generalizability. Overall, our pipeline strategy achieves 2× to 3× speedup over a conventional parallel design of bottom-up analysis. Such speedup is significant enough to make many overly lengthy analyses useful in practice.

1.3 Organization

The remainder of the thesis is organized as follows. Chapter 2 introduces the preliminaries and surveys the related work of static program analysis for bug finding. Chapter 3 presents the technique that allows us to escape from the "pointer trap" and path-sensitively check a value-flow property in millions of lines of code within

only several hours. Chapter 4 introduces how we conquer the extensional scalability problem by utilizing the mutual synergy among the procedures of checking different value-flow properties. Chapter 5 further introduces how we scale up static bug finding by relaxing the calling dependence between functions, so that the parallelism of bug finding can be significantly improved. We conclude this thesis and discuss our future work towards static bug finding in Chapter 6.

Chapter 2

Preliminaries and Background

2.1 Preliminaries

The thesis focuses on a technique of static bug finding, known as sparse value-flow analysis. Basically, sparse value-flow analysis is a kind of optimized data flow analysis. The latter propagates program information along control flows while the former propagates along data dependence to skip unnecessary control flows. To be clear, in this section, we first explain some basic concepts and terminologies of data flow analysis, followed by an introduction to sparse value-flow analysis.

2.1.1 Data Flow Analysis

Data flow analysis [66] is a well known technique for collecting information about how a possible set of values, also known as data flow facts, propagate in the control flow graph of a program. Formally, a control flow graph is defined as below.

Definition 2.1 (Control Flow Graph [1]). A control flow graph $G = (V, E)$ is a directed graph with a distinguished *entry* vertex and a distinguished *exit* vertex. The *entry* vertex does not have any predecessors and can reach every vertex. The *exit* vertex does not have any successors and it is reachable from every vertex.

A data flow problem is to determine how data flow facts propagate in a control flow graph. Formally, it is defined as below.

Definition 2.2 (Data Flow Problem [1]). A data flow problem is a five-tuple, $D = (L, \sqcap, G, M, c)$, where

- L is the domain of data flow facts and $(L, \sqcap)$ is a semi-lattice.
- $G = (V, E)$ is a control flow graph.
- $M : V \mapsto (L \mapsto L)$ is a map from the vertices to a set of transfer functions.
- $c \in L$ is the data flow fact associated with the *entry* (*exit*) vertex if it is a forward (backward) data flow problem.

For a forward data flow problem, the maximal fixed point solution is the maximal fixed point of the following equations over the set of variables $\{x_u : u \in V\}$ [1]:

$$\forall u \in (V \setminus \{entry\}), v \to u \in E : x_u = M(u)(\bigsqcap_{v \to u} x_v);\ \ x_{entry} = M(entry)(c)$$

For a backward data flow problem, the maximal fixed point solution is the maximal fixed point of the following equations over the set of variables $\{x_u : u \in V\}$ [1]:

$$\forall v \in (V \setminus \{exit\}), v \to u \in E : x_v = M(v)(\bigsqcap_{v \to u} x_u);\ \ x_{exit} = M(exit)(c)$$

Example 2.1 (Constant Propagation). Figure 2.1 shows how a data flow analysis addresses the problem of constant propagation, a forward data flow problem. The right side of the figure shows the semi-lattice of a single value, which is defined on the set $L = \mathbb{Z} \cup \{\text{UNDEF}, \text{NAC}\}$. Here, UNDEF means a value is undefined and NAC means Not A Constant. The meet operation, $\sqcap$, of the semi-lattice defines how to merge data flow facts at a merge point of multiple control flow paths. The meet operation follows the rules below.

- $\forall v \in L : \text{UNDEF} \sqcap v = v$. At a merge point of two paths, if a variable is UNDEF along a path but has a value $v \in L$ along the other, the variable will still have the value v after merging.
- $\forall v \in L : \text{NAC} \sqcap v = \text{NAC}$. At a merge point of two paths, if a variable is NAC along one of the two paths, the variable will be NAC after merging.
- $\forall v \in L : v \sqcap v = v$. At a merge point of two paths, if a variable has the same value along the two paths, the variable will keep the value after merging.
- $\forall v_1, v_2 \in \mathbb{Z} : v_1 \neq v_2 \Rightarrow v_1 \sqcap v_2 = \text{NAC}$. At a merge point of two paths, if a variable has different constant values along the two paths, the variable will be NAC after merging.

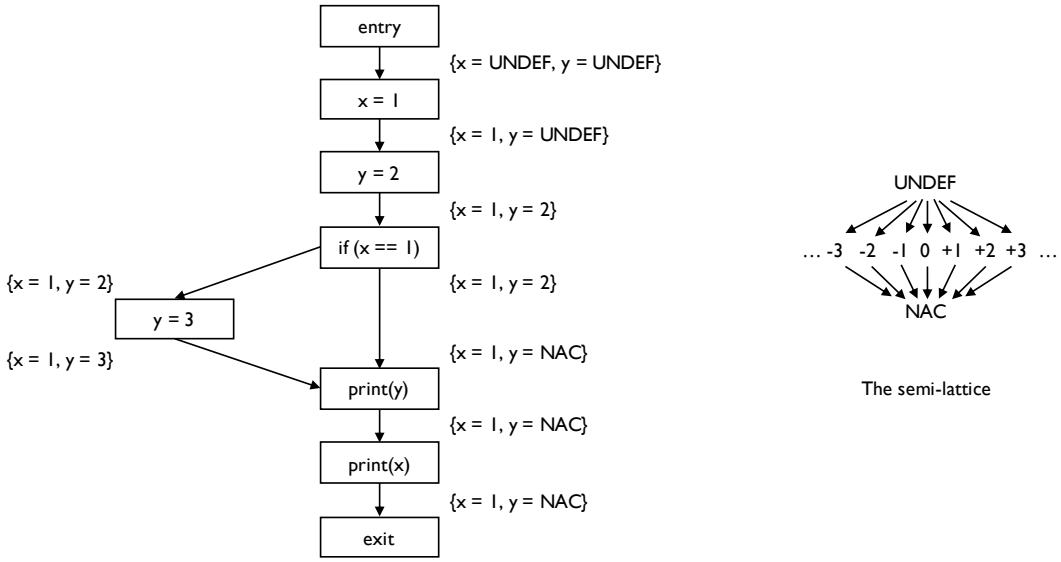

Figure 2.1: Constant propagation.

The left side of the figure shows how data flow facts are propagated via the transfer functions and the merging operations. At the very beginning, both of the variables, x and y, are undefined. After the first statement, the variable x is defined to be the constant 1. At the merge point, since the data flow facts, $y = 3$ and $y = 2$, hold for the two merging paths, respectively, the value of the variable y becomes NAC because $y = 2 \sqcap 3 = \text{NAC}$ according to the semi-lattice.

2.1.2 Sparse Value-Flow Analysis

The basic idea of sparse value-flow analysis is to skip irrelevant statements when propagating data flow facts in a data flow analysis. Typically, we can first transform a program to its static single assignment (SSA) form [28] where each variable only has one definition and def-use relations are explicitly encoded. Figure 2.2 demonstrates the SSA form of the program in Figure 2.1. Specially, at the merge point, a ϕ function is inserted to merge the value of the variable y from different paths. Using the SSA form, the data flow analysis in Example 2.1 then can propagate constant via def-use chains instead of control flows. As a result, it is not necessary to maintain all data flow facts at every program point. Thus, the analysis performance is improved. For example, we do not need to maintain the data flow fact $x = 1$ at the statements like $y = 2$. Wegman and Zadeck [119] proved that the SSA form improves the time complexity of constant propagation from $O(|E| \times |V|^2)$ to $O(|V|)$ in practice.

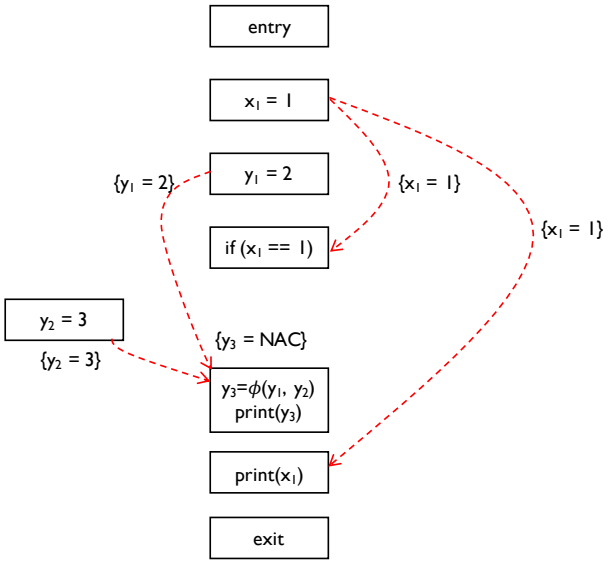

Figure 2.2: Constant propagation using a sparse analysis.

In addition to constant propagation, many classic data flow analysis can be performed in a "sparse" fashion based on the SSA form of a program. Typical examples include pointer analysis [55, 56], static bug-finding [115], and so on. However, in modern software, SSA form cannot effectively encode def-use relations hidden by intensive pointer operations, such as the relation between the variable a and the variable b in the code **p=a; q=p; b=*q*. To encode such indirect def-use relations, sparse value-flow graph has been introduced in recent studies [73, 115, 106], which is the fundamental data structure of our techniques in the thesis.

With no loss of generality, we assume the code in each function is in SSA form. We say the value of a variable a flows to a variable b if a is assigned to b directly (via an assignment, such as $b=a$) or indirectly (via pointer dereferences, such as **p=a; q=p; b=*q*). In addition, the value of a variable a directly flows to a statement if a is used in the statement. Formally, sparse value-flow graph is defined as below.

Definition 2.3 (Sparse Value-Flow Graph). A sparse value-flow graph (SVFG) is a directed graph $G = (V, E)$, where V and E are defined as following:

- V is a set of vertices, each of which is denoted by $v@s$, meaning that the variable v is defined or used in the statement s. When it does not cause any confusion, we directly write v or s in short.
- $E \subseteq V \times V$ is a set of edges, each of which represents a data dependence relation or value flow. $(v_1@s_1, v_2@s_2) \in E$ means that the value of $v_1@s_1$ flows to $v_2@s_2$.

A sparse value-flow analysis is to analyze the value flows in the SVFG, which underpins the inspection of a very broad category of software properties, such as memory safety (e.g., null dereference, double free, etc.), resource usage (e.g., memory leak, file usage, etc.), and security properties (e.g., the use of tainted data). In addition, there are a large and growing number of domain-specific value-flow properties. For instance, mobile software requires that the personal information cannot be passed to an untrusted code [6], and, in web applications, tainted database queries are not allowed to be executed [116]. Fortify, a commercial static code analyzer, checks nearly ten thousand value-flow properties from hundreds of unique categories. Value-flow properties exhibit a very high degree of versatility, which poses great challenges to the effectiveness of general-purpose program analyzers.

Value-flow properties checked in our static analyzer are related to well-known type-state properties [111, 112]. Generally, we can regard a value-flow property as a type-state property with at most two states. Nevertheless, value-flow properties have covered a wide range of program issues as discussed above. Thus, a scalable value-flow analyzer is really necessary and useful in practice. Modeling a program issue as a value-flow property has many advantages. For instance, Cherem et al. [21] pointed out that we can utilize the sparseness of value-flow graph to avoid tracking unnecessary value propagation in a control flow graph, thereby achieving better performance and outputting more concise issue reports.

2.2 Background

In what follows, we discuss the representative related work in three groups: scaling up static bug finding with high precision, scaling up static bug finding for multiple checkers, and scaling up static bug finding via parallelization.

2.2.1 Scaling up Static Bug Finding with High Precision

For the existing techniques for static bug finding, one major factor to the paradox between high scalability and high precision, as we observe, is related to the pointer analysis – a precise pointer analysis limits the scalability and an imprecise one seriously undermines the precision. To the best of our knowledge, all existing static bug-finding techniques utilizing value flows rely on a pre-computed points-to analysis to build data dependence. Since a precise pointer analysis is expensive [60], they

usually adopt a flow-insensitive analysis to avoid getting stuck in the pre-computation phase [75, 108, 114, 40, 21, 115, 30]. In contrast, the holistic design proposed in Chapter 3 of this thesis allows us to avoid an expensive pointer analysis but keep fully path-sensitive for bug finding.

There are also techniques adopting client- or demand-driven pointer analysis to reduce redundancy in static bug finding techniques. Client-driven pointer analysis [53, 54, 86] only performs higher-precision analysis in some parts of a program and cannot achieve the precision of inter-procedural path-sensitivity. In contrast, we can compute path-sensitive results in any part of the whole program. Demand-driven points-to analysis [58, 109, 101, 131, 126, 6] is in a fixed precision but computes only the necessary part of the solution. Existing approaches are not path sensitive. P/Taint [50] also integrates pointer analysis with value-flow analysis, but in a different manner: the value-flow analysis is implemented as an extension of the pointer analysis while our approach decomposes the cost of pointer analysis for value-flow analysis.

In addition to pointer analysis, there are also many other factors to the deficiency of existing static bug detectors. Abstraction based approach like SLAM [9], BLAST [59], and SATABS [26] adopt abstract refinement to improve scalability. However, the scalability degrades with the refinement of abstraction. CBMC [25, 26] also suffers from the scalability issue because it feeds constraints to an SAT solver regardless of whether they are relevant or not. Do et al. [39] proposed an approach built on the IFDS framework [93]. It is similar to our approach as it does local analysis first and then gradually extends to the whole project. Magic [20], Saturn [37, 124, 125], Calysto [8], Compass [38], and Blitz [22] are similar to our approach in terms of compositional analysis. However, these approaches have been demonstrated to be inefficient in detecting bugs that can be modeled as value-flow paths because, as discussed before, they are non-sparse techniques and unnecessary data-flow facts are tracked along control flows [89, 21, 115].

2.2.2 Scaling up Static Bug Finding for Multiple Checkers

To the best of our knowledge, a very limited number of existing static analyses have studied how to statically check multiple program properties at once, despite that the problem is very important at an industrial setting. Goldberg et al. [49] make unsound assumptions and intentionally stop the analysis on a path after finding the first bug. Apparently, the approach will miss many bugs, which violates our design goal. Different from our approach that reduces unnecessary program exploration

via cross-property optimization, Mordan and Mutilin [83] studied how to distribute computing resources, so that the resources are not exhausted by a few properties. Cabodi and Nocco [17] studied the problem of checking multiple properties in the context of hardware model checking. Their method has a similar spirit to our approach as it also tries to exploit the mutual synergy among different properties. However, it works in a different manner specially designed for hardware. In order to avoid state-space explosion caused by large sets of properties, some other approaches studied how to decompose a set of properties into small groups [19, 5]. Owing to the decomposition, we cannot share the analysis results across different groups.

There are also some static analyzers such as Semmle [7] and DOOP [15] that take advantage of Datalog engines for multi-query optimization. However, they are usually not path-sensitive and their optimization methods are closely related to the sophisticated datalog specifications. In this thesis, we focus on value-flow properties that can be simply described as conventional graph reachability queries and, thus, cannot benefit much from the datalog engines.

Clang[1] and Infer[2] currently are two of the most famous open-source static analyzers with industrial strength. Clang is a symbolic-execution-based, exhaustive, and whole-program static analyzer. As a symbolic execution, it suffers from the path-explosion problem [67]. To be scalable, it has to make unsound assumptions as in the aforementioned related work [49], limit its capability of detecting cross-file bugs, and give up full path-sensitivity by default. Infer is an abstract-interpretation-based, exhaustive, and compositional static analyzer. To be scalable, it also makes many trade-offs: giving up path-sensitivity and discarding sophisticated pointer analysis in most cases. Similarly, Tricoder, the analyzer in Google, only works intra-procedurally in order to analyze large code base [98, 99].

In the past decades, researchers have proposed many general techniques that can check different program properties but do not consider how to efficiently check them together [93, 9, 25, 20, 125, 37, 8, 38, 22, 114, 106]. Thus, we study different problems. In addition, there are also many techniques tailored only for a special program property, including null dereference [75], use after free [127], memory leak [124, 21, 115, 43], and buffer overflow [70], to name just a few. Since we focus on the extensional scalability issue for checking multiple properties, our approach is different from them.

[1]Clang Static Analyzer: `https://clang-analyzer.llvm.org`
[2]Infer Static Analyzer: `http://fbinfer.com`

2.2.3 Scaling up Static Bug Finding via Parallelization

Parallel and distributed algorithms for data flow analysis is an active area of research. In order to utilize the modular structure of a program to parallelize the analyses in different functions, developers usually implement a data flow analysis in a top-down fashion or a bottom-up manner. Top-down approaches work by processing the call graph of a program downwards from callers while bottom-up approaches work by processing the call graph upwards from callees. In our opinion, the top-down approach and the bottom-up approach are two separate schools of methodologies to implement program analysis. Bottom-up approaches analyze each function only once and generate summaries reusable at all calling contexts. Top-down approaches generate summaries that are specific to individual calling contexts and, thus, may need to repeat analyzing a function. For analyses that need high precision like path-sensitivity, repetitively analyzing a function is costly. Thus, we may expect better performance from bottom-up analysis when high precision is required.

Albarghouthi et al. [3] presented a generic framework to distribute top-down algorithms using a map-reduce strategy. Parallel worklist approaches, a kind of top-down analysis, operate by processing the elements on an analysis worklist in parallel [36, 96, 48]. These approaches are different from ours because our static analyzer works in a bottom-up manner. Compared to top-down analysis, bottom-up analysis has been traditionally easier to parallelize. Existing static analyses, such as Saturn [125], Calysto [8], Pinpoint [106], and Infer [18], have utilized the function-level parallelization to improve their scalability. However, none of them presented any techniques to further improve its parallelism. McPeak et al. [79] pointed out that the CPU utilization rate may drop in the dense part of the call graph where the parallelism is significantly limited by the calling dependence. Although they presented an optimized scheduling method to mitigate the performance issue, the calling dependence was not relaxed and the function-level parallelism was not improved. We believe that their scheduling method is complementary to our approach and their combination has the potential for the greater scalability.

In contrast to top-down and bottom-up approaches, partition-based approaches [51, 77, 41, 82, 24, 71, 11, 61] do not utilize the modular structure of a program but partition the state space and distribute the state-space search to several threads or processors. Another category of data flow analyses (e.g., [15, 4, 57]) are modeled as Datalog queries rather than the graph reachability queries in our approach. They can benefit from parallel Datalog engines to improve the scalability [65, 103, 47, 64, 104, 105, 78, 129, 123, 122].

Recently, some other parallel techniques have been proposed. Many of them focus on pointer analysis [74, 42, 113, 80, 87, 91] rather than general data flow analysis. Mendez-Lojo et al. [81] proposed a GPU-based implementation for inclusion-based pointer analysis. EigenCFA [90] is a GPU-based flow analysis for higher-order programs. Graspan [118] and Grapple [132] turn sophisticated code analysis into big data analytics. They utilize recent advances on solid-state disks to parallelize and scale program analysis. These techniques are not designed for compositional data flow analysis and, thus, are different from our approach.

In addition to automatic techniques, Ball et al. [10] used manually created harnesses to specify independent device driver entry points so that an embarrassingly parallel workload can be created.

Chapter 3

Scaling up Sparse Value-Flow Analysis with High Precision

3.1 Introduction

Sparse value-flow analysis (SVFA) underpins many recent techniques in statically finding bugs such as null pointer deference [8, 62, 63], memory leak [115, 21, 114, 124], use-after-free and double-free [21, 44, 35]. It is known to be more scalable than conventional data-flow analysis because it tracks the flow of values via data dependence on the sparse value-flow graph (SVFG), thus eliminating the unnecessary value propagation along control flows. However, we observe that the state-of-the-art SVFAs still compromise the following goals – scalability, precision, the capability of finding bugs hidden behind deep calling contexts and intensive pointer operations – which creates major obstacles of adoption.

3.1.1 The Pointer Trap

Existing SVFAs follow a "layered" design, which depends on an independent pointer analysis to build SVFG. However, since a highly precise pointer analysis is difficult to scale to millions of lines of code [60], these "layered" SVFA techniques often give up the flow- or the context-sensitivity in the pointer analysis and avoid using SMT solvers to determine path-feasibility, such as in the cases of Fastcheck [21] and Saber [115]. Choosing a scalable but imprecise pointer analysis blows up the SVFG with false edges, overloads SMT solvers, and generates many false warnings, which we refer to as the "pointer trap".

In this work, we make no claims of breakthroughs to the innate scalability limitations of pointer analysis and solving path conditions using SMT solvers. However, we note that the conventional "layered" approaches can significantly exacerbate the impact of these limitations on the perceived performance of SVFA, for which we are able to address. Our key insight is that an independent pointer analysis is unaware of the high-level properties being checked and, thus, causes a great deal of redundancy in computing pointer relations.

Let us illustrate this insight using the example in Figure 3.1(a), which contains an inter-procedural use-after-free bug, triggered when the "freed" pointer c in the function *bar* propagates to the dereference site at Line 9 of the function *foo*. Following a representative "layered" approach [21], we first build a global SVFG labeled with path conditions. To determine the value flow incurred by the expression *f=*ptr* at Line 8 of the function *foo*, a pointer analysis, whether exhaustive or demand-driven, is needed to discover that the pointer *ptr* can point to the five variables, a, b, c, d, and e, resolving the corresponding calling contexts of the function *bar* and the function *qux*, as well as checking the satisfiability of the five path conditions for the pointer relations. After building the SVFG, to find the use-after-free bug, we can traverse the graph from the vertex *free(c)*, generating a value-flow path, $(free(c), c, f, print(*f))$, that may trigger the bug with the associated path condition: $\theta_1 \wedge \theta_2 \wedge \theta_3$.

To summarize, in the above example, the "layered" conventional approach computes over five inter-procedural pointer relations, two calling contexts and six path conditions. However, if we take a "holistic" view across the layers of SVFA: pointer analysis, SVFG construction, and bug detection, it is easy to discover that, in this example, only the pointer relation between *ptr* and c is needed, one calling context between *bar* and *foo* required, and one path condition, $\theta_1 \wedge \theta_2 \wedge \theta_3$, to be solved.

3.1.2 Escaping from the Pointer Trap

In this work, we advocate a novel "holistic" approach to SVFA, in which, instead of hiding pointer analyses behind pointer query interfaces, we create an analysis slice, including pointer queries, value flows, and path conditions, that is just sufficient for the checked properties. We present Pinpoint, a technique that decomposes the cost of high-precision pointer analysis by precisely discovering local data dependence first and delaying the expensive inter-procedural data dependence analysis through symbolically memorizing the non-local data dependence relations and path conditions.

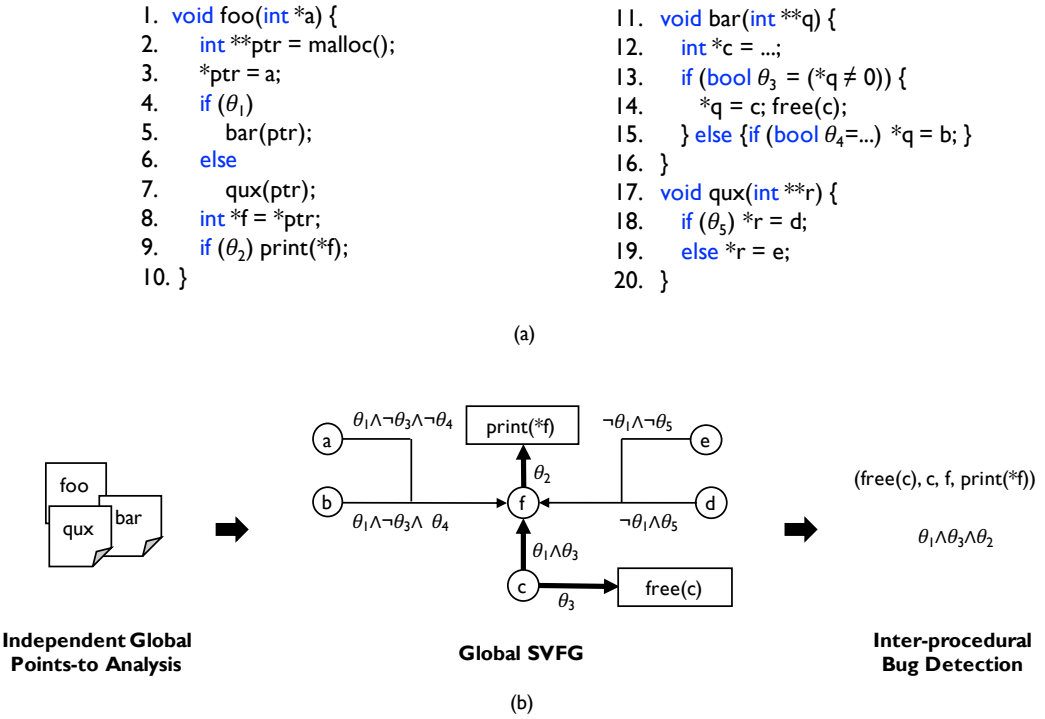

Figure 3.1: The "layered" design of SVFA.

At the bug detection step, only the relevant parts of these mementos are further "carved out" in a demand-driven way to go for a high precision.

The local analysis in Pinpoint is cheap due to a lightweight pointer analysis that identifies infeasible paths without an expensive SMT solver. In addition, to enable the inter-procedural and context-sensitive analysis, we only clone the memory access-path expressions that are rooted at a function parameter and incur certain side-effects. These clones serve as the context-sensitive "conduits" to allow values of interests flow in and out of the function scope on demand when answering value-flow queries. Summaries and path conditions are not cloned but memorized instead by our intra-procedural SVFG called the symbolic expression graph (SEG). Program properties are then checked by stitching together and traversing relevant SEGs. Along the way, data dependence relations hidden behind deep calling contexts, as well as the feasibility of the vulnerable paths, are determined altogether at the SMT solving stage.

Like many of the bug finding techniques [75, 124, 8, 115], Pinpoint is soundy [76]. However, it is comparatively much more scalable without sacrificing much precision and recall. We have used Pinpoint to check critical safety properties, such as use-after-free, double-free, and taint issues, on a large set of popular open-source

C/C++ systems. Although these systems have been checked by numerous free and commercial tools, we are still able to report and confirm over 40 previously-unknown use-after-free and double-free vulnerabilities, some of which are so serious and even assigned with CVE IDs. We show that Pinpoint has good scalability as it can build high precision SVFG up to >400X faster with only 1/4 memory space, compared to the state of the art. In addition, it is able to complete the inter-procedurally path-sensitive checking of a 2 MLoC code-base in 1.5 hours, fastest in terms of scale and precision, to the best of our knowledge. In summary, this chapter makes the following contributions:

- An efficient approach to building precise data dependence without an expensive global pointer analysis.
- A new type of SVFG, i.e., symbolic expression graph, which enables efficient path-sensitive analysis.
- A demand-driven and compositional approach to detecting bugs that can be modeled as value-flow paths.
- An implementation and an experiment that evaluates Pinpoint's scalability, precision, and recall.

3.2 Overview

To find the use-after-free vulnerability in Figure 3.1(a), our analysis, Pinpoint, runs in two steps: a semantic-preserving transformation that allows us to build precise SVFG and a graph traversal for inter-procedural bug detection.

3.2.1 Semantic-Preserving Transformation

Our analysis begins with an intra-procedural pointer analysis to analyze each function in a bottom-up manner, where we discover data dependence and function side-effects. Here, side-effects has a broader meaning, including both referencing and modifying non-local memory locations in a function. We then perform a semantic-preserving transformation of each function to explicitly expose side-effects on its interface, i.e., its parameters and return values.

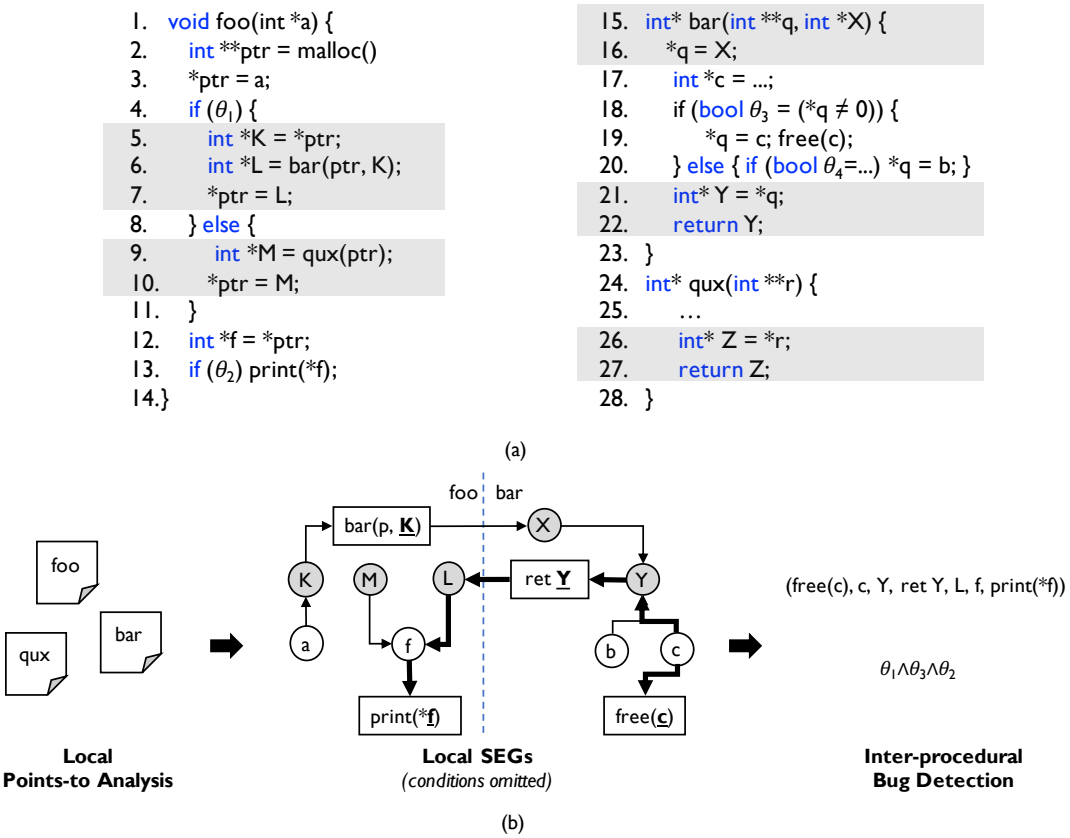

Figure 3.2: The "holistic" design of SVFA.

For instance, as illustrated in Figure 3.2(a), our pointer analysis identifies the side-effect incurred by the formal parameter q of the function *bar*: a load statement $*q{\neq}0$ and two store statements $*q{=}c$ and $*q{=}b$. We transform the function *bar* so that the value stored in the non-local memory, $*q$, is explicitly passed in via an extra formal parameter X and returned via an extra return value Y. To reflect the change of the signature of the function *bar*, its call site is transformed correspondingly as shown in Lines 5-7. The transformation of the function *qux* is similar. These transformations in the function *foo* set the stage for the same local pointer analysis for the function *foo*.

Based on both the local pointer analysis results and the transformed program, we build our local SVFG for each function, referred to as the symbolic expression graph (SEG), as shown in Figure 3.2(b). For example, to build data dependence for the variable f at Line 12, we first obtain the local points-to set, $\{(L, \theta_1), (M, \neg\theta_1)\}$ of the pointer *ptr*. Here, the points-to set means that, in the condition θ_1, the pointer *ptr* points-to the variable L and, otherwise, it points-to the variable M. Note that we do not invoke SMT solvers on path conditions θ_1 and $\neg\theta_1$ at this point but store them compactly in SEG, detailed later in the following sections.

3.2.2 Inter-procedural Bug Detection

To detect the use-after-free vulnerability, we traverse the SEG in Figure 3.2(b) and obtain a complete value-flow path, $(\textit{free(c)}, c, Y, \textit{return } Y, L, f, \textit{print(*f)})$, with a conjunction of all path conditions. Its feasibility is finally checked by an SMT solver. Notice that this path automatically prunes away the unrelated points-to target, M, together with its associated path condition, $\neg\theta_1$. Moreover, the path condition, θ_1, of the other target, L, is checked as part of the overall path condition of the vulnerability. To sum up, Pinpoint only computes one inter-procedural data dependence relation and solves one path condition.

In the next section, we formally present the function transformation rules and the construction algorithms for SEG. We will also explain how SEG facilitates the generation of function summaries, which enable fast inter-procedural analysis for bug detection.

3.3 A Holistic Design

The key design goal of Pinpoint is to escape from the pointer traps incurred by the conventional layered design of SVFAs [108, 114, 75, 21, 115]. In this section, we first explain how we decompose the pointer analysis so that the cheap data dependence is built first. We then define the symbolic expression graph (SEG) and explain SEG enables the demand-driven checking of properties, which simultaneously resolves the inter-procedural, context- and path-sensitive pointer relations.

To present our approach formally, we use the following simple call-by-value language similar to the previous work [38, 37]:

$$
\begin{array}{rcl}
\textit{Program } P & := & F + \\
\textit{Function } F & := & f\ (v_1, v_2, \cdots)\ \{\ S;\ \} \\
\textit{Statement } S & := & v_1 \leftarrow v_2 \mid v \leftarrow \phi(v_1, v_2, \cdots) \mid v_1 \leftarrow v_2\ \textit{binop}\ v_3 \mid v_1 \leftarrow\ \textit{unop}\ v_3 \\
& & \mid v_1 \leftarrow *(v_2, k \in \mathbb{N}^+) \mid\ *(v_1, k \in \mathbb{N}^+) \leftarrow v_2 \\
& & \mid \textbf{if}\ (v)\ \textbf{then}\ S_1; \textbf{else}\ S_2 \mid \textbf{return}\ v \mid r \leftarrow f(v_1, v_2, \cdots) \\
& & \mid S_1; S_2 \\
\textit{binop} & := & + \mid\ -\ \mid\ \wedge\ \mid\ \vee\ \mid\ >\ \mid\ =\ \mid\ \neq\ \mid\ \cdots \\
\textit{unop} & := & -\ \mid\ \neg\ \mid\ \cdots
\end{array}
$$

Statements in this language include common assignments, ϕ-assignments (assuming the SSA form), binary and unary operations, loads, stores, branches, returns, calls, and sequencing. With no loss of generality, we assume each function has only one return statement. We name the variable r at a call statement the "receiver" of the callee's return value. The operational semantics of most statements are standard and omitted. Specially, in a load/store statement, $*(v, k \in \mathbb{N}^+)$ means v is dereferenced k times, where k is a positive integer. We write $*v$ as a shorthand when $k = 1$.

3.3.1 Decomposing the Cost of Data Dependence Analysis

Building precise SVFGs requires to resolve data dependence through expensive context- and path-sensitive pointer analysis. Our solution is to perform a "quasi" path-sensitive and intra-procedural pointer analysis to resolve both the local data dependence and the function side-effects (also known as MOD/REF sets [8, 125]). Through a connector model, we compute the inter-procedural data dependence path- and context-sensitively in a demand-driven way, which significantly alleviates the cost of path and context explosion.

(1) A Quasi Path-Sensitive Pointer Analysis. We first perform a local pointer analysis for each function in a "quasi" path-sensitive manner, without expensive SMT solvers, but is able to prune most points-to relations that involve infeasible paths. The conditions of feasible paths are recorded to determine the feasibility of a value-flow path that may lead to a bug at the bug finding stage. In our experiment, we observed that about 70% of the path conditions constructed during the pointer analysis are satisfiable. Therefore, if we employ a full SMT solver at this local stage, the constraints of feasible points-to relations will be solved again at the bug finding stage, causing a great deal of redundancy, as illustrated by our motivating example.

Our solution is to introduce a linear-time constraint solver to filter out the "easy" unsatisfiable path conditions, i.e., the ones including apparent contradictions such as $a \wedge \neg a$. This is because, based on our observations, more than 90% of the unsatisfiable path conditions are easy constraints. The linear time constraint solver works in the way of continuously collecting positive and negative atomic constraints,[1] denoted by $P(C)$ and $N(C)$, respectively, during the construction of a constraint C. If there exists an atomic constraint $a \in P(C) \cap N(C)$, it means the constraint C contains an

[1] An atomic constraint is a bool-type expression without logic operators $\wedge, \vee$, and $\neg$. For example, $x = y + 1$ and z are two atomic constraints in $(x = y + 1) \wedge \neg z$.

apparent contradiction $a \wedge \neg a$ and, thus, is unsatisfiable. $P(C)$ and $N(C)$ are built using the following rules:

$$\begin{aligned} C = a \Rightarrow & P(C) = \{a\}, N(C) = \emptyset \\ C = \neg C_1 \Rightarrow & P(C) = N(C_1), N(C) = P(C_1) \\ C = C_1 \wedge C_2 \Rightarrow & P(C) = P(C_1) \cup P(C_2), N(C) = N(C_1) \cup N(C_2) \\ C = C_1 \vee C_2 \Rightarrow & P(C) = P(C_1) \cap P(C_2), N(C) = N(C_1) \cap N(C_2) \end{aligned}$$

Because the time complexity of the solver is linear to the number of atomic constraints, we pay a very low price to replace 90% of constraints that would otherwise require a full SMT solver. The path conditions found feasible by our linear time solver will be compactly encoded in our new type of SVFG, i.e., SEG, introduced later in Section 3.3.2.

(2) A Connector Model for Inter-procedural Data Dependence Analysis. The outcome of the pointer analysis is used to build the local data dependence obtained through pointer operations, e.g., connecting the load statement $p \leftarrow *q$ to the store statement $*u \leftarrow w$ if $*q$ and $*u$ are aliased. However, q and u could point-to non-local memory locations passed in by function invocations. Conventional summary-based approaches record the load and store statements that access non-local memory locations as the side-effect or the MOD/REF summary [8, 125], which is then cloned and instantiated at every call site of the summarized function in the upper-level callers. Due to a large number of the load and store statements in programs, the size of the side-effect summary can quickly explode and become a significant obstacle to scalability [2].

We noticed that the IFDS/IDE approaches solve this problem much more efficiently [93], in which the summary edges are built after analyzing each function, transferring the input data flow facts to the output without re-analyzing the function. These input-to-output fast tracks are used on-demand to the relevant data-flow problems and, therefore, avoid blindly inlining the unused data flow results to the callers. This idea inspires us to build the "connectors" for representing the input and output side-effects for each function.

For example, in Figure 3.2, the vertices X and Y are the input and output connectors for the function *bar*. Each input or output connector represents a memory location read from or write to via some load or store statements. At a call site, we build the call-site connectors, which work as actual parameters and return-value receivers. For example, the vertices K and L are the call-site connectors for the call

statement at Line 6. Then we can connect the vertex K to the vertex X and the vertex L to the vertex Y path- and context-sensitively if they are involved in building the inter-procedural data dependence. As described later, this connector model is sufficient to run a standard value-flow analysis for checking a value-flow property.

In Pinpoint, the input and output connectors are implemented by two kinds of auxiliary variables: the auxiliary formal parameter and the auxiliary return value.

Definition 3.1 (Auxiliary Variables). An auxiliary formal parameter is a local variable that stands for a non-local memory location referenced through a pointer expression $*(p, k \in \mathbb{N}^+)$, where p is a formal parameter. An auxiliary return value is defined similarly but the non-local memory location is modified.

Figure 3.3 defines the rules for inserting the auxiliary variables to represent the input and output connectors for functions and call sites. The code starting with ▶ is the target to transform and the result of each rule is the transformation result. In addition to the connectors, we also insert the load and store statements to model the relations between an auxiliary variable and corresponding actual or formal parameters, just as illustrated in Figure 3.2.

(3) Summary. In summary, the quasi path-sensitive pointer analysis and the connector model enable a holistic design: the pointer analysis is aware of its subsequent clients, including both the construction of SVFG and the subsequent analysis. Thus, the expensive context- and path-sensitive computations in the pointer analysis are delayed until the bug-finding phase. This holistic result cannot be achieved by an independent pointer analysis, whether exhaustive or demand-driven, in the conventional "layered" design. This is because, being unaware of the properties being checked, an independent pointer analysis always performs the expensive path- and context-sensitive computations for building the inter-procedural data dependence, which will be computed again in the bug detection phase.

3.3.2 Symbolic Expression Graph

Our analysis is based on a new type of SVFG, the symbolic expression graph (SEG), which enables the efficient and fully path-sensitive analysis through the following features. First, it compactly and precisely encodes all the conditional and unconditional data dependence, as well as the control dependence. Second, it enables the convenient query of the "efficient path conditions [108]" to provide the

$$\frac{\begin{array}{c}\blacktriangleright\ f(v_1, v_2, \cdots)\{\cdots; \textbf{return}\ v_0;\} \\ F_i \text{ is an auxiliary formal parameter of } f \\ F_i = *(v_j, k) \text{ at the beginning of } f (j > 0) \\ R_p \text{ is an auxiliary return value of } f \\ R_p = *(v_q, r) \text{ at the end of } f (q \geq 0)\end{array}}{\begin{array}{l} f(v_1, v_2, \cdots, F_1, F_2, \cdots)\ \{ \\ \quad *(v_j, k) \leftarrow F_i;\ \text{/* for all } (i, j, k). \text{ */} \\ \quad \cdots; \\ \quad R_p \leftarrow *(v_q, r);\ \text{/* for all } (p, q, r). \text{ */} \\ \quad \textbf{return}\ \{v_0, R_1, R_2, \cdots\}; \\ \}\end{array}}$$

(a)

$$\frac{\begin{array}{c}\blacktriangleright\ u_0 \leftarrow f(u_1, u_2, \cdots) \\ f(v_1, v_2, \cdots, F_1, F_2, \cdots)\{\cdots; \textbf{return}\ \{v_0, R_1, R_2, \cdots\};\} \\ F_i = *(v_j, k)(j > 0);\ R_p = *(v_q, r)(q \geq 0)\end{array}}{\begin{array}{c} A_i \leftarrow *(u_j, k);\ \text{/* for all } (i, j, k). \text{ */} \\ \{u_0, C_1, C_2, \cdots\} \leftarrow f(u_1, u_2, \cdots, A_1, A_2, \cdots); \\ *(u_q, r) \leftarrow C_p;\ \text{/* for all } (p, q, r). \text{ */}\end{array}}$$

(b)

Figure 3.3: Rules of the semantic-preserving transformation.

full support of path-sensitive analysis. Finally, it is separately built for each function, not only saving time costs, but also enabling the efficient compositional analysis.

(1) Definition. Formally, we define the symbolic expression graph, a new kind of SVFG, as below.

Definition 3.2. The symbolic expression graph (SEG) of a function consists of two sub-graphs, i.e., $\mathcal{G}_d = (\mathcal{V} \cup \mathcal{O}, \mathcal{E}_d, \mathcal{L}_d)$ and $\mathcal{G}_c = (\mathcal{V}, \mathcal{E}_c, \mathcal{L}_c)$, describing the data dependence and the control dependence, respectively:

- $\mathcal{V}$ is a set of vertices, each of which is denoted by $v@s$, meaning the variable v defined or used at a statement s. If v is defined at s, we write $v@s$ as v for short, because v is defined exactly once in SSA form and the abbreviation will not cause ambiguity. $\mathcal{V}_b \subseteq \mathcal{V}$ is the set of all boolean variables in $\mathcal{V}$. $\mathcal{O}$ is a set of binary or unary operator vertices, each of which represents a symbolic expression.

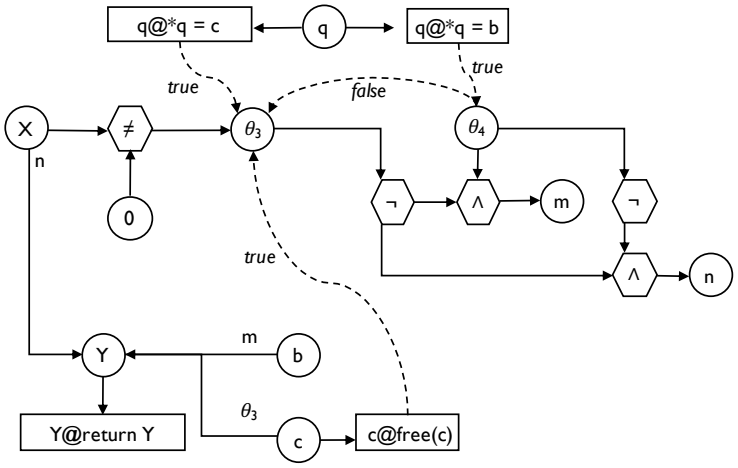

Figure 3.4: The complete SEG of the function *bar*.

- $\mathcal{E}_d \subseteq (\mathcal{V} \cup \mathcal{O}) \times (\mathcal{V} \cup \mathcal{O})$ is a set of directed edges, each of which represents a data dependence relation. The labeling function, $\mathcal{L}_d : \mathcal{E}_d \mapsto \{true\} \cup \mathcal{V}_b$, represents the condition on which a data dependence relation holds. Specially, a directed edge $(v_1@s_1, o) \in \mathcal{V} \times \mathcal{O} \subseteq \mathcal{E}_d$, labeled by *true*, means the variable v_1 defined at the statement s_1 is used as an operand of the operator o. A directed edge $(o, v_1@s_1) \in \mathcal{O} \times \mathcal{V} \subseteq \mathcal{E}_d$, labeled by *true*, means the result of the operator o defines the variable v_1 at the statement s_1.

- $\mathcal{E}_c \subseteq \mathcal{V} \times \mathcal{V}_b$ is a set of directed edges, each of which represents a control dependence relation. The labeling function, $\mathcal{L}_c : \mathcal{E}_c \mapsto \{true, false\}$, implies that only if $v_2@s_2 = \mathcal{L}_c((v_1@s_1, v_2@s_2))$, $v_1@s_1$ is reachable.

Following Definition 3.2, we build the SEG for each function. As an example, the SEG of the function *bar* in Figure 3.2 is shown in Figure 3.4. Solid edges in the figure represent data dependence. The label *true* for unconditional data dependence is omitted. Dashed edges represent control dependence.

In SEG, the definition and the use of all variables, as well as operators, are modeled as vertices, which are similar to those in the conventional approaches [21, 114, 115]. Vertices for operators are used to represent symbolic expressions, as illustrated in Example 3.1. These operator vertices enable us to efficiently query symbolic expressions (e.g., $a = b + c$) instead of simple def-use relations (e.g., b and c are used to define a). Thus, they can help construct path conditions.

Example 3.1. As shown in Figure 3.4, the expression "$X \neq 0$" is explicitly presented by an operator vertex "$\neq$" and two other vertices standing for its operands, i.e., "X" and "0".

Following the previous work [45], each directed edge in SEG represents either a data dependence relation or a control dependence relation, labeled with the condition on which the dependence holds. The data dependence concealed by pointer operations are collected by the pointer analysis. For each ϕ-assignment, $v \leftarrow \phi(v_1, v_2, \cdots)$, the condition for selecting v_i is known as the gated function, which can be computed in almost linear time [117]. Example 3.2 shows two concrete examples for unconditional and conditional data dependence in SEG, respectively. Control dependence represents the branch conditions on which a statement is reachable at runtime [45]. The control dependence of a statement is in the form v or $\neg v$ where v is a branch-condition variable. Example 3.3 shows a concrete example of control dependence in SEG.

Example 3.2. In Figure 3.4, the data dependence edge $(q, *q = b)$ does not have any label, because the dependence is unconditional ($*q = b$ always depends on q). The data dependence edge (b, Y) is labeled m, because the dependence is conditional: $m \Rightarrow Y = b$. According to the pointer analysis, m is equal to $\neg\theta_3 \wedge \theta_4$, which is encoded in the graph using the same method described in Example 3.1.

Example 3.3. In Figure 3.4, the control dependence of the statement $*q = b$ is θ_4 and, thus, there is an $\mathcal{L}_c$-labeled edge from the statement to θ_4 (labeled *true*). In addition, the control dependence of the statement defining θ_4 is $\neg\theta_3$ and, thus, there is an $\mathcal{L}_c$-labeled edge from θ_4 to θ_3 (labeled *false*).

(2) Querying "efficient path conditions" on SEG. The design of SEG enables us to conveniently query the "efficient path condition [108]" of a value-flow path, which is much more succinct than those computed according to the definition of path condition [67]. Intuitively, an efficient path condition only contains the necessary data dependence and control dependence so that the value-flow path is feasible at runtime. The following is an example.

Example 3.4. Based on the SEG in Figure 3.4, the "efficient path condition" on which the statement "return Y" is reachable is *true*, because there are no control-dependence edges outgoing from the vertex Y@return Y. It does not contain any unnecessary branch-condition variables like θ_3 and θ_4. In comparison, if we follow

the canonical definition [67] to compute the path condition of the same statement, it will be the disjunction of the path conditions of all paths from the entry to the exit of the function, i.e., $\theta_3 \vee (\neg\theta_3 \wedge \theta_4) \vee (\neg\theta_3 \wedge \neg\theta_4)$, which is verbose and inefficient.

Given a value-flow path, π=$(v_1@s_1, \cdots, v_n@s_n)$, in $\mathcal{G}_d$, the basic idea of computing the "efficient path condition" is to conjunct the data dependence and control dependence associated with this path. For a given vertex, $v@s$, in SEG, we introduce two functions, $\mathrm{DD}(v@s)$ (see Example 3.5) and $\mathrm{CD}(v@s)$ (see Example 3.6), to compute the constraints that describe the data dependence and the control dependence, respectively. The path condition of the path π is computed as following:

$$\begin{aligned}\mathrm{PC}(\pi) = & \bigwedge_{i=1\cdots n} \mathrm{CD}(v_i@s_i) \wedge \bigwedge_{i=2\cdots n} (v_{i-1}@s_{i-1} = v_i@s_i) \\ & \wedge \bigwedge_{i=2\cdots n} \mathcal{L}_d((v_{i-1}@s_{i-1}, v_i@s_i)) \\ & \wedge \bigwedge_{i=2\cdots n} \mathrm{DD}(\mathcal{L}_d((v_{i-1}@s_{i-1}, v_i@s_i)))\end{aligned} \tag{3.1}$$

As shown in the above equation, a path condition includes following parts: (1) $\mathrm{CD}(v_i@s_i)$ represents the condition on which s_i is reachable at runtime; (2) $v_{i-1}@s_{i-1} = v_i@s_i$ describes the fact that the value stored in $v_{i-1}@s_{i-1}$ flows to $v_i@s_i$; (3) the remaining part represents the condition on which the value flow from $v_{i-1}@s_{i-1}$ to $v_i@s_i$ is feasible.

Example 3.5. Assume we are computing the data dependence of Y shown in Figure 3.4. $\mathrm{DD}(Y)$ will result in the constraint: $(n \Rightarrow Y = X) \wedge (m \Rightarrow Y = b) \wedge (\theta_3 \Rightarrow Y = c) \wedge \mathrm{DD}(n) \wedge \mathrm{DD}(X) \wedge \mathrm{DD}(m) \wedge \mathrm{DD}(b) \wedge \mathrm{DD}(\theta_3) \wedge \mathrm{DD}(c)$. This is because the sources of incoming edge of Y are X, b, and c, labeled by n, m, and θ_3, respectively. Also, we should recursively compute the data dependence of n, X, m, b, θ_3, and c.

Example 3.6. To compute the control dependence of $q@*q = b$, shown in Figure 3.4, $\mathrm{CD}(q@*q = b)$ results in the constraint: $\theta_4 \wedge \neg\theta_3 \wedge \mathrm{DD}(\theta_4) \wedge \mathrm{DD}(\theta_3)$. This is because there is a *true*-labeled control-dependence edge from the vertex to θ_4 and a *false*-labeled control-dependence edge from the θ_4 to θ_3. Also, we should recursively compute the data dependence of θ_3 and θ_4.

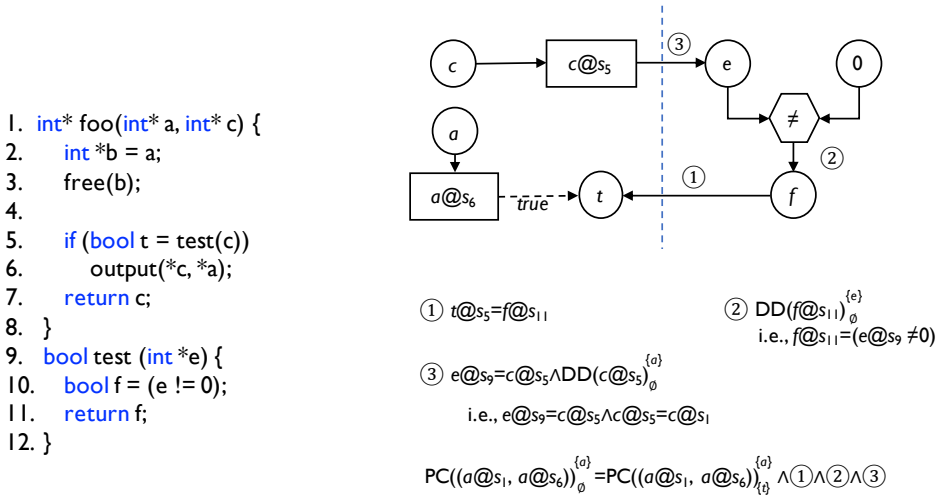

Figure 3.5: An example to illustrate our inter-procedural analysis.

3.3.3 Global Value-Flow Analysis

The inter-procedural analysis in Pinpoint addresses two problems to achieve precision and efficiency. The first is how to achieve path- and context-sensitivity when stitching value flows from different functions. The other is how to reuse analysis results to avoid repeated computation, thereby improving efficiency. We now explain how to perform path- and context-sensitive SVFA in a demand-driven way.

(1) Achieving inter-procedural path-sensitivity. To achieve inter-procedural path-sensitivity, the key is to compute the path condition of a global value-flow path, for which we need to address two problems. First, given a local value-flow path π in a function, the afore-defined $\mathrm{PC}(\pi)$ (Equation (3.1) only computes the path condition based on the function's local SEG. Thus, the resulting condition loses the constraints from both of its callers and callees, which we should be able to recover. Second, we also should be able to compute the path condition of any global value-flow path.

To explicitly describe what is lost in a formula like $\mathrm{PC}(\cdot)$, we rewrite it as $\mathrm{PC}(\cdot)^P_R$ where P and R are the sets of function parameters and return-value receivers, of which the constraints are lost, respectively. The following is an example to illustrate P and R.

Example 3.7. In the example, we use s_i to stand for the statement at Line i. In Figure 3.5, for the local value-flow path $(a@s_1, a@s_6)$, according to Equation (3.1), its path condition will be $t@s_5 = true \wedge a@s_1 = a@s_6$, where the constraints of the parameter $a@s_1$ and the return-value receiver $t@s_5$ are lost. Thus, we can write $\mathrm{PC}((a@s_1, a@s_6))^{\{a@s_1\}}_{\{t@s_5\}} = (t@s_5 = true \wedge a@s_1 = a@s_6)$.

Because Pinpoint performs a bottom-up program analysis that always analyzes callees before callers, we only can recover the lost constraints from the callees when analyzing a function. That is, we only can eliminate the dependence on the return-value receivers in $\mathrm{PC}(\cdot)_R^P$, so that it can be written as $\mathrm{PC}(\cdot)_\emptyset^{P'}$. Note that the dependence on P then can be eliminated by adding the constraints of the actual parameters when the caller function is analyzed. The basic idea of eliminating the dependence on R is that, for each return-value receiver in R, we add the constraints of the corresponding return value, which can be computed based on the callee's SEG. The following is an example.

Example 3.8. Following the last example, we need to add the constraints of $t@s_5$ into $\mathrm{PC}((a@s_1, a@s_6))_{\{t@s_5\}}^{\{a@s_1\}}$, so that the dependence on the return-value receiver $t@s_5$ can be eliminated. As a result, we get the precise path condition, which can be recorded as $\mathrm{PC}((a@s_1, a@s_6))_\emptyset^{\{a@s_1, c@s_1\}}$. Apparently, the constraint to add for $t@s_5$ is $t@s_5 = f@s_{11} \wedge\ f@s_{11} = (e@s_9 \neq 0) \wedge e@s_9 = c@s_5 \wedge c@s_5 = c@s_1$, which consists of three parts:

① $t@s_5 = f@s_{11}$ describes the fact that the return-value receiver is equal to the corresponding return value.

② $f@s_{11} = (e@s_9 \neq 0)$ describes the value range of the callee's return value $f@s_{11}$, which depends on the function's parameter $e@s_9$ that is passed in via the actual parameter $c@s_5$ at Line 5.

③ $e@s_9 = c@s_5 \wedge c@s_5 = c@s_1$ describes the dependence of the actual parameter.

Following the above example, formally, we can convert $\mathrm{PC}(\pi)_R^P$ to $\mathrm{PC}(\pi)_\emptyset^{P'}$ by adding the three parts of conditions (① - ③) as

$$\begin{aligned}\mathrm{PC}(\pi)_\emptyset^{P'} = \ \mathrm{PC}(\pi)_R^P \wedge \bigwedge_{v_i@s_i \in R} \underbrace{v_i@s_i = \mathbb{M}(v_i@s_i)}_{①} \wedge \underbrace{\boldsymbol{\mathrm{DD}(\mathbb{M}(v_i@s_i))_\emptyset^{Q_i}}}_{②} \wedge \\ \underbrace{\bigwedge_{v_j@s_j \in Q_i} v_j@s_j = \mathbb{M}(v_j@s_j) \wedge \mathrm{DD}(\mathbb{M}(v_j@s_j))_\emptyset^{P_j}}_{③}\end{aligned} \tag{3.2}$$

In the equation, the bold part is the constraints from the callee function. $\mathbb{M}$ represents a mapping between a pair of formal and actual parameters or a pair of return value

and its receiver. P' is the union of P and all P_j. $\mathrm{DD}(\cdot)_{\emptyset}^{P'}$ can be converted from $\mathrm{DD}(\cdot)_{R}^{P}$ recursively in a similar way.

The next problem is to compute the precise path condition of a global value-flow path across different functions. That is, given two local value-flow paths from two functions, π_1=$(v_1@s_1, \cdots, v_n@s_n)$ and π_2=$(u_1@r_1, \cdots u_n@r_n)$, we need to generate the path-condition of their connection $\pi_1\pi_2$, where $v_n@s_n$ and $u_1@r_1$ is a pair of formal and actual parameters or a pair of return value and its receiver. With no loss of generality, we assume $v_n@s_n$ is an actual parameter and $u_1@r_1$ is the corresponding formal parameter. Then π_1 is in a caller function and π_2 is in one of its callees. The precise path condition of $\pi_1\pi_2$ can be generated as below where the bold part is the constraints from the callee function.

$$\begin{aligned}\mathrm{PC}(\pi_1\pi_2)_{\emptyset}^{P} =& \mathrm{PC}(\pi_1)_{\emptyset}^{P_1} \wedge \mathbf{PC(\pi_2)_{\emptyset}^{P_2}} \wedge v_n@s_n = u_1@r_1 \wedge \\ & \bigwedge_{v_i@s_i \in P_2} v_i@s_i = \mathbb{M}(v_i@s_i) \wedge \mathrm{DD}(\mathbb{M}(v_i@s_i))_{\emptyset}^{Q_i}\end{aligned} \tag{3.3}$$

The first row of the equation includes the path conditions of both paths, as well as the fact $v_n@s_n$ flows to $u_1@r_1$. Because the path condition of π_2 may depend on the callee's formal parameters, we add the conditions of the corresponding actual parameters in the second row of the above equation. Apparently, P is the union of P_1 and all Q_i.

(2) Achieving context-sensitivity. We follow the cloning-based approach to achieve context-sensitivity [69, 120]. That is, if a function is used at multiple call sites, constraints computed based on the function's SEG is cloned to distinguish different call sites.

(3) Demand-driven searching. Because the bug detection process is to search the value-flow paths starting from a bug-specific source vertex, the path- and context-sensitive computations are only carried out for the bug-related paths. Therefore, this is a demand-driven process that avoids the exhaustive path- and context-sensitive computation.

(4) Compositional approach to bug detection. It is well known that bottom-up compositional approach can improve the efficiency of program analysis, because we can summarize function behaviors and reuse function summaries at different call sites [8, 125]. According to the computation of inter-procedural path condition, whenever we analyze a function, we actually require two kinds of information from the callees (see the bold parts in Equations (3.2) and (3.3)). One is the data

dependence, $\mathrm{DD}(v@s)^P_\emptyset$, where $v@s$ is a callee's return value. The other is $\mathrm{PC}(\pi)^P_\emptyset$ where π is a value-flow path in certain callee function. Thus, we generate two types of summaries for them, the return-value (RV) summary and the value-flow (VF) summary, respectively.

As described by the data-dependence constraints of a return value, $\mathrm{DD}(v@s)^P_\emptyset$, an RV summary, which summarizes the value range of a function's return value, is a three-tuple consisting of:

- An SEG vertex $v@s$ that stands for a return value.
- A constraint that restricts the range of the return value, i.e., $\mathrm{DD}(v@s)^P_\emptyset$.
- A subset P of the function's formal parameters that the constraint depends on.

As described by the path condition, $\mathrm{PC}(\pi)^P_\emptyset$, a VF summary, which summarizes value flows in a function, is a three-tuple:

- A list of SEG vertices standing for a value-flow path π.
- The condition on which the summarized value-flow path is feasible at runtime, i.e., $\mathrm{PC}(\pi)^P_\emptyset$.
- A subset P of the function's formal parameters that the condition depends on.

To detect a bug that can be modeled as a global value-flow path between a pair of bug-specific "source" and "sink" vertices, we define four kinds of VF summaries:

- VF1 summarizes a value-flow path from a function parameter to a return value.
- VF2 summarizes a value-flow path from a "source" to a return value.
- VF3 summarizes a value-flow path from a function parameter to a "source".
- VF4 summarizes a value-flow path from a function parameter to a "sink".

The above VF summaries describe all possible relations between the bug-specific vertices (i.e., sources and sinks) and the function interface values (i.e., function parameters and return values). VF1 determines whether an actual parameter at a call site would flow back to the return-value receiver at the same call site. Thus, when reaching an actual parameter during path-searching, VF1 decides whether we

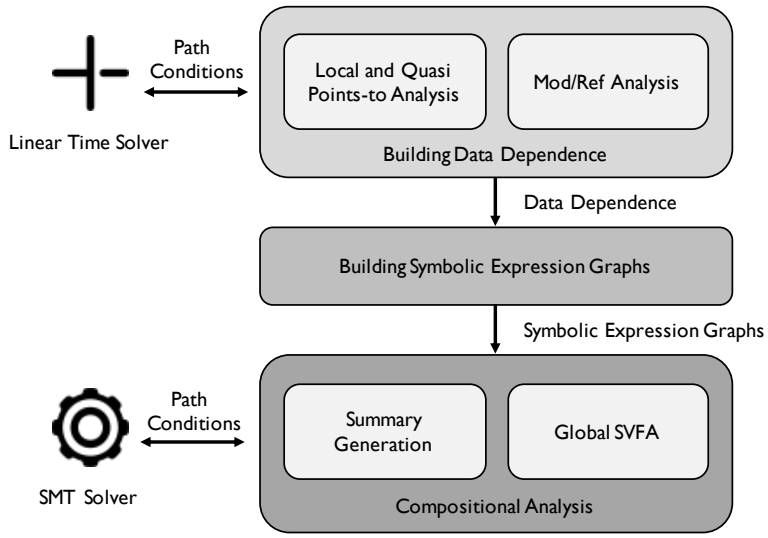

Figure 3.6: The architecture of Pinpoint.

should continue the search starting from the return-value receiver. VF2 and VF3 determine if a return-value receiver and an actual parameter would become buggy (i.e., get value from a bug-specific source) after a call statement, respectively. They help to decide whether we should start the path search from a return-value receiver or an actual parameter when analyzing a function. We show an example of the VF3 summary in Figure 3.5. In order to detect the use-after-free vulnerability, we create a VF3 summary containing the value-flow path $(a@s_1, b@s_2, b@s_3)$, which summarizes the behavior of function *foo*: after calling function foo, the function parameter a is "freed". VF4 determines if an actual parameter at a call site would flow to a sink in the callee. A bug may happen in the callee if we reach an actual parameter during the path search and the callee has a VF4 summary.

3.4 Implementation

Pinpoint is implemented on top of LLVM 3.6 [68] using Z3 [31] as the SMT solver. Its main architecture is shown in Figure 3.6.

3.4.1 Checkers

To evaluate Pinpoint as a general framework, we have been continuously adding "checkers" in addition to those for use-after-free and double-free. In our experience, problems that can be modeled as value-flow paths are straightforward to solve using

Pinpoint. For instance, a path-traversal vulnerability, which is a taint issue, allows an attacker to access files outside of a restricted directory.[2] It can be modeled as a value-flow path starting with SEG vertices representing user inputs like *input@input=fgetc()*, and ending with SEG vertices representing operations on files like *path@fopen(path, ...)* [95]. Similarly, a data transmission vulnerability may leak sensitive data to attackers.[3] It can be modeled as a value-flow path starting with SEG vertices representing sensitive data like *password@password=getpass(...)*, and ending with SEG vertices representing statements that may leak information like *data@sendto(data, ...)* [95]. Similar to the previous taint analysis work [6], we have not modeled the sanitization operations in our taint-issue checkers.

3.4.2 Soundness

Pinpoint is soundy [76] as it shares the same "standard assumptions" with previous techniques that aim to find bugs rather than rigorous verification [75, 125, 21, 8, 115]. In our implementation, we regard all elements in an array or a union structure to be aliases and unroll each loop once in control flow graphs and call graphs. Following Saturn [125], we currently have not modeled inline-assembly and function pointers, but we adopt a class hierarchy analysis to resolve virtual function calls [32]. Also, we assume distinct parameters of a function do not alias with each other, which potentially can be improved using the idea of partial transfer function [121] in the future. For library code, we manually model some standard C libraries like *memset* and *memcpy*, which are significant for the points-to analysis, but have not modeled standard template libraries, such as *std::vector* and *std::map*.

3.5 Evaluation

We aim to, as systematic as possible, evaluate the precision, the recall, and the scaling effect of Pinpoint, due to the extensive work from both academia and industry in scaling static bug finding to industrial-sized software systems. We not only compared Pinpoint to the state-of-the-art techniques of SVFA, but also conducted comparison experiments on the tools using abductive inference (Infer) and symbolic execution (Clang). We also sought to evaluate other prominent static bug detection

[2]Relative Path Traversal: `https://cwe.mitre.org/data/definitions/23.html`
[3]Resource Leak: `https://cwe.mitre.org/data/definitions/402.html`

implementations such as Saturn, Compass, and Calysto. However, they are either unavailable or outdated for the operating systems we are able to set up.

The subjects we used include the standard benchmark SPEC CINT2000,[4] commonly used in the SVFA literature, as well as eighteen real-world open source C/C++ projects such as PHP, FFmpeg, MySQL, and Firefox. We note that many of these subjects are extensively and frequently scanned by commercial tools such as Coverity SAVE[5] and, thus, expected to have very high quality. The sizes of these subjects range from a few thousand LoC to close to ten million with 470 KLoC on average.

Our results show that Pinpoint is quite promising: it can complete a deep scan, i.e., inlining six levels of calls, of eight million lines of code in just four hours; at the time of writing, it has found more than forty confirmed and previously unknown vulnerabilities. Some of them are from high-quality systems such as MySQL, while others are even assigned with CVE IDs[6] for their high impact on software security. Pinpoint is also quite precise with an average false positive rate around 25%. This performance is aligned with the common industrial requirement of checking millions-of-LoC code in 5-10 hours with less than 30% false positives [79, 13].

3.5.1 Comparing to Static Value-Flow Analyzer

We compared Pinpoint to the most recent and relevant work, SVF [114], based on the so-called fully-sparse value-flow graph (FSVFG). FSVFG captures memory-related data dependence by performing a flow- and context-insensitive points-to analysis with a flow-sensitive refinement. To the best of our knowledge, this is the most precise and efficient SVFA technique we can get our hands on. Both SVF and Pinpoint are targeting value flow problems, and we choose to check use-after-free, including double-free, for assessing the quality of our tool. We did not choose other properties for the assessment because, unlike most of the previous approaches, to reduce the subjectivity of evaluation, we set a high bar for "true positive": bugs confirmed by the developers of the evaluated subjects. Our experience showed that developers are much more responsive to the reports of use-after-free vulnerabilities due to its critical importance to security [16]. This allows us to complete our quantification of bug finding capability within a reasonable period of time.

[4]SPEC CINT2000 benchmarks: `https://www.spec.org/cpu2000/CINT2000`
[5]Coverity Scan: `https://scan.coverity.com/projects`
[6]Common Vulnerabilities and Exposures: `https://cve.mitre.org`

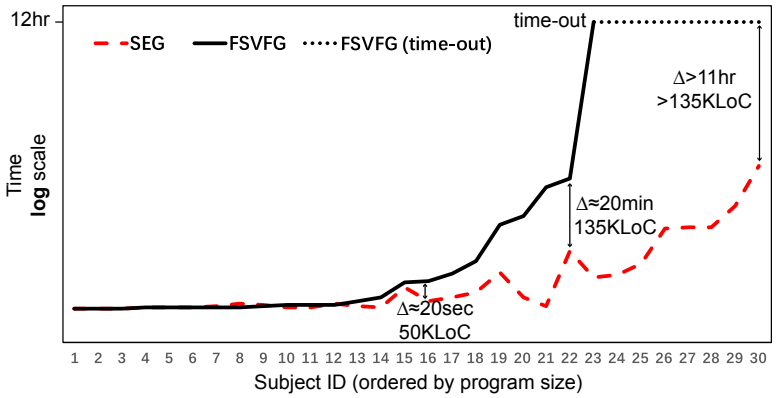

Figure 3.7: Time cost: building SEG vs. building FSVFG.

The real obstacle for scaling SVFA to millions of lines of code is the cost for building SVFG, which is the core problem solved in this chapter. Therefore, we compared the time and memory cost for building SEG and FSVFG, as well as the total time and memory consumed by Pinpoint and SVF to complete bug finding. For precision, we compared the false positive rates of both checkers. Since we cannot flood developers with all the warnings the tools report, we manually pre-screened the bug reports before sending them out.

Measuring recall is challenging as it requires the existence of a golden standard, which is hard to establish for the subjects we evaluate. We used Juliet Test Suite [14], a test suite developed by the National Security Agency's Center for Assured Software, because it provides the ground truth with a collection of known use-after-free and double-free vulnerabilities.

The number of nested levels of calling context is set to six and the timeout to twelve hours. All the experiments were performed on a server with eighty "Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz" processors and 256GB of memory running Ubuntu-16.04.

(1) Scalability. Figure 3.7 and Figure 3.8 show the comparison of the time and the memory cost between SEG and FSVFG. We observe that the two techniques perform similarly when the code size is less than 135 KLoC. For the subjects larger than 135 KLoC, the construction of FSVFG always timeouts while consuming 40-60G more memory space. Building SEG takes less than an hour, up to 400× faster.

As for the bug checking process, we observe that Pinpoint is also much more time and memory efficient than SVF. Pinpoint finished checking MySQL (2 MLoC) in 1.5 hours and Firefox (8 MLoC) in approximately 4 hours, whereas SVF took

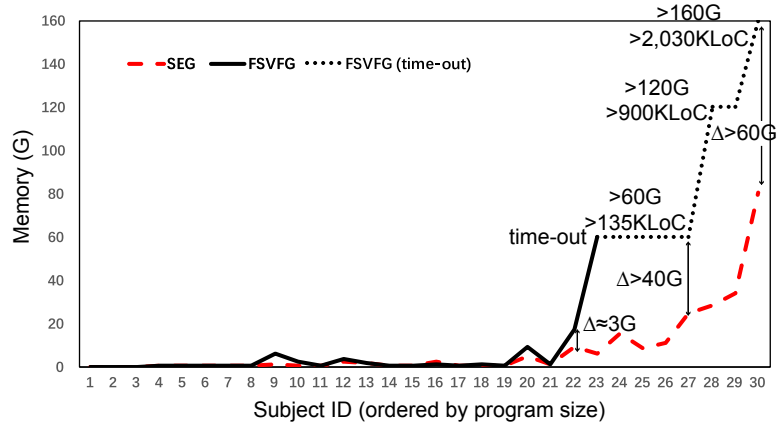

Figure 3.8: Memory cost: building SEG vs. building FSVFG.

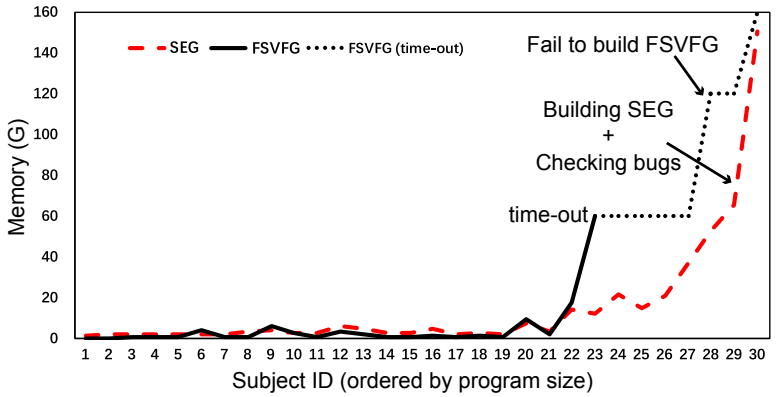

Figure 3.9: Memory cost: SEG- vs. FSVFG-based checkers.

more than twelve hours to complete fifteen out of thirty subjects and timed out on eight of them. Pinpoint also requires significantly less memory compared to SVF as shown in Figure 3.9: for the subjects larger than 135 KLoC, SVF uses 10-30G additional memory compared to Pinpoint, while unable to finish building FSVFG for these subjects.

We adopt the curve fitting approach [102] to study the observed time- and memory-complexity of Pinpoint. Figure 3.10 shows the fitting curves and their coefficients of determination R^2. $R^2 \in [0, 1]$ is a statistical measure of how close the data are to the fitting curve. The more R^2 is close to 1, the better the fitting curve is. It shows that Pinpoint's time and memory cost grow almost linearly in practice ($R^2 > 0.9$) and, thus, scale up quite gracefully.

(2) Precision and Recall. Pinpoint reported fourteen use-after-free vulnerabilities with twelve true positives and a false-positive rate of $(14 - 12)/14 = 14.3\%$. All the true positives are previously-unknown and have been confirmed by the developers.

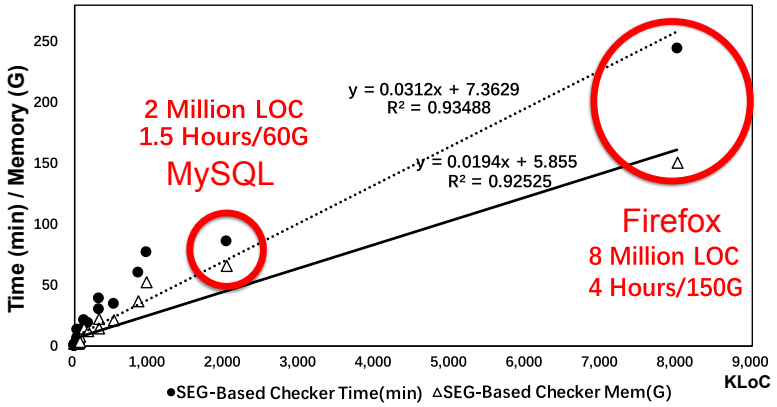

Figure 3.10: Scalability of an SEG-based checker.

A stark contrast is that Pinpoint generates very few reports in total as shown by Table 3.1, whereas SVF reports nearly 10,000 (about 1,000X more) warnings. Since we are unable to manually inspect all of them, we randomly sample a hundred warnings for inspection if a project has too many warnings. Unfortunately, SVF did not find any true positive after the manual filtering. Simply speaking, Pinpoint is more precise because our approach enables to build path-sensitive data dependence while SVF cannot do so because of the "pointer trap".

To measure if the scalability and precision of Pinpoint are achieved by sacrificing the recall, we run Pinpoint on the Juliet Test Suite, which contains 1421 use-after-free vulnerabilities, caused by 51 different types of flaws in the code. The experimental results show that Pinpoint can detect all of them.

(3) Detected Real Bugs. Pinpoint can detect vulnerabilities of high complexity for which the original developers have to use expensive methods such as the debugger to confirm. For example, Pinpoint detected a use-after-free in MySQL (Bug #87203[7]), the most popular open-source database engine, in a function of approximately 1,000 LoC. The control flow involved in the bug spans across 36 functions over 11 compiling units. Consequently, our communications with the developers met with denial twice until the final confirmation as a true bug after extensive manual code analyses.

Pinpoint also detected a use-after-free vulnerability in the code of LibICU (Bug #13301[8]), a unicode manipulation library. This library is widely used by products from hundreds of organizations and companies such as Microsoft, Apple, Google, etc. Although this library is frequently checked by mature error-detection tools such as

[7]Bug #87203: `https://bugs.mysql.com/bug.php?id=87203`
[8]Bug #13301: `http://bugs.icu-project.org/trac/ticket/13301`

Table 3.1: Results of the use-after-free checker.

Program		Size	Pinpoint		SVF	
Origin	**Name**	**(KLoC)**	**#FP**	**#Reports**	**FP Rate**	**#Reports**
SPEC CINT 2000	mcf	2	0	0	0	0
	bzip2	3	0	0	0	0
	gzip	6	0	0	100%	46
	parser	8	0	0	0	0
	vpr	11	0	0	100%	55
	crafty	13	0	0	100%†	546
	twolf	18	0	0	100%†	145
	eon	22	0	0	100%†	1324
	gap	36	0	0	0	0
	vortex	49	0	0	100%†	125
	perkbmk	73	0	0	100%	13
	gcc	135	0	0	0	0
Open Source	webassembly	23	0	1	100%†	902
	darknet	24	0	0	100%†	152
	html5-parser	31	0	0	100%	32
	tmux	40	0	0	100%†	2041
	libssh	44	0	1	100%	102
	goacess	48	0	1	100%†	312
	shadowsocks	53	0	2	100%†	1972
	swoole	54	0	0	100%†	534
	libuv	62	0	0	0	0
	transmission	88	0	1	100%†	802
	git	185	0	0	NA	NA
	vim	333	0	0	NA	NA
	wrk	340	0	0	NA	NA
	libicu	537	0	1	NA	NA
	php	863	0	0	NA	NA
	ffmpeg	967	0	0	NA	NA
	mysql	2,030	1	5	NA	NA
	firefox	7,998	1	2	NA	NA

† We only inspect one hundred randomly-selected reports.

Coverity SAVE, the bug has been hidden for more than ten years. This vulnerability is serious enough to deserve its CVE ID: CVE-2017-14952.

In total, we have detected hundreds of vulnerabilities from many open-source projects, including famous software systems like MySQL, FireFox, Python, Apache and OpenSSL, as well as fundamental libraries like LibSSH and LibICU.

3.5.2 Study of the Taint Analysis

As a general framework, Pinpoint should enable the same performance characteristics for other types of bug finding tasks that it can support. For this purpose, we also evaluated two additional checkers for taint issues as described in Section 3.4.

Table 3.2: Results of the SEG-based taint analysis.

Checkers	Memory	Time	#FP/#Reports
Path Traversal Vuln.	43.1G	1.4hr	11/56
Data Transmission Vuln.	52.6G	1.5hr	24/92

Table 3.3: Results of Infer and Clang.

Program	Size (KLoC)	Infer		Clang	
		Time (min)	#FP/#Rep	Time (min)	#FP/#Rep
webassembly	23	0.1	0/0	0.5	0/0
darknet	24	2.5	0/0	1.4	0/0
html5-parser	31	NA	NA	0.2	0/0
tmux	40	1.0	5/5	1.0	6/6
libssh	44	0.1	0/0	0.2	1/1
goaccess	48	0.5	4/4	0.3	0/1
shadowsocks	53	NA	NA	NA	NA
swoole	54	NA	NA	0.5	0/0
libuv	62	0.5	1/1	0.2	0/0
transmission	88	1.0	0/0	0.5	0/0
git	185	2.5	3/3	1.4	2/2
vim	333	NA	NA	1.4	0/0
wrk	340	NA	NA	2.5	0/0
libicu	537	3.3	8/8	2.6	0/0
php	863	NA	NA	6.9	4/4
ffmpeg	967	21.1	1/1	3.3	0/0
mysql	2030	42.6	13/13	15.8	6/7
firefox	7998	NA	NA	54.0	5/5
Total			**35/35**		**24/26**

NA means we fail to run Clang or Infer on the benchmark programs.

The corresponding evaluation results are summarized in the Table 3.2. Because of the page limits, we only present the memory and time cost for checking MySQL (2 MLoC, typical code size in industry). This cost is similar to that of use-after-free. Like in the previous taint analysis work [6], we have not modeled the sanitization operations in our analysis. Thus, a report is regarded as a false positive only if we can manually identify an infeasible value-flow path, which leads to a false positive rate of 23.6%.

3.5.3 Comparing to Other Static Analyzers

To better understand the performance of Pinpoint in comparison to other types of bug finding techniques, we also ran Pinpoint against two prominent and mature open-source static bug detection tools, Infer and Clang, on finding the use-after-free vulnerabilities. The results are reported in Table 3.3. Our evaluation shows both

Clang and Infer run faster compared to Pinpoint. The primary reason is that both Infer and Clang confine their activities within each compilation unit and do not fully track path correlations. This is at the cost of generating more false warnings and of the failure of finding bugs across multiple compilation units. As Table 3.3 shows, if we allow the concurrent analysis of fifteen threads, both tools can finish checking within one hour. However, all of the thirty-five use-after-free reports of Infer are false positives. Only two of the twenty-six warnings reported by Clang are true positives, which are also reported by Pinpoint.

3.6 Conclusion

We have described Pinpoint, embodying a holistic design of sparse value-flow analysis that allows us to escape from the pointer trap and, thus, simultaneously achieve precision and observed linear scalability for millions of lines of code. Pinpoint has discovered hundreds of vulnerabilities, confirmed by developers of many well-known systems and code libraries. Pinpoint is promising in providing industrial-strength capability in static bug finding.

Chapter 4

Scaling up Sparse Value-Flow Analysis for Multiple Checkers

4.1 Introduction

Sparse value-flow analysis (SVFA) [106, 21, 114, 75], which tracks how values are stored and loaded in a program, underpins the inspection of a very broad category of software properties, such as memory safety (e.g., null dereference, double free, etc.), resource usage (e.g., memory leak, file usage, etc.), and security properties (e.g., the use of tainted data). In addition, there are a large and growing number of domain-specific value-flow properties. For instance, mobile software requires that the personal information cannot be passed to an untrusted code [6], and, in web applications, tainted database queries are not allowed to be executed [116]. Fortify, a commercial static code analyzer, checks nearly ten thousand value-flow properties from hundreds of unique categories. Value-flow properties exhibit a very high degree of versatility, which poses great challenges to the effectiveness of general-purpose program analyzers.

4.1.1 The Extensional Scalability Problem

Faced with such a massive number of properties and the need of extension, existing approaches, such as Fortify, Clang, and Infer, provide a customizable framework together with a set of property interfaces that enable the quick customization for new properties. For instance, Clang uses a symbolic-execution engine such that, at every statement, it invokes the callback functions registered for the properties. These callback functions are overwritten by the property-checker writers to collect the

symbolic-execution results, such as the symbolic memory and the path conditions, so that we can judge the presence of any property violation at the statement. Despite the existence of many Clang-like frameworks, when high precision like path-sensitivity is required, existing static analyzers still cannot scale well with respect to a large number of properties to check, which we refer to as the *extensional scalability issue*. For example, our evaluation shows that Clang cannot path-sensitively check twenty properties for many programs in ten hours. Pinpoint, our analyzer introduced in the previous chapter, exhausted 256GB of memory for only eight properties.

We observe that a major factor for the extensional scalability issue is that, in the conventional extension mechanisms, such as that of Clang, the core static analysis engine is oblivious to the properties being checked. Although the property obliviousness gives the maximum flexibility and extensibility to the framework, it also prevents the core engine from utilizing the property-specific analysis results for optimization. This scalability issue is slightly alleviated by a class of approaches that are property-aware and demand-driven [43, 9, 70]. These techniques are scalable with respect to a small number of properties because the core engine can skip certain program statements by understanding what statements are relevant or irrelevant to the properties. However, in these approaches, the semantics of properties are also opaque to each other. As a result, when the number of properties grows very large, the performance of the demand-driven approaches will quickly deteriorate because property-irrelevant program statements become fewer and fewer, such as in the case of Pinpoint. To the best of our knowledge, the number of literature specifically addressing the extensional scalability issue is very limited. Readers can refer to Section 2.2.2 for a detailed discussion.

4.1.2 Conquering the Extensional Scalability Problem

In this work, we advocate an inter-property-aware design to relax the property-property and the property-engine opaqueness so that the core static analysis engine can exploit the mutual synergy among different properties for optimization. To check a value-flow property, instead of conforming to conventional callback interfaces, property-checker writers of our framework need to explicitly declare a simple property specification, which picks out source and sink values, respectively, as well as the predicate over these values for the satisfaction of the property. For instance, for a null deference property, our property model only requires the checker writers to indicate where a null pointer may be created and where the null dereference may happen using

pattern expressions, as well as a simple predicate that constrains the propagation of the null pointer. Surprisingly, given a set of properties specified in our property model, our static analyzer can automatically understand the overlaps and inconsistencies of the properties to check. Based on the understanding, before analyzing a program, we can make dedicated analysis plans so that, at runtime, the analyzer can share the analysis results on path-reachability and path-feasibility among different properties for optimization. The optimization allows us to significantly reduce redundant graph traversals and unnecessary invocations of the SMT solver, two critical performance bottlenecks of conventional approaches. We provide some examples in Section 4.2 to illustrate our approach.

We have implemented our approach, named Catapult, which is a new demand-driven and compositional static analyzer with the precision of path-sensitivity. Like a conventional compositional analysis [125], our implementation allows us to concurrently analyze functions that do not have calling relations. In Catapult, we have included all C/C++ value-flow properties that Clang checks by default. In the evaluation, we compared Catapult to three state-of-the-art bug-finding tools, Pinpoint, Clang, and Infer, using a standard benchmark and ten popular industrial-sized software systems. The experimental results demonstrate that Catapult is more than 8× faster than Pinpoint but consumes only 1/7 of the memory. It is as efficient as Clang and Infer in terms of both time and memory cost but is much more precise. Such promising scalability of Catapult is not achieved by sacrificing the capability of bug finding. In our experiments, although the benchmark software systems have been checked by numerous free and commercial tools, Catapult is still able to detect many previously-unknown bugs, in which thirty-nine have been fixed by the developers and four have been assigned CVE IDs due to their security impact. In summary, we make the following contributions in this chapter:

- An inter-property-aware design for checking value-flow properties, which mitigates the extensional scalability issue.
- A series of cross-property optimization rules that can be made use of for general SVFA frameworks.
- A detailed implementation and a systematic evaluation that demonstrates our high scalability, precision, and recall.

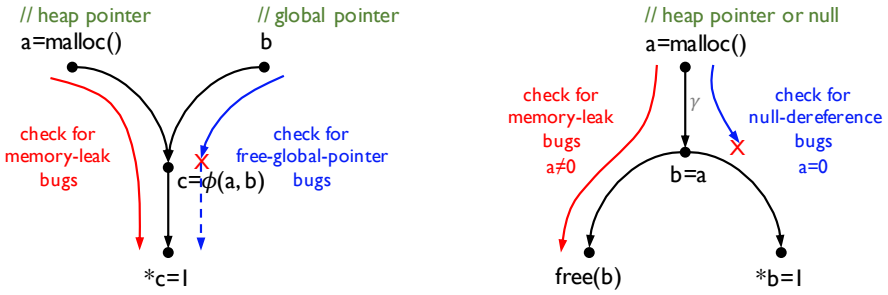

Figure 4.1: Path overlapping and contradiction among different properties on the value-flow graph.

4.2 Overview

The key factor that allows us to conquer the extensional scalability problem is the exploitation of the mutual synergy among different properties. In this section, we first use two simple examples to illustrate this mutual synergy and then provide a running example used in the whole chapter.

4.2.1 Mutual Synergy

We observe that the mutual synergy among different properties are primarily in the forms of path overlapping and path contradiction.

In Figure 4.1(a), to check the memory-leak bug, we need to track value flows from the newly-created heap pointer a to check if the pointer will be freed.[1] To check the free-global-pointer bug, we track value flows from the global variable b to check if it will be freed.[2] As illustrated in the figure, the value-flow paths to search for these two bugs overlap from the vertex $c{=}\phi(a,b)$ to the vertex $^*c{=}1$. Being aware of the overlap, when traversing the graph from the vertex $a{=}malloc()$ for the memory-leak bug, we record that the vertex $c{=}\phi(a,b)$ cannot reach any "free" operation. Therefore, when checking the free-global-pointer bug, we can use this recorded information to immediately stop the graph traversal at the vertex $c{=}\phi(a,b)$, thereby avoiding redundant graph traversals.

[1] We say a pointer p is "freed" if it is used in the function call $free(p)$. We will detail how to use the value-flow information to check bugs later.

[2] Freeing a pointer pointing to non-heap memory (e.g., memory allocated by global variables) is buggy. See details in `https://cwe.mitre.org/data/definitions/590.html`.

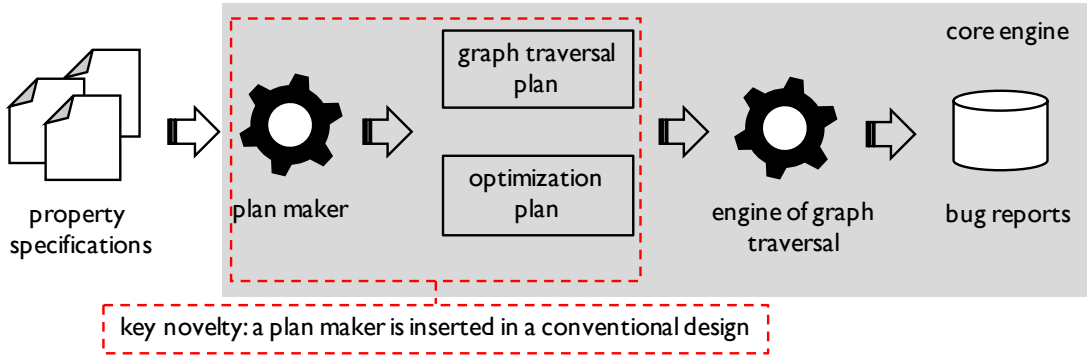

Figure 4.2: The workflow of our approach.

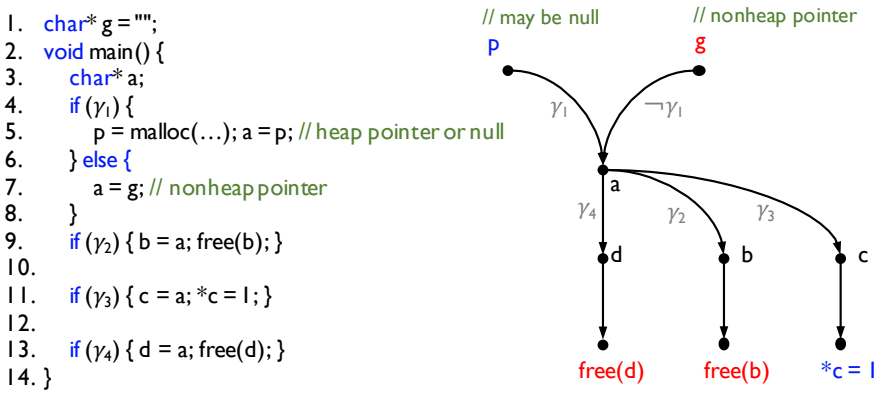

Figure 4.3: An example to illustrate our method.

In Figure 4.1(b), to check the memory-leak bug, we track value flows from the newly-created pointer a to where it is freed. To check the null-dereference bug, considering that the function *malloc* may return a null pointer when the memory allocation fails, we track the value flows from the same pointer a to where it is dereferenced. The two properties have an inconsistent constraint: the former requires $a{\neq}0$ for a to be a valid heap pointer while the latter requires $a{=}0$ for a to be a null pointer. Being aware of this inconsistency, when traversing the graph for checking the null-dereference bug, we check and record if the path condition γ of the path from the vertex $a{=}malloc()$ to the vertex $b{=}a$ conflicts with the null pointer condition $a{=}0$. If the path condition γ is satisfiable but conflicts with the null pointer condition $a{=}0$, i.e., the conjunction $\gamma{\wedge}a{=}0$ is unsatisfiable, we can conclude that the conjunction $\gamma{\wedge}a{\neq}0$ must be satisfiable without an expensive constraint-solving procedure when checking the memory-leak bug.

4.2.2 A Running Example

Figure 4.3 shows a running example using the value-flow graph where we check the null-deference and the free-global-pointer bugs following the workflow illustrated in Figure 4.2. Given a program, we first follow the previous work [115, 21, 106] to build the value-flow graph in order to check the two properties with the precision of path-sensitivity. Here, path-sensitivity means that when searching paths on the value-flow graph, we invoke an SMT solver to solve path conditions and other property-specific constraints to prune infeasible paths.

(1) The Property Specifications. The users of our framework need to declaratively specify the value-flow properties, which consists of the simple descriptions of the sources, the sinks, and the predicates for triggering the bug. For instance, the specifications of the aforementioned two properties are described by the following two quadruples, respectively:

$$\mathsf{prop}\ \textit{null-deref} := (v = \textit{malloc(\_)}; \_ = *v, *v = \_; v = 0; \mathsf{never})$$

$$\mathsf{prop}\ \textit{free-glob-ptr} := (\textit{glob}; \textit{free(v)}; \textit{true}; \mathsf{never})$$

Separated by the semicolons, the first and second components denote the descriptors of the source and the sink, respectively, specified using pattern expressions to represent the values used or defined in some program statements. The "don't-care" values are written as underscores. In the running example, the source values of the properties *null-deref* and *free-glob-ptr* are the return pointer of the function *malloc* and the global pointer g, respectively. The sink value of the property *null-deref* is the dereferenced value c at the statement *c=1. The sink values of the property *free-glob-ptr* are the freed values at the statements *free(b)* and *free(d)*.

The third component is a property-specific constraint, representing the triggering condition of the bug. In our example, the constraint of the property *null-deref* is $v = 0$, meaning that the value on a value-flow path should be a null pointer. The constraint of the property *free-glob-ptr* is *true*, meaning that the value on a value-flow path is unconstrained.

The built-in predicate "never" means that value-flow paths between the specified sources and sinks should never be feasible. Otherwise, a bug exists.

(2) The Core Static Analysis Engine. Given these declarative specifications, our core engine automatically makes analysis plans before the analysis begins, including both the graph traversal plan and the optimization plan. In the example,

we make the following optimization plans: (1) checking the property *free-glob-ptr* before the property *null-deref*; (2) when traversing the graph for the property *free-glob-ptr*, we record the vertices that cannot reach any sink vertex of the property *null-deref*. The graph traversal plan in the example is trivial, which is to perform a depth-first search on the value-flow graph from every source vertex of the two properties.

In Figure 4.3, when traversing the value-flow graph from the global pointer g to check the property *free-glob-ptr*, the core engine visits all vertices except the vertex p to look for "free" operations. According to the optimization plan, during the graph traversal, we record that the vertices b and d cannot reach any dereference operation.

To check the property *null-deref*, we traverse the value-flow graph from the vertex p. When visiting the vertex b and the vertex d, since the previously-recorded information tells us that they cannot reach any sink vertices, we prune the subsequent paths from the two vertices.

It is noteworthy that if we check the property *null-deref* before the property *free-glob-ptr*, we only can prune one path from the vertex c for the property *free-glob-ptr* based on the results of the property *null-deref*. We will further explain the rationale of our analysis plans in the following sections.

4.3 Value-Flow Properties

This section provides a specification model for value-flow properties. We first define the specification and then provide several examples to show what properties can be described using the specification. The specification sets the foundation for our optimized analysis for multiple value-flow properties.

4.3.1 Property Specification

We define the property specification with two motivations. First, we observe that many property-specific constraints play a significant role in performance optimization. The specific constraints of one property can be used to optimize checking of not just the property itself, but also of other properties being checked together.

Second, despite many studies on value-flow analysis [75, 106, 115, 114, 21], we still have a lack of general and extensible specification models that can maximize

Table 4.1: Pattern expressions used in the specification.

p	::=	:: **patterns**
	$\mid p_1, p_2, \cdots$	:: **pattern list**
	$\mid v_0 = sig(v_1, v_2, \cdots)$	:: **call**
	$\mid v_0 = *v_1$	:: **load**
	$\mid *v_0 = v_1$	:: **store**
	$\mid v_0 = v_1$	:: **assign**
	$\mid glob$	:: **globals**
v	::=	:: **symbol**
	$\mid sig$	:: **character string**
	$\mid \_$	:: **uninterested value**

Examples:

$v = malloc(\_)$	the return value of any statement calling *malloc*;
$\_ = send(\_,v,\_,\_)$	the 2nd argument of any statement calling *send*;
$\_ = *v$	the dereferenced value at every load statement;

the opportunities of sharing analysis results across the processes of checking different properties. Some of the existing studies only focus on checking a specific property (e.g., memory leak [115]), while others adopt different specifications to check the same value-flow property (e.g., double free [106, 21]).

In a similar style to existing approaches [73, 115, 106], we assume that the code of a program is in static single assignment (SSA) form, where every variable has only one definition [29]. As defined below, we model a value-flow property as an aggregation of value-flow paths.

Definition 4.1 (Value-Flow Property). A value-flow property, x, is a quadruple: $\mathsf{prop}\ x := (\mathsf{src}; \mathsf{sink}; \mathsf{psc}; \mathsf{agg})$, where

- src and sink are two pattern expressions (Table 4.1) that specify the sources and the sinks of the value-flow paths to track.
- psc is a first-order logic formula, representing the property-specific constraint that every value on the value-flow path needs to satisfy.
- $\mathsf{agg} \in \{\mathsf{never}, \mathsf{never\text{-}sim}, \mathsf{must}, \cdots\}$ is an extensible predicate that determines how to aggregate value-flow paths to check the specified property.

4.3.2 Property Examples

In practice, we can use the quadruple defined above to specify a wide range of value-flow properties. As discussed below, we put the properties into three categories, which are checked by aggregating a single, two, or more value-flow paths, respectively.

(1) Single-Path Properties. We can check many program properties using a single value-flow path, such as the properties, *null-deref* and *free-glob-ptr*, defined in Section 4.2.2, as well as a broad range of taint issues that propagate a tainted object to a program point consuming the object [33].

(2) Double-Path Properties. A wide range of bugs happen in a program execution because two program statements (e.g., two statements calling the function *free*) consecutively operate on the same value (e.g., a heap pointer). Typical examples include the use-after-free bug, a general form of the double-free bug, as well as the ones that operate on expired resources such as a closed file descriptor or a closed network socket. We check them using two value-flow paths from the same source value. As an example, the specification for checking the double-free bugs can be specified as

$$\mathsf{prop}\ \textit{double-free} := (v = \textit{malloc(\_)}; \textit{free(v)}; v \neq 0; \mathsf{never\text{-}sim})$$

In the specification, the property-specific constraint $v \neq 0$ requires the initial value (or equivalently, all values) on the value-flow path is a valid heap pointer. This is because $v = 0$ means the function *malloc* fails to allocate memory and returns a null pointer. In this case, the "free" operation is harmless. The aggregate predicate "never-sim" means that two value-flow paths from the same pointer should never occur simultaneously. In other words, there is no control-flow path that goes through two "free" operations on the same heap pointer. Otherwise, a double-free bug exists.

In Figure 4.3, for the two value-flow paths from the vertex p to the two "free" operations, we check the constraint $(\gamma_1 \wedge \gamma_2) \wedge (\gamma_1 \wedge \gamma_4) \wedge (p \neq 0)$ to find double-free bugs. Here, $(\gamma_1 \wedge \gamma_2)$ and $(\gamma_1 \wedge \gamma_4)$ are the path conditions of the two paths.

(3) All-Path Properties. Many bugs happen because we do not properly handle a value in all program paths. For instance, a memory-leak bug happens if there exists a feasible program path where we do not free a heap pointer. Other typical examples include many types of resource leaks such as the file descriptor leak and the socket leak. We check them by aggregating all value-flow paths from the same source value. As an example, we write the following specification for checking

memory leaks:

$$\mathsf{prop}\ \textit{mem-leak} := (v = \textit{malloc(\_)}; \textit{free(v)}; v \neq 0; \mathsf{must})$$

Compared to the property *double-free*, the only difference in the specification is the aggregate predicate. The aggregate predicate "must" means that the value-flow path from a heap pointer must be able to reach a "free" operation. Otherwise, a memory leak exists in the program.

In Figure 4.3, for the value-flow paths from the vertex p to the two "free" operations, we can check the disjunction of their path conditions, i.e., $\neg((\gamma_1 \wedge \gamma_2) \vee (\gamma_1 \wedge \gamma_4)) \wedge \gamma_1 \wedge (p \neq 0)$, to determine if a memory leak exists. Here, $(\gamma_1 \wedge \gamma_2)$ and $(\gamma_1 \wedge \gamma_4)$ are the path conditions of these two paths, respectively. The additional γ_1 is the condition on which the heap pointer is created.

4.4 Inter-property-aware Analysis

Given a number of value-flow properties specified as the quadruples defined in Definition 4.1, our inter-property-aware static analyzer searches the value-flow paths and checks bugs based on the path conditions, the property-specific constraint, and the aggregate predicate. In this chapter, we concentrate on how to exploit the mutual synergy arising from the interactions of different properties to improve the searching efficiency of value-flow paths.

4.4.1 A Naïve Static Analyzer

For multiple value-flow properties, a naïve static analyzer checks them independently in a demand-driven manner. As illustrated by Algorithm 4.1, for each value-flow property, the static analyzer traverses the value-flow graph from each of the source vertices. At each step of the graph traversal, we check if the property-specific constraint psc is satisfiable with respect to the current path condition. If it is not satisfiable, we can stop the graph traversal along the current path. This path-pruning process is illustrated in the shaded part of Algorithm 4.1, which is a critical factor to improve the performance.

```
Input: the sparse value-flow graph of a progam to check
Input: a set of value-flow properties to check
Output: paths between sources and sinks for each property
foreach property in the input property set do
    foreach source v in its source set do
        while visit v' in the depth-first search from v do
            π ← the current path that ends with v';
            if psc cannot be satisfied with respect to the path condition of the
            path π then
                stop the search from v';
```

Algorithm 4.1: The naïve static analyzer.

The key optimization opportunities come from the observation that the properties to check usually introduce overlaps and inconsistencies during the graph traversal, which cannot be exploited if they are independently checked as in the naïve approach.

4.4.2 Optimized Intra-procedural Analysis

As summarized in Table 4.2, given the property specifications, our inter-property-aware static analysis engine carries out two types of optimizations when traversing the value-flow graph: the first aiming at pruning paths and the second focusing on sharing paths when multiple properties are being checked. Each row of the table is a rule describing the specific precondition, the corresponding optimization, as well as its benefit. For the clarity of the discussion, we explain the rules in the context of processing a single-procedure program, followed by the discussion on the inter-procedural analysis in the next subsection.

Given the property specifications, we adopt Rules 1 – 4 in Table 4.2, the optimization plans, to facilitate the path pruning.

(1) Ordering the Properties (Rule 1). Given a set of properties with different source values, we need to determine the order in which they are checked. While we leave the finding of the perfect order that guarantees the optimal optimization to our future work, we observe that a random order can significantly affect the effectiveness of the path pruning and must be circumvented.

Let us consider the example in Figure 4.3 again. In Section 4.2.2, we have explained that if the property *free-glob-ptr* is checked before the property *null-deref*, we can prune the two paths from the vertex b and the vertex d when checking the latter. However, if we flip the checking order, only one path from the vertex c

Table 4.2: Rules of making analysis plans for a pair of properties.

Optimization Plans

prop $x := (\mathsf{src}_1; \mathsf{sink}_1; \mathsf{psc}_1; \mathsf{agg}_1)$ and prop $y := (\mathsf{src}_2; \mathsf{sink}_2; \mathsf{psc}_2; \mathsf{agg}_2)$, $\mathsf{src}_1 \neq \mathsf{src}_2$

ID	Rule Name	Precondition	Plan	Benefit
1	property ordering	$\#\mathsf{sink}_1 > \#\mathsf{sink}_2$	check x before y	more chances to prune paths
2	result recording	check x before y	record vertices that cannot reach sink_2	prune paths at a vertex
3		check x before y, $\mathsf{psc}_1 = \mathsf{psc}_2$	record unsat cores that conflict with psc_2	prune paths if going through a set of edges
4		check x before y, $\mathsf{psc}_1 \neq \mathsf{psc}_2$	record interpolants that conflict with psc_2	

Graph Traversal Plans

prop $x := (\mathsf{src}_1; \mathsf{sink}_1; \mathsf{psc}_1; \mathsf{agg}_1)$ and prop $y := (\mathsf{src}_2; \mathsf{sink}_2; \mathsf{psc}_2; \mathsf{agg}_2)$, $\mathsf{src}_1 = \mathsf{src}_2$

ID	Rule Name	Precondition	Plan	Benefit
5	traversal merging	-	search from src_1 for both properties	sharing path conditions
6	psc-check ordering	$\mathsf{psc}_1 \wedge \mathsf{psc}_2 = \mathsf{psc}_1$	check psc_1 first	if satisfiable, so is psc_2
7		$\mathsf{psc}_1 \wedge \mathsf{psc}_2 \neq \mathit{false}$	check $\mathsf{psc}_1 \wedge \mathsf{psc}_2$	if satisfiable, both psc_1 and psc_2 can be satisfied
8		$\mathsf{psc}_1 \wedge \mathsf{psc}_2 = \mathit{false}$	check psc_1 first	if unsatisfiable, psc_2 can be satisfied

can be pruned. This is because, when checking the property *null-deref*, the core engine records that the vertex c cannot reach any sinks specified by the property *free-glob-ptr*.

Intuitively, what causes the fluctuation in the number of prunable paths is that the number of the "free" operations is more than the dereference operations in the value-flow graph. That is, the more sink vertices we have in the value-flow graph, the fewer paths we can prune for the property. Inspired by this intuition, the order of checking the properties is arranged according to the number of sink vertices. That is, the more sink vertices a property has in the value-flow graph, the earlier we check this property.

(2) Recording Sink-Reachability (Rule 2). Given a set of properties $\{\mathsf{prop}_1, \mathsf{prop}_2, \cdots\}$, when checking the property prop_i by traversing the value-flow graph, we record if each visited vertex may reach a sink vertex of the property $\mathsf{prop}_j (j \neq i)$. With the recorded information, when checking the property $\mathsf{prop}_j (j \neq i)$ and visiting a vertex that cannot reach any of its sinks, we prune the paths from the vertex. Section 4.2.2 illustrates the method.

(3) Recording the Checking Results of Property-Specific Constraints (Rules 3 & 4). Given a set of properties $\{\mathsf{prop}_1, \mathsf{prop}_2, \cdots\}$, when we check the property prop_i by traversing the value-flow graph, we record the path segments, i.e., a set of edges, that conflict with the property-specific constraint psc_j of the property $\mathsf{prop}_j (j \neq i)$. When checking the property $\mathsf{prop}_j (j \neq i)$, we prune the paths that include the path segments.

Let us consider the running example in Figure 4.3 again. When traversing the graph from the vertex g to check the property *free-glob-ptr*, the core engine records that the condition of the edge from the vertex a to the vertex c, i.e., $a \neq 0$, conflicts with the property-specific constraint of the property *null-deref*, i.e., $a = 0$. With this information, when checking the property *null-deref*, we can prune the subsequent path after the vertex c.

Thanks to the advances in the area of clause learning [12], we are able to efficiently compute some reusable facts when using SMT solvers to check path conditions and property-specific constraints. Specifically, we compute two reusable facts when a property-specific constraint psc_i conflicts with the current path condition pc.

When $\mathsf{pc} \wedge \mathsf{psc}_i$ is unsatisfiable, we record the unsatisfiable core [34], which is a set of Boolean predicates in the path condition pc, e.g., $\{\gamma_1, \gamma_2, \cdots\}$, such that $\gamma_1 \wedge \gamma_2 \wedge \cdots \wedge \mathsf{psc}_i = \mathit{false}$. Since the path condition pc is the conjunction of the edge constraint on the value-flow path, each predicate γ_i corresponds to the condition of an edge ϵ_i on the value-flow graph. Thus, we can record an edge set $E = \{\epsilon_1, \epsilon_2, \cdots\}$, which conflicts with the property-specific constraint psc_i. When checking the other property with the same property-specific constraint, if a value-flow path contains these recorded edges, we can prune the remaining paths.

In addition to the unsatisfiable cores, we also can record the interpolation constraints [23], which are even reusable for properties with a different property-specific constraint. In the above example, assume that the property-specific constraint psc_i is $a = 0$ and the predicate set $\{\gamma_1, \gamma_2, \cdots\}$ is $\{a+b > 3, b < 0\}$. In the constraint solving phase, an SMT solver can refute the satisfiability of $(a + b > 3) \wedge (b < 0) \wedge (a = 0)$ by finding an interpolant γ' such that $(a + b > 3) \wedge (b < 0) \Rightarrow \gamma'$ but $\gamma' \Rightarrow \neg(a = 0)$. In the example, the interpolant γ' is $a > 3$, which provides a detailed explanation why the γ set conflicts with the property-specific constraint $a = 0$. In addition, the interpolant also indicates that the γ set conflicts with many other constraints such as $a < 0$ and $a < 3$. Thus, given a property whose specific constraint conflicts with the interpolation constraint, it is sufficient to conclude that any value-flow path passing through the edge set E can be pruned.

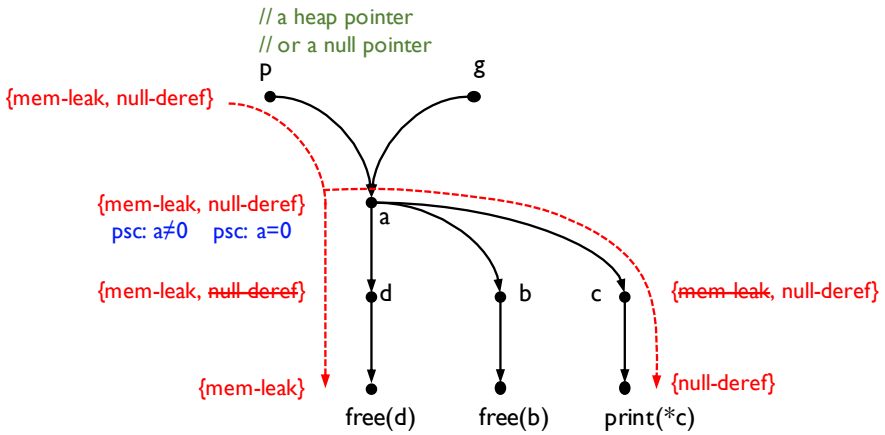

Figure 4.4: Merging the graph traversal.

Different from the optimization plan that aims to prune paths, The graph traversal plan is to provide strategies of sharing paths among different properties.

(4) Merging the Graph Traversal (Rule 5). We observe that many properties actually share the same or a part of source vertices and even the same sink vertices. If the core engine checks each property one by one, it will repetitively traverse the graph from the same source vertex for different properties. Therefore, our graph traversal plan merges the path searching processes for different properties.

As an example, in Figure 4.3, since the vertex p may represent either a heap pointer or a null pointer, checking both the property *null-deref* and the property *mem-leak* needs to traverse the graph from the vertex p. Figure 4.4 illustrates how the merged traversal is performed. That is, we maintain a property set during the graph traversal to record what properties the current path contributes to. Whenever visiting a vertex, we check if a property needs to be removed from the property set. For instance, at the vertex d, we may remove the property *null-deref* from the property set if we can determine the vertex d cannot reach any of its sinks. When the property set becomes empty, the graph traversal stops immediately.

(5) Ordering the Checks of Property-Specific Constraints (Rules 6 – 8). Since the graph traversals are merged for different properties, at a vertex, e.g., a in Figure 4.4, we have to check multiple property-specific constraints, e.g., $a \neq 0$ for the property *mem-leak* and $a = 0$ for the property *null-deref*, with respect to the path condition. In a usual manner, we have to invoke an expensive SMT solver to check each property-specific constraint, significantly affecting the analysis performance when there are many properties to check. We mitigate this issue by utilizing various

relations between the property-specific constraints, so that we can reuse SMT-solving results and reduce the invocations of the SMT solver.

Given two property-specific constraints, psc_1 and psc_2, we consider all three possible relations between them: $\mathsf{psc}_1 \wedge \mathsf{psc}_2 = \mathsf{psc}_1$, $\mathsf{psc}_1 \wedge \mathsf{psc}_2 \neq \mathit{false}$, and $\mathsf{psc}_1 \wedge \mathsf{psc}_2 = \mathit{false}$. Since the property-specific constraints are often simple, these relations are easy to compute. These relations make it possible to check both psc_1 and psc_2 by invoking an SMT solver only once.

The first relation, $\mathsf{psc}_1 \wedge \mathsf{psc}_2 = \mathsf{psc}_1$, implies that any solution of the constraint psc_1 also satisfies the constraint psc_2. In this case, we first check if the constraint psc_1 conflicts with the current path condition pc by solving the conjunction, $\mathsf{pc} \wedge \mathsf{psc}_1$. If it is satisfiable, we can conclude that the conjunction, $\mathsf{pc} \wedge \mathsf{psc}_2$, is also satisfiable.

The second relation, $\mathsf{psc}_1 \wedge \mathsf{psc}_2 \neq \mathit{false}$, implies that there exists a solution that satisfying both the constraint psc_1 and the constraint psc_2. In this case, we first check the conjunction, $\mathsf{pc} \wedge \mathsf{psc}_1 \wedge \mathsf{psc}_2$. If it is satisfiable, we can conclude that both of the constraints, psc_1 and psc_2, are satisfiable with respect to the path condition.

The third relation, $\mathsf{psc}_1 \wedge \mathsf{psc}_2 = \mathit{false}$, implies that there does not exist any solution that satisfies both the constraint psc_1 and the constraint psc_2. In this case, we check any of the constraints, psc_1 and psc_2, first. If the current path is feasible but the conjunction $\mathsf{pc} \wedge \mathsf{psc}_1$ is not satisfiable, we can conclude that the conjunction $\mathsf{pc} \wedge \mathsf{psc}_2$ can be satisfied without invoking SMT solvers.

4.4.3 Modular Inter-procedural Analysis

Scalable program analyses need to exploit the modular structure of a program. They build function summaries, which are reused at different calling contexts [27, 125]. In Catapult, we can seamlessly extend our optimized intra-procedural analysis to modular inter-procedural analysis by exploring the local value-flow graph of each function and then stitching the local paths together to generate complete value-flow paths. In what follows, we explain our design of the function summaries.

In our analysis, for each function, we build three kinds of value-flow paths as the function summaries defined below. Intuitively, these summaries describe how function boundaries, i.e., formal parameters and return values, partition a complete value-flow path. Using the property *double-free* as an example, a complete value-flow path from the vertex p to the vertex *free(b)* in Figure 4.5 is partitioned to a sub-path

from the vertex p to the vertex *ret p* by the boundary of the function *xmalloc*. This sub-path is an output summary of the function *xmalloc* as defined below.

Definition 4.2 (Transfer Summary). A transfer summary of a function f is a value-flow path from one of its formal parameters to one of its return values.

Definition 4.3 (Input Summary). An input summary of a function f is a value-flow path from one of its formal parameters to a sink value in the function f or in the callees of the function *f*.

Definition 4.4 (Output Summary). An output summary of a function f is a value-flow path from a source value to a return value of the function. The source value is in the function f or in the callees of the function *f*.

After generating the function summaries, to avoid separately storing them for different properties, each function summary is labeled with a bit vector to record what properties it is built for. Assume that we need to check there properties, i.e., *null-deref*, *double-free*, and *mem-leak*, in Figure 4.5. We assign three bit vectors, $0b001$, $0b010$, and $0b100$, to the three properties as their identities, respectively. As explained before, all three properties regard the vertex p as the source. The sink vertices for checking the properties *double-free* and *mem-leak* are the vertices *free(b)* and *free(u)*. There are no sink vertices for the property *null-deref*. According to Definitions 4.2–4.4, we generate the following function summaries:

Function	**Summary Path**	**Label**	**Type**
xmalloc	(p, *ret p*)	$0b111$	output
xfree	(u, *ret u*)	$0b111$	transfer
	(u, *free(u)*)	$0b110$	input

The summary (p, *ret p*) is labeled with $0b111$ because all three properties regard p as the source. The summary (u, *ret u*) is also labeled with $0b111$ because the path does not contain any property-specific vertices and, thus, may be used to check all three properties. The summary (u, *free(u)*) is only labeled with $0b110$ because we do not regard the vertex *free(u)* as a sink of the property *null-deref*.

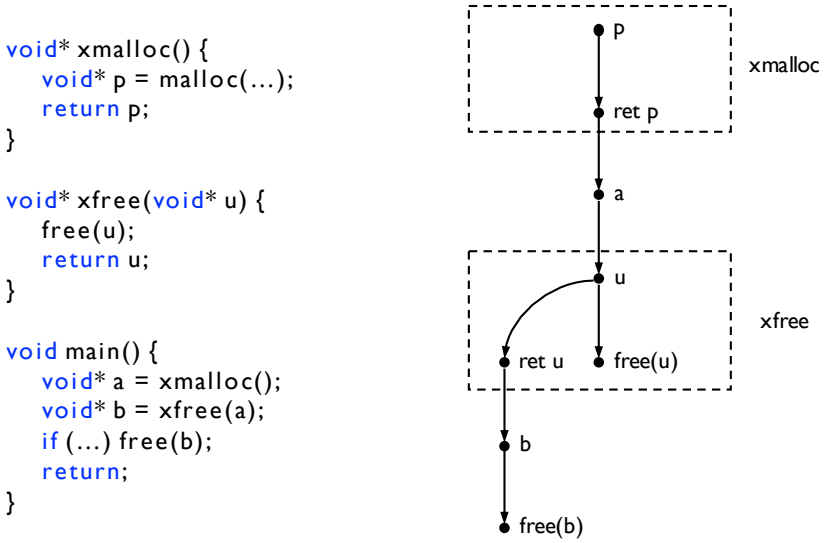

Figure 4.5: An example to show the inter-procedural analysis.

When analyzing the main function, we concatenate its intra-procedural paths with summaries from its callees to generate a complete path. For example, a concatenation is illustrated below and its result is labeled by $0b110$, meaning that the resulting path only works for the property *double-free* and the property *mem-leak*.

$$\begin{aligned}(p, \mathit{ret}\ p)^{0b111} \circ (a) \circ (u, \mathit{free}(u))^{0b110} \\ &= (p, \mathit{ret}\ p, a, u, \mathit{free}(u))^{0b111\&0b110} \\ &= (p, \mathit{ret}\ p, a, u, \mathit{free}(u))^{0b110}\end{aligned}$$

We observe that using value-flow paths as function summaries has a significant advantage for checking multiple properties. That is, since value flow is a common program relations, it can be reused across different properties. This is different from existing approaches that utilize state machine to model properties and generate state-specific function summaries [43, 30]. Since different properties usually have different states, compared to our value-flow-based function summaries, such state-specific function summaries have fewer opportunities to be reused across properties.

4.5 Implementation

In this section, we present the implementation details as well as the properties to check in our framework.

Table 4.3: Properties to check.

ID	Property Name	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 function arguments
19	alpha.unix.SimpleStream	Check for misuses of C stream APIs, e.g., an opened file is not closed
20	alpha.unix.Stream	Check stream handling functions, e.g., using a null file handle in fseek

4.5.1 Path-sensitivity and Parallelization

We have implemented our approach as a prototype tool called Catapult on top of Pinpoint [106]. Given the source code of a program, we first compile it to LLVM bitcode, on which our analysis is performed. To achieve path-sensitivity, we build a path-sensitive value-flow graph and compute path conditions following the method of Pinpoint. The path conditions in our analysis are first-order logic formulae over bit vectors. A program variable is modeled as a bit vector, of which the length is the bit width (e.g., 32) of the variable's type (e.g., int). The path conditions are solved by Z3 [31], a state-of-the-art SMT solver, to determine the path feasibility.

Our analysis is performed in a bottom-up manner, in which a function is always analyzed before its callers. After a function is analyzed, its function behavior is summarized as function summaries, which can be reused at different call sites. Thus, it is easy to run in parallel by analyzing functions without caller-callee relations independently [125]. Our special design for checking multiple properties together does not prevent the analysis from this parallelization strategy.

4.5.2 Properties to Check

Catapult currently supports twenty C/C++ properties, briefly introduced in Table 4.3, defined by Clang.[3] These properties include all Clang's default C/C++ value-flow properties. All other default C/C++ properties in Clang but not in Catapult are simple ones that do not require a path-sensitive analysis. For example, the property security.insecureAPI.bcopy requires Clang report a warning whenever a program statement calling the function *bcopy* is found.

4.5.3 Soundness

We implement Catapult in a soundy manner [76]. This means that the implementation soundly handles most language features and, meanwhile, includes some well-known unsound design decisions as previous works [125, 21, 8, 115, 106]. For example, in our implementation, virtual functions are resolved by classic class hierarchy analysis [32]. However, we do not handle C style function pointers, inline assembly, and library functions. We also follow the common practice to assume distinct function parameters do not alias with each other [75] and unroll each cycle twice on the call graph and the control flow graph. These unsound choices significantly improve the scalability but have limited negative impacts on the bug-finding capability.

4.6 Evaluation

To demonstrate the scalability of our approach, we compared the time and the memory cost of Catapult to three existing industrial-strength static analyzers. We also investigated the capability of finding real bugs in order to show that the increased scalability is not at the cost of sacrificing the bug-finding capability.

[3]More details of the properties can be found on `https://clang-analyzer.llvm.org/`.

Table 4.4: Subjects for evaluation.

ID	Program	Size (KLoC)	ID	Program	Size (KLoC)
1	mcf	2	13	shadowsocks	32
2	bzip2	3	14	webassembly	75
3	gzip	6	15	transmission	88
4	parser	8	16	redis	101
5	vpr	11	17	imagemagick	358
6	crafty	13	18	python	434
7	twolf	18	19	glusterfs	481
8	eon	22	20	icu	537
9	gap	36	21	openssl	791
10	vortex	49	22	mysql	2,030
11	perlbmk	73			
12	gcc	135	**Total**		5,303

We first compared Catapult to Pinpoint, the value-flow analyzer introduced in the previous chapter. Both techniques are demand-driven, compositional, and sparse with the precision of inter-procedural path-sensitivity. In addition, we also compared Catapult to two open-source bug finding tools, Clang and Infer. They were configured to use fifteen threads to take advantage of parallelization.

We also tried to compare to other static bug detection tools such as Saturn [125], Calysto [8], Semmle [7], Fortify, and Klocwork.[4] However, they are either unavailable or not runnable on the experimental environment we are able to set up. The open-source static analyzer, FindBugs,[5] is not included in our experiments because it only works for Java while we focus on the analysis of C/C++ programs. We do not compare to Tricoder [98], the static analysis platform from Google. This is because it uses Clang as the C/C++ analyzer, which is included in our experiments.

To avoid possible biases on the benchmark programs, we include the standard and widely-used benchmarks, SPEC CINT2000 (ID = $1 \sim 12$ in Table 4.4), in our evaluation. Meanwhile, in order to demonstrate the efficiency and effectiveness of Catapult on real-world projects, we also include ten industrial-sized open-source C/C++ projects (ID = $13 \sim 22$ in Table 4.4), of which the size ranges from a few thousand to two million lines of code.

All experiments were performed on a server with eighty "Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz" processors and 256GB of memory running Ubuntu-16.04.

[4]Klocwork Static Analyzer: `https://www.roguewave.com/products-services/klocwork`
[5]Findbugs Static Analyzer: `http://findbugs.sourceforge.net`

Table 4.5: Effectiveness (Catapult vs. Pinpoint).

Program	Catapult		Pinpoint	
	# Rep	# FP	# Rep	# FP
shadowsocks	9	0	9	0
webassembly	10	2	10	2
transmission	24	2	24	2
redis	39	5	39	5
imagemagick	26	8	-	-
python	48	7	48	7
glusterfs	59	22	59	22
icu	161	31	-	-
openssl	48	15	-	-
mysql	245	88	-	-
% FP	26.9%		20.1%	

Table 4.6: Effectiveness (Catapult vs. Clang, and Infer).

Program	Catapult		Clang (Z3)		Clang (Default)		Infer†	
	# Rep	# FP	# Rep	# FP	# Rep	# FP	# Rep	# FP
shadowsocks	8	2	24	22	25	23	15	13
webassembly	4	0	1	0	6	2	12	12
transmission	31	10	17	12	26	21	167*	82
redis	19	6	15	7	32	20	16	7
imagemagick	24	7	34	21	78	61	34	18
python	37	7	62	40	149*	77	82	63
glusterfs	28	5	0	0	268*	82	-	-
icu	55	11	94	67	206*	69	248*	71
openssl	39	19	44	26	44	26	211*	85
mysql	59	20	271*	59	1001*	79	258*	80
% FP	28.6%		64.9%		75.7%		78.6%	

* We inspected one hundred randomly-sampled bug reports.
† We fail to run the tool on glusterfs.

4.6.1 Comparing to Static Value-Flow Analyzer

We first compared Catapult to Pinpoint, the state-of-the-art value-flow analyzer. To quantify the effect of the graph traversal plan and the optimization plan separately, we also configured Catapult* to only contain the traversal plan.

In this experiment, we performed the whole program analysis by linking all compilation units of a project into a single file for the static analyzers to perform the cross-file analysis. Before the analysis, both Pinpoint and Catapult need to build the value-flow graph as the program intermediate representation. Since Catapult is built on top of Pinpoint, the pre-processing time and the size of value-flow graph are the same for both tools, which are almost linear to the size of a program [106]. Typically,

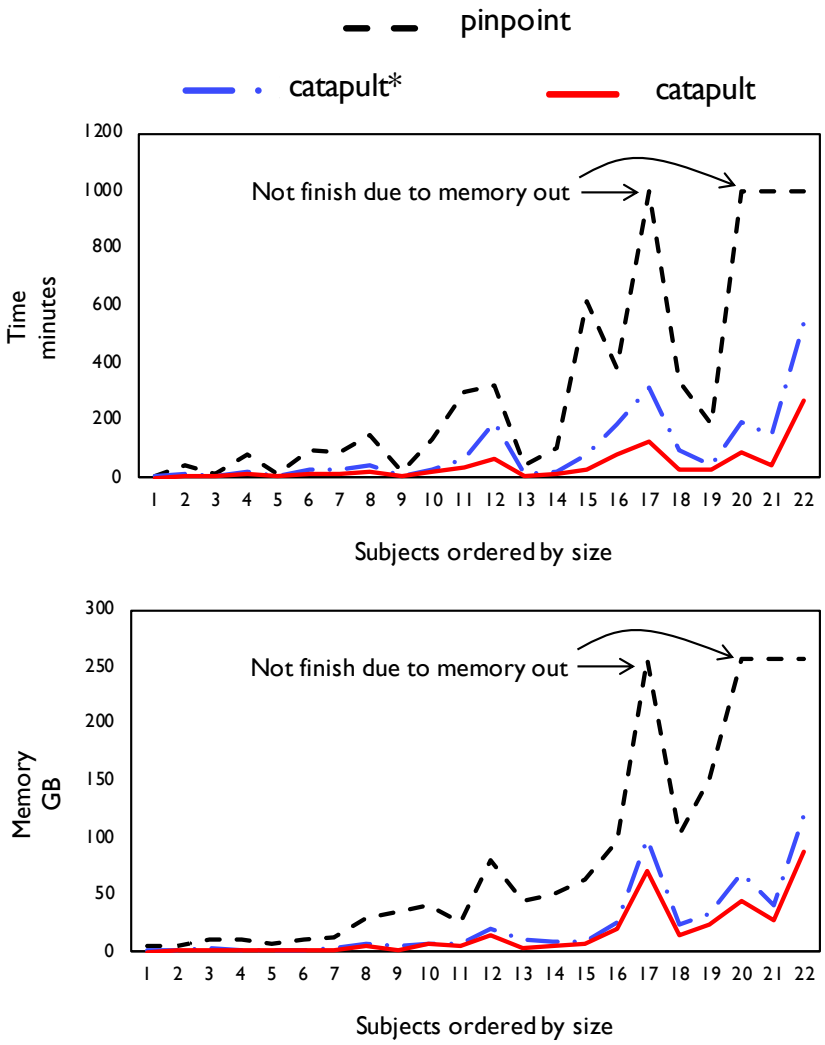

Figure 4.6: Comparing the time and the memory cost to Pinpoint.

for MySQL, a program with about two million lines of code, it takes twenty minutes to build a value-flow graph with seventy million nodes and ninety million edges.

(1) Efficiency. The time and memory cost of checking each benchmark program is shown in Figure 4.6. Owing to the inter-property-awareness, Catapult is about 8× faster than Pinpoint and takes only 1/7 of the memory on average. Typically, Catapult can finish checking MySQL in 5 hours, which is aligned with the industrial requirement of finishing an analysis in 5 to 10 hours [13, 79].

When the optimization plan is disabled, Catapult* is about 3.5× faster than Pinpoint and takes 1/5 of the memory on average. Compared to the result of Catapult, it implies that the graph traversal plan and the optimization plan contribute to 40% and 60% of the time cost reduction, respectively. Meanwhile, they contribute to 70% and 30% of the memory cost reduction, respectively. As a summary, the two plans

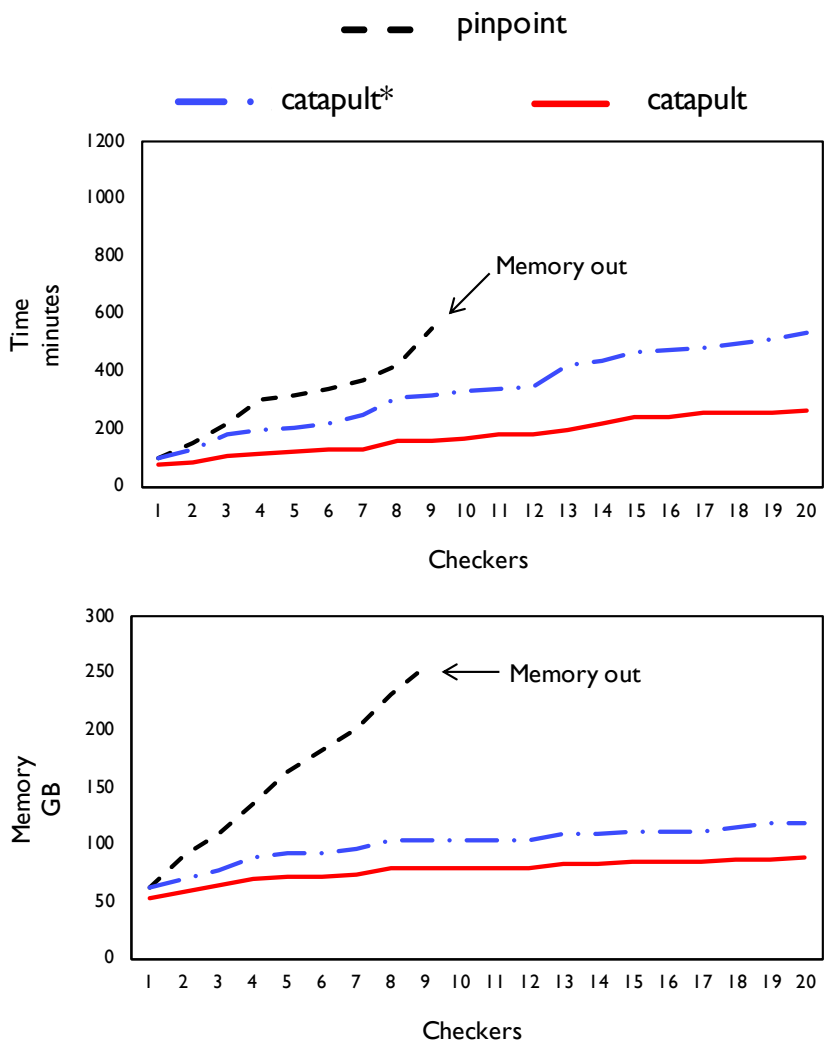

Figure 4.7: The growth curves of the time and the memory overhead when comparing to Pinpoint.

contribute similar to the time cost reduction, and the graph traversal plan is more important for the memory cost reduction because it allows us to avoid duplicate data storage by sharing analysis results across different properties.

Using the largest subject, MySQL, as an example, Figure 4.7 illustrates the growth curves of both the time and the memory overhead when the properties in Table 4.3 are added into the core engine one by one. Figure 4.7 shows that, in terms of both time and memory overhead, Catapult grows much slower than Pinpoint and, thus, scales up quite gracefully.

It is noteworthy that, except for the feature of inter-property-awareness, Catapult follows the same method of Pinpoint to build value-flow graph and perform path-sensitive analysis. Thus, they have the similar performance to check a single property.

Catapult performs better than Pinpoint only when multiple properties are checked together.

(2) Effectiveness. Since both Catapult and Pinpoint check programs with the precision of inter-procedural path-sensitivity, as shown in Table 4.5, they produce a similar number of bug reports (# Rep) and false positives (# FP) for all the real-world programs except for the programs that Pinpoint fails to analyze due to the out-of-memory exception.

4.6.2 Comparing to Other Static Analyzers

To better understand the performance of Catapult in comparison to other types of property-unaware static analyzers, we also ran Catapult against two prominent and mature static analyzers, Clang (based on symbolic execution) and Infer (based on abductive inference). Note that Infer does not classify the properties to check as Table 4.3 but targets at a similar range of properties, such as null dereference, memory leak, and others.

In our experiment, Clang was run with two different configurations: one is its default configuration where a fast but imprecise range-based solver is employed to solve path conditions, and the other uses Z3 [31], a full-featured SMT solver, to solve path conditions. To ease the explanation, we denote Clang in the two configurations as Clang (Default) and Clang (Z3), respectively. Since Clang separately analyzes each source file and Infer only has limited capability of detecting cross-file bugs, for a fair comparison, all tools in the experiments were configured to check source files separately, and the time limit for analyzing each file is set to 60 minutes. Since a single source file is usually small, we did not encounter memory issues in the experiment but missed a lot of cross-file bugs as discussed later. Also, since we build value-flow graphs separately for each file and do not need to track cross-file value flows, the time cost of building value-flow graphs is almost negligible. Typically, for MySQL, it takes about five minutes to build value-flow graphs for all of its source code. This time cost is included in the results discussed below.

Note that we did not change other default configurations of Clang and Infer. This is because the default configuration is usually the best in practice. Modifying their default configuration may introduce more biases.

(1) Efficiency (Catapult vs. Clang (Z3)). When both Catapult and Clang employ Z3 to solve path conditions, they have similar precision (i.e., full path-

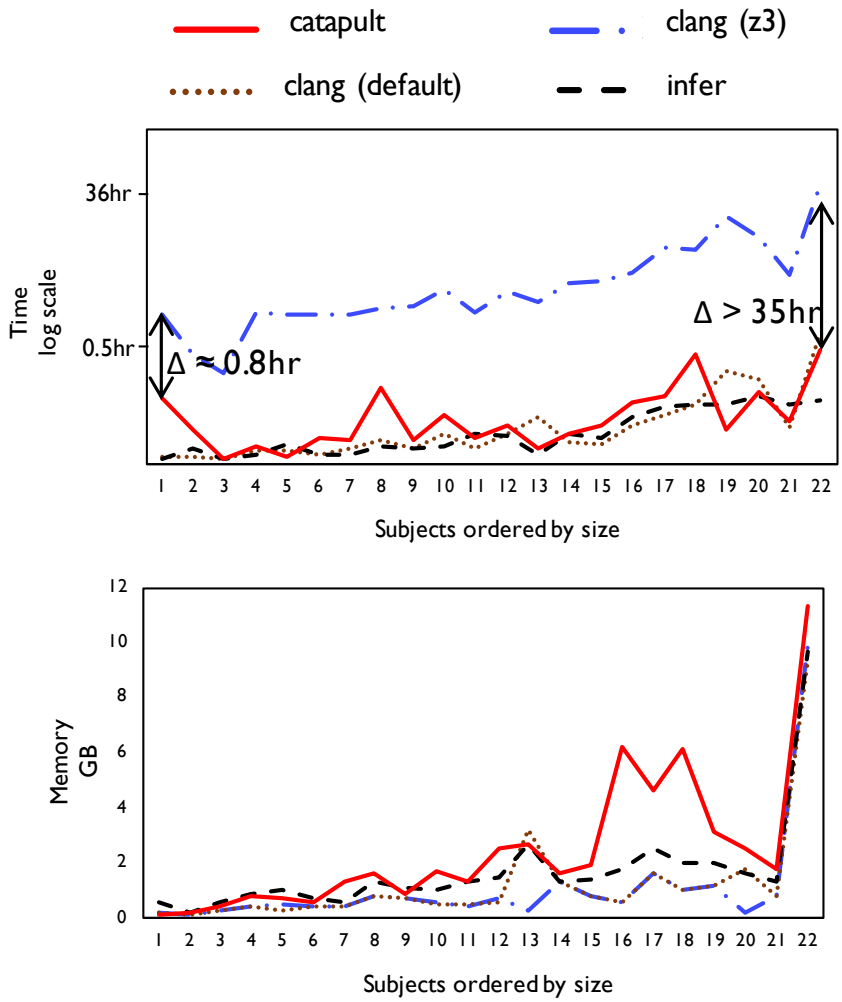

Figure 4.8: Comparing the time and the memory cost to Clang and Infer.

sensitivity) in theory. However, as illustrated in Figure 4.8, Catapult is much faster than Clang and consumes a similar amount of memory for all of the subjects. For example, for MySQL, it takes about 36 hours for Clang to finish the analysis while Catapult takes only half an hour, consuming a similar amount of memory. On average, Catapult is 68× faster than Clang at the cost of only 2× more memory space. Both analyses can finish in 12GB of memory, available in common personal computers.

(2) Efficiency (Catapult vs. Clang (Default) and Infer). As illustrated in Figure 4.8, compared to both Infer and the default version of Clang, Catapult consumes a similar, sometimes a little higher, amount of time and memory. For instance, for MySQL, the largest subject program, all three tools finish the analysis in 40 minutes and consume about 10GB of memory. With similar efficiency, Catapult, as a fully path-sensitive analysis, is much more precise than the other two. The lower precision of Clang and Infer leads to many false positives as discussed below.

(3) Effectiveness. In addition to the efficiency, we also investigate the bug-finding capability of the tools. Table 4.6 presents the results. Since we only perform file-level analysis in this experiment, the bugs reported by Catapult is much fewer than those in Table 4.5. Because of the prohibitive cost of manually inspecting all of the bug reports, we randomly sampled a hundred reports for the projects that have more than one hundred reports. Our observation shows that, on average, the false positive rate of Catapult is much lower than both Clang and Infer. In terms of recall, Catapult reports more true positives, which cover all those reported by Clang and Infer. Clang and Infer miss many bugs due to the trade-offs they make in exchange for efficiency. For example, Clang often stops its analysis on a path after it finds the first bug.

Together with the results on efficiency, we can conclude that Catapult is much more scalable than Clang and Infer because they have similar time and memory overhead but Catapult is much more precise and able to detect more bugs.

4.6.3 Detected Real Bugs

We note that the real-world software used in our evaluation is frequently scanned by commercial tools such as Coverity SAVE and, thus, is expected to have very high quality. Nevertheless, due to the high efficiency, precision, and recall, Catapult still can detect many deeply-hidden software bugs that existing static analyzers, such as Pinpoint, Clang, and Infer, cannot detect.

At the time of writing, thirty-nine previously-unknown bugs have been confirmed and fixed by the software developers, including seventeen null pointer dereferences, ten use-after-free or double-free bugs, eleven resource leaks, and one stack-address-escape bug. Four of them even have been assigned CVE IDs due to their significant security impact.

As an example, Figure 4.9 presents a null-deference bug detected by Catapult in ImageMagick, which is a software suite for processing images. This bug is of high complexity, as it occurs in a function of more than 1,000 lines of code and the control flow involved in the bug spans across 56 functions over 9 files.

Since both Clang and Infer make many unsound trade-offs to achieve scalability, neither of them detects this bug. Pinpoint also cannot detect the bug because it is not memory-efficient and has to give up its analysis after the memory is exhausted.

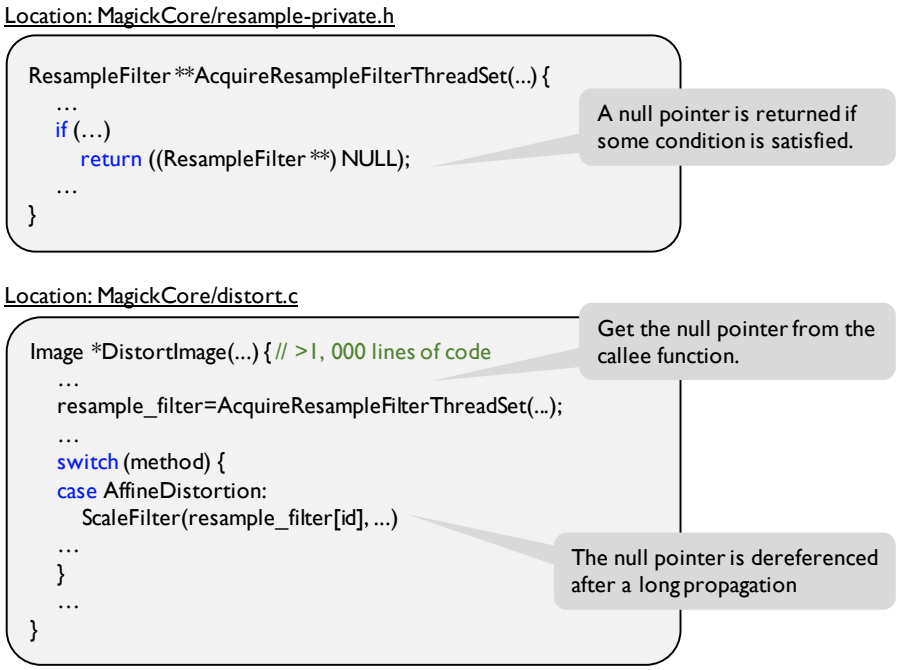

Figure 4.9: A null-dereference bug in ImageMagick.

4.7 Conclusion

We have presented Catapult, a scalable approach to checking multiple value-flow properties together. The critical factor that makes our technique fast is to exploit the mutual synergy among the properties to check. Since the number of program properties to check is quickly increasing nowadays, we believe that it will be an important research direction to study how to scale up static program analysis for simultaneously checking multiple properties.

Chapter 5

Scaling up Sparse Value-Flow Analysis via Parallelization

5.1 Introduction

Bottom-up analyses work by processing the call graph of a program upwards from the leaves – before analyzing a function, all its callee functions are analyzed and summarized as function summaries [125, 8, 106, 18, 124, 20, 37, 22, 38]. These analyses have two key strengths: the function summaries they compute are highly reusable and they are easy to parallelize because the analyses of functions are decoupled. While almost all existing bottom-up analyses, including our approach, take advantage of such function-level parallelization, there is little progress in improving its parallelism. As reported by recent studies, it still needs to take a few hours, even tens of hours, to precisely analyze large-scale software. For example, it takes 6 to 11 hours for Saturn [125] and Calysto [8] to analyze programs of 685KLoC [8].

5.1.1 The Limit of Parallelization

With regard to the performance issues, McPeak et al. [79] pointed out that the parallelism often drops off at runtime and, thus, the CPU resources are usually not well utilized. Specifically, this is because the parallelism is significantly limited by the *calling dependence* – functions with caller-callee relations have to be analyzed sequentially because the analysis of a caller function depends on the analysis results, i.e., function summaries, of its callee functions. To illustrate this phenomenon, let us consider the call graph in Figure 5.1 where the function f calls the functions, g and h. In a conventional bottom-up analysis, only functions without caller-callee relations,

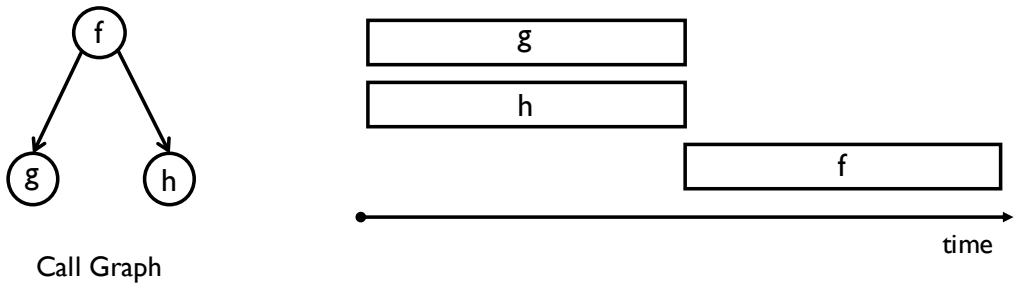

Figure 5.1: Conventional parallel design of bottom-up program analysis. Each rectangle represents the analysis task for a function.

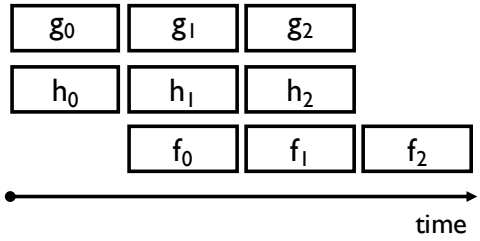

Figure 5.2: The analysis task of each function is partitioned into multiple sub-tasks. All sub-tasks are pipelined.

e.g., the function g and the function h, can be analyzed in parallel. The analysis of the function f cannot start until the analyses of the functions, g and h, complete. Otherwise, when analyzing a call site of the function, g or h, in the function f, we may miss some effects of the callees due to the incomplete analysis.[1]

5.1.2 Breaking the Limit of Parallelization

In this chapter, we present Cheetah, a framework of bottom-up data flow analysis that breaks the limits of function boundaries, so that functions having calling dependence can be analyzed in parallel. As a result, we can achieve much higher parallelism than the conventional parallel design of bottom-up analysis. Our key insight is that many analysis tasks of a caller function only depend on partial analysis results of its callee functions. Thus, the analysis of the caller function can start before the analyses of its callee functions complete. Therefore, our basic idea is to partition the analysis task of a function into multiple sub-tasks, so that we can pipeline the sub-tasks to generate function summaries. The key to the partition

[1]This is different from a top-down method that can let the analysis of the function f run first but stop to wait for the analysis results of the function g when analyzing a call statement calling the function g.

is a soundness criterion, which requires a sub-task only depends on the summaries produced by the sub-tasks finished in the callees. Violating this criterion will cause the analysis to neglect certain function effects and make the analysis unsound.

To illustrate, assume that the analysis task of each function in Figure 5.1, e.g., the function f, is partitioned into three sub-tasks, f_0, f_1, and f_2, each of which generates one kind of function summaries. These sub-tasks satisfy the constraints that the sub-task f_i only depends on the function summaries produced by the sub-task g_j and the sub-task h_j ($j \leq i$). As a result, these sub-tasks can be pipelined as illustrated in Figure 5.2, where the analysis of the function f starts immediately after the sub-tasks g_0 and h_0 finish. Clearly, the parallelism in Figure 5.2 is much higher than that in Figure 5.1, providing a significant speedup over the conventional parallel design of bottom-up analysis.

In this work, we formalize our idea under the IFDS/IDE framework for a wide range of data flow problems known as the inter-procedural finite distributive subset or inter-procedural distributive environment problems [93, 100]. In both problems, the data flow functions are required to be distributive over the merge operator. Although this is a limitation in some cases, the IFDS/IDE framework has been widely used for many practical problems such as secure information flow [6, 88, 52], typestate [46, 84], alias sets [85], specification inference [107], and shape analysis [94, 128]. Given any of those IFDS/IDE problems, conventional solutions compute function summaries either in a bottom-up fashion (e.g., [130, 97]) or in a top-down manner (e.g., [93, 100]), depending on their specific design goals. In this work, we focus on the bottom-up solutions and aim to improve their performance via the pipeline parallelization strategy. We also discuss how to apply the pipeline parallelization strategy to sparse value-flow analysis.

We implemented Cheetah to path-sensitively check null dereferences and taint issues in C/C++ programs. Our evaluation of Cheetah is based on standard benchmark programs and many large-scale software systems, which demonstrates that the calling dependence significantly limits the parallelism of bottom-up data flow analysis. By relaxing this dependence, our pipeline strategy achieves 2×-3× speedup over the conventional parallel design of bottom-up analysis. Such speedup is significant enough to make many overly lengthy analyses useful in practice. In summary, the main contributions of this chapter include the following:

- We propose the design of pipelineable function summaries, which enables the pipeline parallelization strategy for bottom-up data flow analysis.

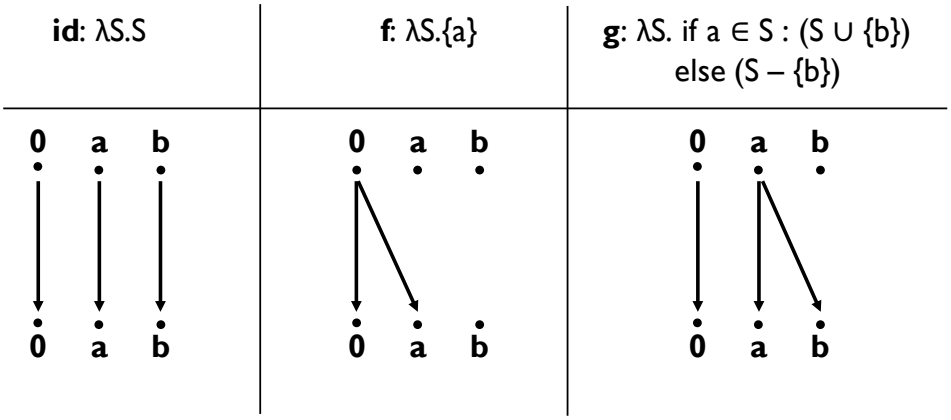

Figure 5.3: Data flow functions and their representation in the exploded super-graph.

- We formally prove the correctness of our approach and apply it to a null analysis and a taint analysis to show its generalizability.
- We conduct a systematic evaluation to demonstrate that we can achieve much higher parallelism and, thus, run faster than the state of the arts.

5.2 Overview

In this section, we introduce the background of the IFDS/IDE framework and provide an example to illustrate how we improve the parallelism of a bottom-up analysis by partitioning the analysis of a function.

5.2.1 The IFDS/IDE Framework

The IFDS/IDE framework aims to solve a wide range of data flow problems known as Inter-procedural Finite Distributive Subset or Inter-procedural Distributive Environment problems [93, 100]. Its basic idea is to transform a data flow problem to a graph reachability problem on the *exploded super-graph*, which is built based on the inter-procedural control flow graph of a program.

(1) The IFDS Framework. In the IFDS framework, every vertex $(s_i, \mathbf{d})$ in the exploded super-graph stands for a statically decidable data flow fact, or simply, fact, $\mathbf{d}$ at a program point s_i. Every edge models the data flow functions between data flow facts. In the chapter, to ease the explanation, we use s_i to denote the program point at Line i in the code. For example, in an analysis to check null dereference,

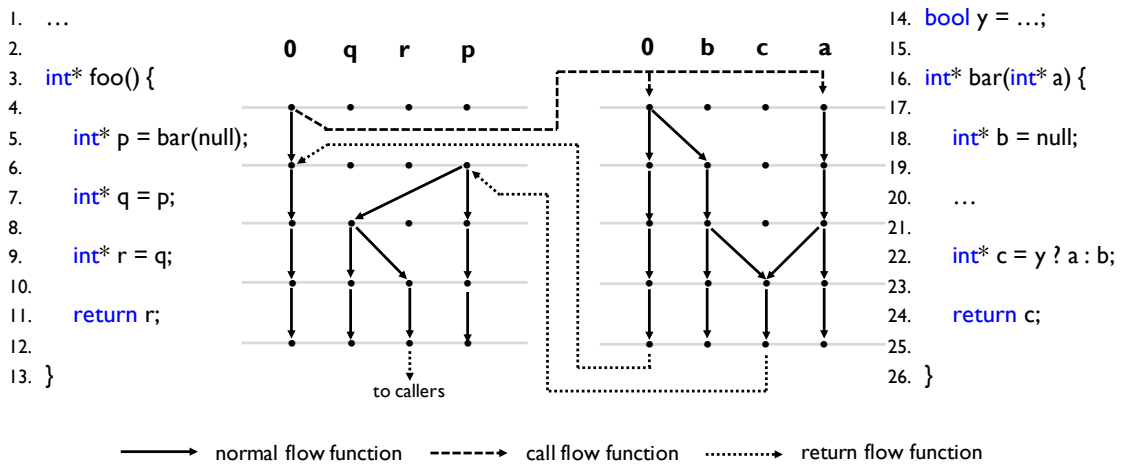

Figure 5.4: An example of the exploded super-graph for a null-dereference analysis.

the vertex $(s_i, \mathbf{d})$ could denote that the variable d is a null pointer at Line i. As for the edges or data flow functions, Figure 5.3 illustrates three examples that show how the commonly-used data flow functions are represented as edges in the exploded super-graph. The vertices at the top are the data flow facts before a program point and the vertices at the bottom represent the facts after the program point.

The first data flow function **id** is the identity function which maps each data flow fact before a program point to itself. It indicates that the statement at the program point has no impacts on the data flow analysis.

The special vertex for the fact **0** is associated with every program point in the program. It denotes a tautology, a data flow fact that always holds. An edge from the fact **0** to a non-**0** fact indicates that the non-**0** fact is freshly created. For example, in the second function in Figure 5.3, the fact **a** is created, which is represented by an edge from the fact **0** to the fact **a**. At the same time, since **a** is the only fact after the data flow function, there is no edge connecting the fact **b** before and after the program point.

The third data flow function is a typical function that models the assignment b=a. In the exploded super-graph, the variable a has the same value as before. Thus, there is an edge from the data flow fact **a** to itself. The variable b gets the value from the variable a, which is modeled by the edge from the fact **a** to the fact **b**.

It is noteworthy that the data flow facts are not limited to simple values like the local variables in the examples. For example, in alias analysis, the facts can be

sets of access paths [116]. In typestate analysis, the facts can be the combination of different typestates [84].

Figure 5.4 illustrates the exploded super-graph for a data flow analysis that tracks the propagation of null pointers. Since Line 18 assigns a null pointer to the variable b, we have the edge from the vertex $(s_{17}, \mathbf{0})$ to the vertex $(s_{19}, \mathbf{b})$, meaning that we have the data flow fact $b = \text{null}$ at Line 19. Since Line 19 does not change the value of the variable a, we have the edge from the vertex $(s_{17}, \mathbf{a})$ to the vertex $(s_{19}, \mathbf{a})$, which means the data flow fact about the variable a does not change.

Assuming that s_{main} is the program entry point, the IFDS framework aims to find paths, or determine the reachability relations, between the vertex $(s_{\text{main}}, \mathbf{0})$ and the vertices of interests. Each of such paths represents that some data flow fact holds at a program point. For instance, the path from the vertex $(s_4, \mathbf{0})$ to the vertex $(s_{12}, \mathbf{r})$ in Figure 5.4 implies that the fact $r = \text{null}$ holds at Line 12.

The IFDS method is efficient because it computes function summaries only once for each function. Each summary is a path on the exploded super-graph connecting a pair of vertices at the entry and the exit of a function. The path from the vertex $(s_{17}, \mathbf{a})$ to the vertex $(s_{25}, \mathbf{c})$ in Figure 5.4 is such a summary of the function *bar*. When analyzing the callers of the function *bar*, e.g., the function *foo*, we can directly jump from the vertex $(s_4, \mathbf{0})$ to the vertex $(s_6, \mathbf{p})$ using the summary without analyzing the function *bar* again.

(2) The IDE Framework. The IDE framework is a generalization of the IFDS framework [100]. Similar to the IFDS framework, it also works as a graph traversal on an exploded super-graph. There are three major differences. First, each vertex on the exploded super-graph is no longer associated with a simple data flow fact $\mathbf{d}$, but an environment mapping a fact $\mathbf{d}$ to a value $\mathbf{v}$ from a separate value domain, denoted as $\{\mathbf{d} \mapsto \mathbf{v}\}$. Second, due to the first difference, the data flow functions, i.e., the edges on the exploded super-graph, transform an environment $\{\mathbf{d} \mapsto \mathbf{v}\}$ to the other $\{\mathbf{d}' \mapsto \mathbf{v}'\}$. The third important difference is that each edge on the exploded super-graph is labeled with an environment transform function, which makes IDE no longer only a simple graph reachability problem. Instead, it has to find the paths between two vertices of interests and, meanwhile, compose the environment transform functions labeled on the edges along the paths. These differences widen the class of problems that can be expressed in the IFDS framework.

In this work, for simplicity, we describe our work under the IFDS framework. This does not lose the generality for the IDE problems because, intuitively, both problems are solved by a graph traversal on the exploded super-graph.

5.2.2 An Example

Let us briefly explain our approach using the example in Figure 5.4, where the analysis aims to track the propagation of null pointers.

(1) Bottom-up Analysis. For the example in Figure 5.4, a conventional bottom-up analysis firstly analyzes the function *bar* and produces function summaries to summarize its behavior. With the function summaries in hand, the function *foo* then is analyzed.

Using the symbol $\rightsquigarrow$ to denote a path between two vertices, a common IFDS/IDE solution will generate the following two intra-procedural paths as the summaries of the function *bar*:

- The path $(s_{17}, \mathbf{a}) \rightsquigarrow (s_{25}, \mathbf{c})$ summarizes the function behavior that a null pointer created in a caller of the function *bar*, i.e., $a = \text{null}$, may be returned back to the caller.
- The path $(s_{17}, \mathbf{0}) \rightsquigarrow (s_{25}, \mathbf{c})$ summarizes the function behavior that a null pointer created in the function *bar* may be returned to the caller functions.

Note that we do not need to summarize the path $(s_{17}, \mathbf{0}) \rightsquigarrow (s_{25}, \mathbf{0})$ for the function *bar*, because the fact $\mathbf{0}$ is a tautology and always holds.

Next, we analyze the function *foo* by a graph traversal from the vertex $(s_4, \mathbf{0})$, which aims to track the propagation of null pointers and produce function summaries of the function *foo*. During the graph traversal, when the call flow functions (i.e., the dashed edges) are visited, we apply the summaries of the function *bar* and produce two summaries of the function *foo* as following ($[\![\cdot]\!]_{bar}$ are the summaries of the function *bar*):

- The path $(s_4, \mathbf{0}) \rightsquigarrow [\![(s_{17}, \mathbf{a}) \rightsquigarrow (s_{25}, \mathbf{c})]\!]_{bar} \rightsquigarrow (s_6, \mathbf{p}) \rightsquigarrow (s_{12}, \mathbf{r})$ summarizes the behavior that a null pointer in the function *foo* will be returned to its callers.
- The path $(s_4, \mathbf{0}) \rightsquigarrow [\![(s_{17}, \mathbf{0}) \rightsquigarrow (s_{25}, \mathbf{c})]\!]_{bar} \rightsquigarrow (s_6, \mathbf{p}) \rightsquigarrow (s_{12}, \mathbf{r})$ summarizes the function behavior that a null pointer in the callees of the function *foo* will be returned to the callers of the function *foo*.

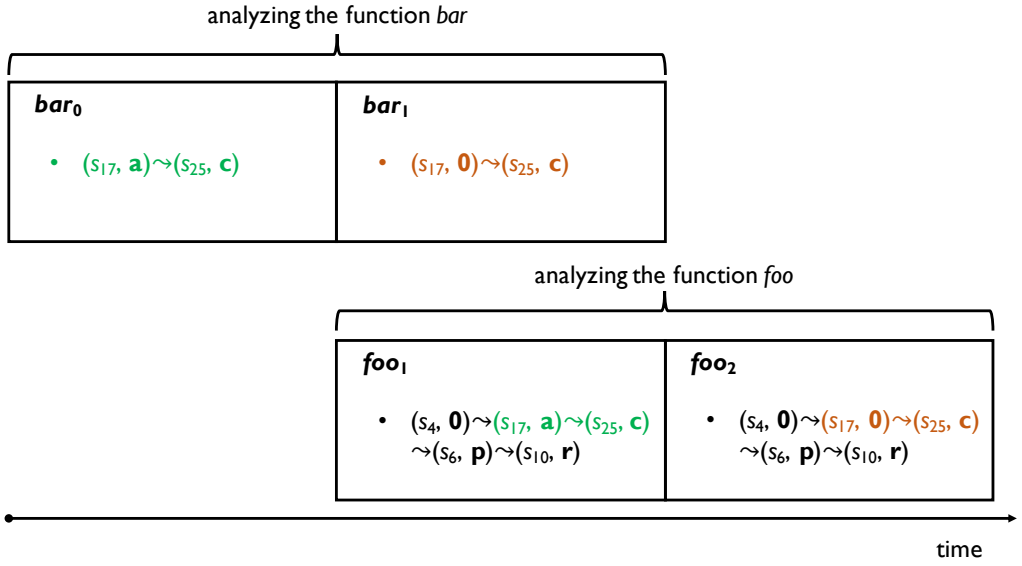

Figure 5.5: The pipeline parallelization strategy.

(2) Our Approach. As discussed before, in a conventional bottom-up analysis, the analysis of a caller function needs to wait for the analysis of its callees to complete. Differently, Cheetah aims to improve the parallelism by starting the analysis of the function *foo* before completing the analysis of the function *bar*. To this end, Cheetah partitions the analysis of each function f into three parts based on where a data flow fact is created. Such a partition categorizes the function summaries into three groups, f_0, f_1, and f_2, which we refer to as the pipelineable summaries:

- f_0 summarizes the behavior that some data flow facts created in the caller functions will be propagated back to the callers through the current function. The first summary of the function *bar* is an example.
- f_1 summarizes the behavior that some data flow facts created in the current function will be propagated back to the caller functions. The second summary of the function *bar* and the first summary of the function *foo* are two examples.
- f_2 summarizes the behavior that some data flow facts created in the callees are propagated to the current function and will continue to be propagated to the caller functions. The second summary of the function *foo* is an example.

According to the partition method, the summaries of the function *foo* is partitioned into two sets, foo_1 and foo_2, just as illustrated in Figure 5.5. Since the function *foo* does not have any function parameters, the set foo_0 is empty and, thus, omitted. Similarly, the summaries of the function *bar* is partitioned into two sets, bar_0 and

bar_1. Since the function *bar* does not have any callees, the set bar_2 is empty and, thus, omitted. As detailed later, the above partition is sound because it satisfies the constraint that summaries in the set foo_i only depends on the summaries in the set $bar_j (j \leq i)$. Thus, we can safely pipeline the analyses of the function *foo* and the function *bar* - we can start analyzing the function *foo* immediately after summaries in the set bar_0 are generated.

In the remainder of this chapter, under the IFDS framework, we formally present how to partition the analysis of a function to generate pipelineable function summaries, so that the parallelism of bottom-up analysis can be improved in a sound manner.

5.3 Pipelined Bottom-up Analysis

To explain our method in detail, we first define the basic notations and terminologies in Section 5.3.1 and then explain the criteria that guide our partition method in Section 5.3.2. Based on the criteria, we present the technical details of our pipeline parallelization strategy from Section 5.3.3 to Section 5.3.5. Finally, we discuss the application of the pipeline parallelization strategy in Section 5.3.6

5.3.1 Preliminaries

(1) Program Model. Given an IFDS problem, a program is modeled as an exploded super graph G that consists of a set of intra-procedural graphs $\{G_f, G_g, G_h \dots\}$ of the functions $\{f, g, h, \dots\}$. Given a function f, its local graph G_f is a tuple $(L_f, e_f, x_f, D_f, E_f)$:

- L_f is the set of program locations in the function.
- $e_f, x_f \in L_f$ are the entry and exit points of the function.
- D_f is the set of data flow facts in the function.
- $L_f \times D_f$ is the set of vertices of the graph.
- $E_f \subseteq (L_f \times D_f) \times (L_f \times D_f)$ is the edge set (see Figure 5.3).

As illustrated in Figure 5.4, the local graphs of different functions are connected by call and return flow functions, respectively.

(2) Function Summaries. For any function f, its function summaries are a set of paths between data flow facts at the entry point and data flow facts at its exit point [93], denoted as $S_f = \{(e_f, \mathbf{a}) \rightsquigarrow (x_f, \mathbf{b}) : \mathbf{a}, \mathbf{b} \in D_f\}$. Apparently, we can generate these summaries by traversing the graph G_f from every vertex at the function entry.

Owing to function calls in a program, the summaries of a function often depend on the summaries of its callees. We say a summary set S depends on the other summary set S' if and only if there exists a path in the set S that subsumes a path in the set S'. As illustrated in Section 5.2.2, the summaries of the function *foo* depend on the summaries of the function *bar*.

(3) Summary Dependence Graph. To describe the dependence between summary sets, we define the summary dependence graph, where a vertex is a set of function summaries and a directed edge indicates the source summary set depends on the destination summary set.

The summary dependence graph is built based on the call graph. Conventionally, vertices of the summary dependence graph are the summary sets $\{S_f, S_g, S_h \dots\}$, and an edge from the summary set S_f to the summary set S_g exists if and only if the function f calls the function g. A bottom-up analysis works by processing the summary dependence graph upwards from the leaves. It starts generating summaries in a summary set if it does not depend on other summary sets or the summary sets it depends on have been generated. Summary sets that do not have dependence relations can be generated in parallel.

(4) Problem Definition. In this work, we aim to find a partition for the summary set of each function, say $\Pi(S_f) = \{S_f^0, S_f^1, S_f^2, \dots\}$,[2] such that a vertex of the summary dependence graph is no longer a complete summary set S_f but a subset S_f^i $(i \geq 0)$. Meanwhile, to improve the parallelism, the bottom-up analysis based on the dependence graph should be able to generate summaries for a pair of caller and callee functions at the same time. In detail, the partition needs to satisfy the criteria discussed in the next subsection.

5.3.2 Partition Criteria

Given a pair of functions where the function f calls the function g, we use the set $\Omega(S_f, S_g) \subseteq \Pi(S_f) \times \Pi(S_g)$ to denote the dependence relations between summary

[2]A set partition needs to satisfy $\cup_{i \geq 0} S_f^i = S_f$ and $\forall i, j \geq 0 : S_f^i \cap S_f^j = \emptyset$.

sets. Generally, an effective partition method must meet the following criteria to improve the parallelism of a bottom-up analysis.

(1) The Effectiveness Criterion. This criterion concerns whether the dependence between summary sets in the conventional bottom-up analysis is actually relaxed, so that the parallelism can be improved. We say the partition is effective if and only if $|\Omega(S_f, S_g)| < |\Pi(S_f) \times \Pi(S_g)|$. Intuitively, this means that some summaries in the caller function do not depend on all summaries in callee functions. Thus, the dependence relation in the conventional bottom-up analysis is relaxed.

(2) The Soundness Criterion. This criterion concerns the correctness after the dependence between summary sets is relaxed. We say the partition is sound if and only if the following condition is satisfied: if the set S_f^i depends on the set S_g^j, then $(S_f^i, S_g^j) \in \Omega(S_f, S_g)$. Violating this criterion will cause the analysis to neglect certain function summaries and make the analysis unsound.

(3) The Efficiency Criterion. This criterion concerns how many computational resources we need to consume in order to determine how to partition a summary set. Since summaries in the summary sets, S_f and S_g, are unknown before an analysis completes, the exact dependence relations between summaries in the two sets are also undiscovered. This fact makes it difficult to perform a fine-grained partition, unless the analysis has been completed and we have known what summaries are generated for each function.

As a trade-off, conventional bottom-up analysis does not partition the summary sets (or equivalently, $\Pi(S_f) = \{S_f\}$ and $\Pi(S_g) = \{S_g\}$). It conservatively utilizes the observation that all summaries in the set S_f may depend on certain summaries in the set S_g, i.e., $\Omega(S_f, S_g) = \{(S_f, S_g)\}$. Such a conservative method satisfies the soundness criterion and does not partition the summary sets. However, apparently, it does not meet the effectiveness criterion because $|\Omega(S_f, S_g)| = |\Pi(S_f) \times \Pi(S_g)| = 1$.

5.3.3 Pipelineable Summary-Set Partition

Generally, it is challenging to partition a summary set satisfying the above criteria because the exact dependence between summaries are unknown before the summaries are generated. We now present a coarse-grained partition method that requires few pre-computations, and thus, meets the efficiency criterion. Meanwhile, it also meets the effectiveness and soundness criteria and, thus, can soundly improve the

parallelism of a bottom-up analysis. We also establish a few lemmas to prove the correctness of our approach.

Intuitively, given a summary set S_f, we partition it according to where a data flow fact is created: in a caller of the function *f*, in the current function *f*, and in a callee of the function *f*. Formally, $\Pi(S_f) = \{S_f^0, S_f^1, S_f^2\}$, where

$$S_f^0 = \{(e_f, \mathbf{a}) \rightsquigarrow (x_f, \mathbf{b}) : \mathbf{a} \neq \mathbf{0}\}$$

$$S_f^1 = \{(e_f, \mathbf{0}) \rightsquigarrow (e_g, \mathbf{a}) \rightsquigarrow (x_f, \mathbf{b}) : f = g \vee \mathbf{a} \neq \mathbf{0}\}$$

$$S_f^2 = \{(e_f, \mathbf{0}) \rightsquigarrow (e_g, \mathbf{0}) \rightsquigarrow (x_f, \mathbf{b}) : f \neq g\}$$

By definition, there is no edge from a non-**0** data flow fact to the fact **0** on the exploded super-graph. An edge from the fact **0** to a non-**0** fact means that the non-**0** fact is freshly created [93]. Thus, any summary path in the set S_f^0 does not go through the fact **0**, meaning that the data flow fact is created in a caller of the function *f*. On the other hand, since a summary path in the set S_f^1 or the set S_f^2 starts with the fact **0**, it means that the non-**0** data flow fact on the summary path must be created in the function *f* or a callee of the function *f*. Specifically, since a summary path in the set S_f^1 does not go through the fact **0** in callee functions, the non-**0** data flow fact on the summary path is created in the function *f*. Similarly, the non-**0** data flow fact on a path from the set S_f^2 must be created in a callee of the function *f*.

The following lemma states that generating summaries in the sets, S_f^0, S_f^1, and S_f^2, does not miss any summary in the set S_f and, meanwhile, does not repetitively generate a summary in the set S_f.

Lemma 5.1. $\bigcup_{i \geq 0} S_f^i = S_f$ and $\forall i, j \geq 0 : S_f^i \cap S_f^j = \emptyset$.

Proof. This follows the definitions of the sets S_f^0, S_f^1, and S_f^2. □

Next, we study whether such a partition method follows the effectiveness and soundness criteria. The key to the problem is to compute the set $\Omega(S_f, S_g)$ of dependence relations between a pair of summary sets, S_f^i and S_g^j, given any pair of caller-callee functions, *f* and *g*.

Lemma 5.2. The sets S_f^0, S_f^1, and S_f^2 depend on the set S_g^0.

Proof. This follows the fact that any summary path in a caller function may go through a callee's summary path and the set S_g^0 is a part of the callee's summaries. □

Lemma 5.3. The set S_f^2 depends on the sets S_g^1 and S_g^2.

Proof. By definition, a summary path in the set S_f^2 needs to go through the vertex $(e_g, \mathbf{0})$. Given the function g, summary paths in both the set S_g^1 and the set S_g^2 start with the vertex $(e_g, \mathbf{0})$. Thus, the set S_f^2 depends on the sets S_g^1 and S_g^2. □

To demonstrate that the above lemmas do not miss any dependence relations, we establish the following two lemmas.

Lemma 5.4. The set S_f^0 does not depend on the sets S_g^1 and S_g^2.

Proof. This follows the fact that a non-**0** data flow fact cannot be connected back to the fact **0** [93], but a summary path in the sets S_g^1 and S_g^2 must start with the fact **0**. □

Lemma 5.5. The set S_f^1 does not depend on the sets S_g^1 and S_g^2.

Proof. By definition, a summary path in the set S_f^1 does not go through the fact **0** in a callee function. However, a summary path in the sets S_g^1 and S_g^2 must start with the fact **0**. Thus, the set S_f^1 does not depend on the sets S_g^1 and S_g^2. □

Putting Lemma 5.2 to Lemma 5.5 together, we have the dependence set $\Omega(S_f, S_g) = \{(S_f^0, S_g^0), (S_f^1, S_g^0), (S_f^2, S_g^0), (S_f^2, S_g^1), (S_f^2, S_g^2)\}$, which does not miss any dependence relation between the set S_f^i and the set S_g^j. Thus, the partition method satisfies the soundness criterion. Meanwhile, $|\Omega(S_f, S_g)| = 5 < |\Pi(S_f) \times \Pi(S_g)| = 9$. Thus, the effectiveness criterion is satisfied, meaning that the dependence between the summary sets is relaxed and, based on the partition, the parallelism of a bottom-up analysis can be improved.

Figure 5.6 illustrates the summary dependence graph for a pair of caller-callee functions, f and g. Based on the graph, when the summaries in the set S_g^0 are generated, a bottom-up analysis does not need to wait for summaries in the sets S_g^1 and S_g^2, but can immediately start generating summaries in the sets S_f^0 and S_f^1.

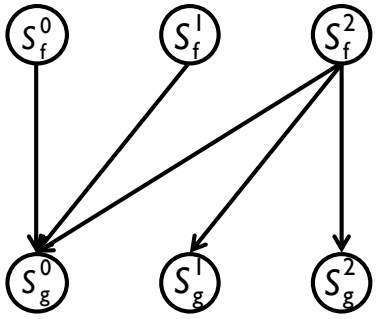

Figure 5.6: The summary dependence graph for a caller-callee function pair, *f* and *g*.

5.3.4 Pipeline Scheduling

As illustrated in Figure 5.6, given a caller-callee function pair, f and g, we have analyzed the dependence relations between the set S_f^i and the set S_g^j and shown that the relaxed dependence provides an opportunity to improve the parallelism of a bottom-up analysis. However, we observe that a key problem here is that there are no dependence relations between the sets S_f^i and S_f^j for a function f, and scheduling the summary-generation tasks for S_f^i and S_f^j in a random order significantly affects the parallelism.

Figure 5.7(a) illustrates the worst scheduling method when only one thread is available for each function, respectively. In the scheduling method, the sets S_f^0 and S_g^0 have the lowest scheduling priority compared to other summary sets. Since all summary sets of the function f depend on the set S_g^0, they have to wait for all summary sets of the function g to generate, which is essentially the same as a conventional bottom-up analysis.

Thus, to maximize the parallel performance, given any function g, we need to determine the scheduling priority of the sets S_g^0, S_g^1, and S_g^2. First, as shown in Figure 5.6, since more summary sets depend on the set S_g^0 than the sets S_g^1 and S_g^2, scheduling the summary-generation task for the set S_g^0 in a higher priority will release more tasks for other summary sets.

Figures 5.7(b) and 5.7(c) illustrate the two possible scheduling methods when for any function g, the set S_g^0 is in the highest priority. In Figure 5.7(b), the set S_g^2 has a higher priority than the set S_g^1. Since the set S_f^2 depends on the sets S_g^0, S_g^1, and S_g^2, it has to wait for all summaries of the function g to generate, leading to a sub-optimal scheduling method. In contrast, Figure 5.7(c) illustrates the best case where the summary-generation tasks are adequately pipelined.

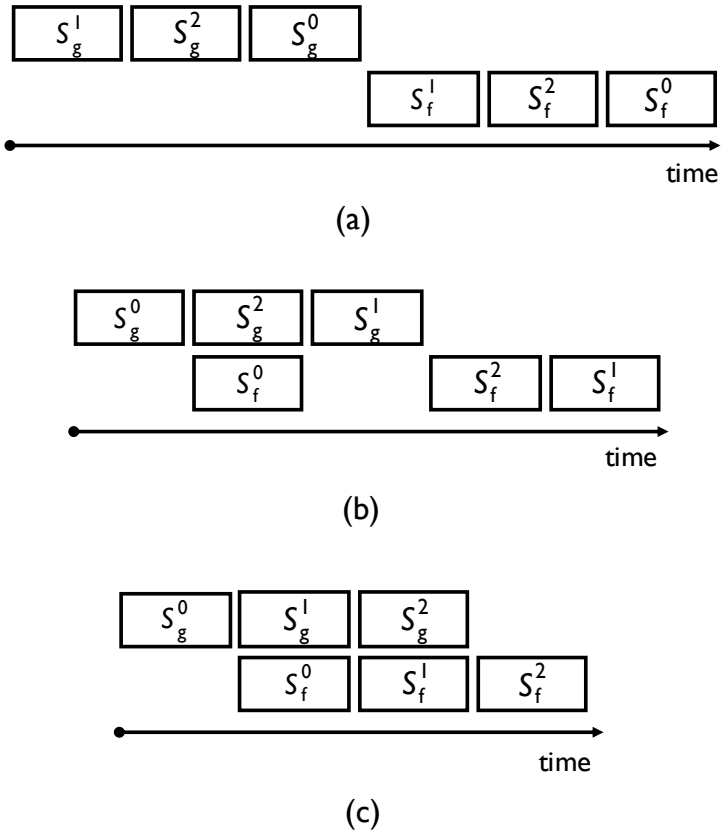

Figure 5.7: Different scheduling methods when one thread available for each function.

To conclude, the scheduling priority for any given function g should be $S_g^0 > S_g^1 > S_g^2$, so that the parallelism of a bottom-up analysis can be effectively improved when a limited number of idle threads are available. Such prioritization does not affect the parallelism when there are enough idle threads available.

5.3.5 ϵ-Bounded Partition and Scheduling

Ideally, the aforementioned partition method evenly partitions a summary set so that the analysis tasks for generating summaries are adequately pipelined, as shown in Figure 5.7(c). However, in practice, it is usually not the case but works as Figure 5.8(a), where the sets S_g^0 and S_g^1 are much larger than other summary sets.

Apparently, if there are extra threads available and we can further partition the summary sets S_g^0 and S_g^1 into two subsets, the analysis performance then will be improved by generating summaries in the subsets in parallel, just as illustrated in Figure 5.8(b). Unfortunately, before a bottom-up analysis finishes, we cannot know

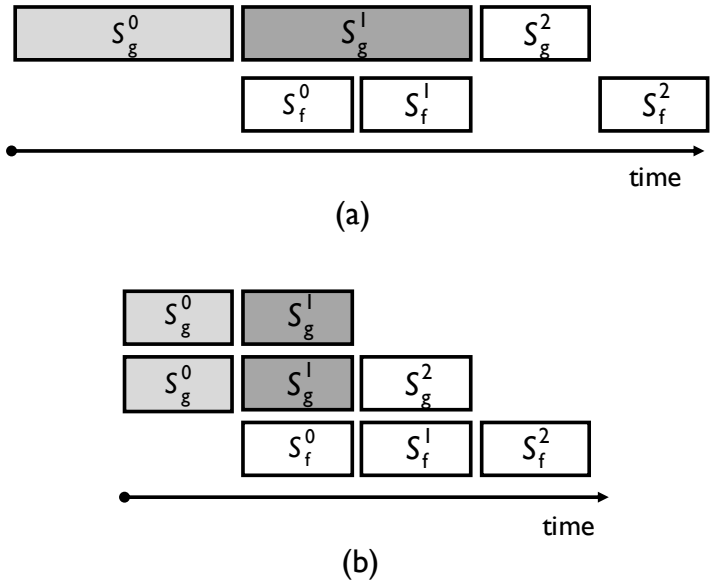

Figure 5.8: Bounded partition and its scheduling method.

the actual size of each summary set and, thus, cannot evenly partition a set. As an alternative, what we can do is to approximate an even partition.

Considering that the analysis task of summary generation is actually to perform a graph traversal from a vertex, we try to further partition a summary set S_f^i based on the number of starting vertices of the graph traversal. To this end, we introduce a client-defined constant ϵ,[3] so that, after the approximately even partition, the graph traversal for generating function summaries in a summary set starts from no more than ϵ vertices.

For example, to generate summaries in the set S_f^0, the analysis needs to traverse the graph G_f from each non-$\mathbf{0}$ data flow fact at the function entry. Suppose the function f has four non-$\mathbf{0}$ data flow facts, $\{\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z}\}$ and $\epsilon = 2$. Then, the set S_f^0 is further partitioned into two subsets $\{(e_f, \mathbf{a}) \rightsquigarrow (x_f, \mathbf{b}) : \mathbf{a} \in \{\mathbf{w}, \mathbf{x}\}\}$ and $\{(e_f, \mathbf{a}) \rightsquigarrow (x_f, \mathbf{b}) : \mathbf{a} \in \{\mathbf{y}, \mathbf{z}\}\}$. After the partition, the graph traversal for both summary sets starts from two vertices.

Similar partition can be performed on the sets S_f^1 and S_f^2 but the following explanation needs to be considered. By definition, it seems difficult to further partition sets S_f^1 and S_f^2 based on the above method, because all summary paths in them start with a single vertex $(e_f, \mathbf{0})$. The key is that, since the fact $\mathbf{0}$ is a tautology and vertices with the fact $\mathbf{0}$ are always reachable from each other [93], the graph traversal to generate summaries in the sets S_f^1 and S_f^2 are not necessary to start

[3]We use $\epsilon = 5$ in our implementation.

from the vertex $(e_f, \mathbf{0})$. For instance, since the set S_f^1 contains the summary paths where data flow facts are created in the function f, we can traverse the graph G_f from every vertex that has an immediate predecessor $(s \in L_f, \mathbf{0})$.[4] Similarly, considering that the set S_f^2 contains the summary paths where data flow facts are created in a callee of the function f, we can traverse the graph G_f from every vertex that has an incoming edge from the callees. With multiple starting vertices for the graph traversal, we then can partition the sets S_f^1 and S_f^2 similarly as the set S_f^0.

It is noteworthy that such a bounded partition aims to parallelize the analysis in a single function and, thus, is applicable to both our pipelining approach and the conventional bottom-up approach. Nevertheless, it is particularly useful to improve the pipeline approach as discussed above.

5.3.6 Pipelining Sparse Value-Flow Analysis

The null analysis and the taint analysis in Cheetah require highly precise pointer information so that they can determine how data flow facts propagate through pointer (load and store) operations. To resolve the pointer relations, we follow the previous work [106] to perform a path-sensitive points-to analysis. The points-to analysis is efficient because it does not exhaustively solve path conditions but records the conditions on the graph edges. When traversing the graph for an analysis, we collect and solve conditions on a path in a demand-driven manner. In Cheetah, we use Z3 [31] as the constraint solver to determine path feasibility. According to our experience and many existing works [106, 125, 37, 8], path-sensitivity is a critical factor to make an analysis practical and make the evaluation closer to a real application scenario. For instance, a path-insensitive null analysis reports >90% false positives and, thus, is impractical.

After building the exploded super-graph with the points-to analysis, we simplify the graph via a program slicing procedure, which removes irrelevant edges and vertices, thereby improving the performance of the subsequent null and taint analyses. This simplification process is almost linear to the graph size and, thus, is very fast [92].

As an example, Figure 5.9(a) is a program where a null pointer is propagated to the variable c through the store and load operations at Line 5 and Line 9. We use the points-to analysis to identify the propagation and build the exploded super-graph as illustrated in Figure 5.9(b). In this graph, the condition of the propagation y and $\neg y$

[4]Recall that an edge from the fact **0** to a non-**0** fact means the non-**0** fact is freshly created.

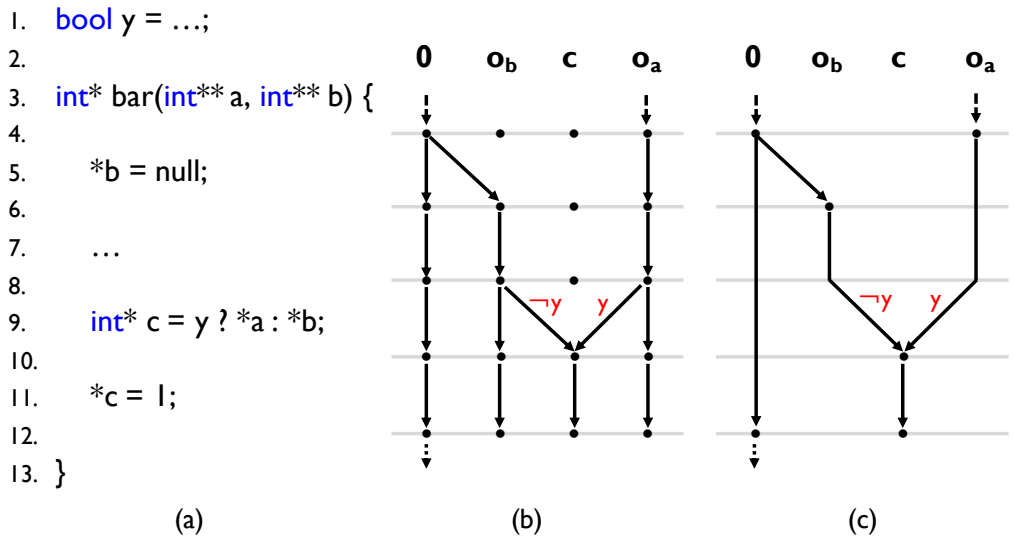

Figure 5.9: Simplifying the exploded super-graph to speedup the analysis.

are labeled on the edges. Figure 5.9(c) illustrates the simplified form of the original graph, where unnecessary edges like $(s_{10}, \mathbf{o_b}) \rightsquigarrow (s_{12}, \mathbf{o_b})$ and unnecessary vertices like $(s_8, \mathbf{o_b})$ are removed. This simplified graph is equivalent to a sparse value-flow graph. Thus, it is easy to apply the pipeline approach to a path-sensitive sparse value-flow analysis like Pinpoint or Catapult.

5.4 Implementation

We have implemented the pipelined IFDS framework, Cheetah, on top of LLVM to path-sensitively analyze C/C++ programs. This section discusses the implementation details. In the evaluation, for a fair comparison, except for the parallel strategy we study in the chapter, all other implementation details are the same in both Cheetah and the baseline approaches.

5.4.1 Parallelization

As illustrated in Figure 5.10, we implement a thread pool to drive our pipeline parallelization strategy. In the figure, the master process cycle maintains the summary dependence graph for all functions. Each vertex in the graph represents a task to generate certain function summaries. Whenever all of the dependent tasks have been completed, it pushes the current task, referred to as the active task, into a queue and

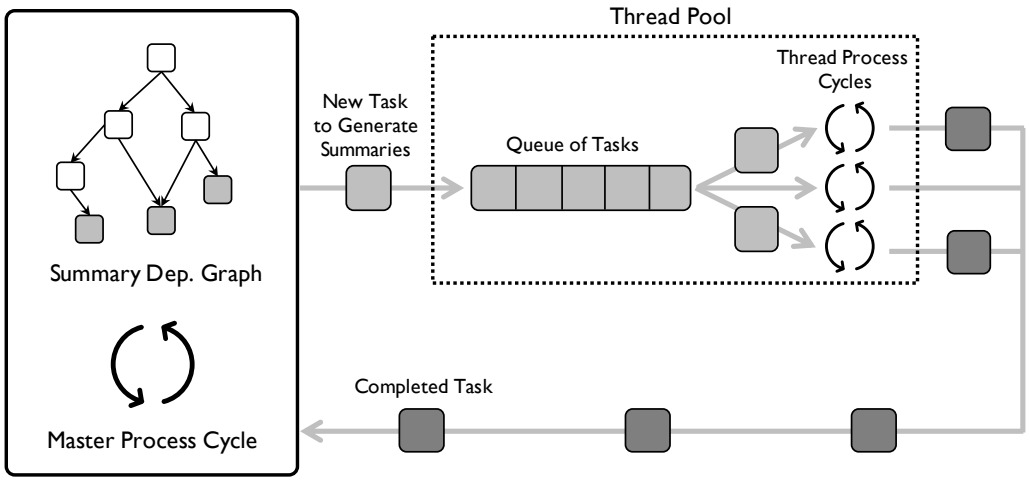

Figure 5.10: Pipelining bottom-up data flow analysis using a thread pool.

waits for an idle thread to consume it. When a task is completed, the master process cycle is notified and continues to find more active tasks on the dependence graph.

In our implementation, instead of randomly scheduling the tasks in the thread pool, we also seek to design a systematic scheduling method so that we can well-utilize CPU resources. However, it is known that generating an optimal schedule to parallelize the computations in a dependence graph is a variant of precedent-constraint scheduling, which is NP-complete [72]. Therefore, we employ a greedy critical path scheduler [79]. A critical path is the longest remaining path from a vertex to the root vertex on the dependence graph. We then replace the task queue in Figure 5.10 with a priority queue and prioritize tasks based on the length of critical paths. It is noteworthy that this heuristic scheduling method does not conflict with the pipeline scheduler presented in Section 5.3.4. The pipeline scheduler prioritizes the analysis tasks in the same function, while the critical-path scheduler only prioritizes the tasks from different functions.

5.4.2 Taint Analysis

To demonstrate that our approach is applicable to a broad range of data flow analysis, in addition to the null analysis discussed in the chapter, we also implement a taint analysis to check two kinds of taint issues. First, we check *relative path traversal*, which allows an attacker to access files outside of a restricted directory. It is modeled as a path on the exploded super-graph from an external input to a file operation. A typical example is a path from a user input *input=gets(...)* to a file

operation *fopen(...)*. Second, we check *transmission of private resources*, which may leak private data to attackers. It is modeled as a path on the exploded super-graph from sensitive data to I/O operations. A typical example is a path from the password *password=getpass(...)* to an I/O operation *sendmsg(...)*.

5.4.3 Soundness

Our implementation of Cheetah is soundy [76], meaning that it handles most language features in a sound manner while we also make some well-identified unsound choices following the previous work [125, 21, 8, 115, 106]. Note that Cheetah aims to find as many bugs as possible rather than rigorously verifying the correctness of a program. In this context, the unsound choices have limited negative impacts as demonstrated in the previous works. In our implementation, like the previous work [56], we use a flow-insensitive pointer analysis [110] to resolve function pointers. We unroll each cycle twice on both the call graph and the control flow graph [8]. Following the work of Saturn [125], a representative static bug detection tool, we do not model inline assembly and library utilities such as *std::vector*, *std::set*, and *std::map* from the C++ standard template library.

5.5 Evaluation

Our goal is to study the scalability of Cheetah, a pipeline parallelization strategy for bottom-up data flow analysis. We did this by measuring the CPU utilization rates and the speedup over a conventional parallel implementation of our bottom-up analysis. More specifically, a conventional parallel implementation only analyzes functions without calling dependence in parallel, just as illustrated in Figure 5.1. To precisely measure and study the scalability of our approach, we introduce an artificial throttle that allows us to switch between our pipeline strategy and the conventional parallel strategy. In this manner, we can guarantee that, except for the parallel strategies, all other implementation details discussed in Section 5.4 are the same for both our approach and the baseline approach. For instance, both approaches accept the same exploded super-graph as the input. Particularly, as discussed in Section 5.3.5, since the ϵ-bounded partition aims to parallelize the analysis in a single function, it is adopted in both our approach and the baseline approach for a fair comparison. Therefore, the speedup of our approach demonstrated in this section is achieved by the pipeline strategy, i.e., the key contribution of this chapter, alone.

Table 5.1: Subjects for evaluation.

Origin	ID	Program	Size (KLoC)	# Functions
SPEC CINT2000	1	mcf	2	26
	2	bzip2	3	74
	3	gzip	6	89
	4	parser	8	324
	5	vpr	11	272
	6	crafty	13	108
	7	twolf	18	191
	8	eon	22	3,367
	9	gap	36	843
	10	vortex	49	923
	11	perlbmk	73	1,069
	12	gcc	135	2,220
Open Source	13	bftpd	5	260
	14	shadowsocks	32	574
	15	webassembly	75	7,842
	16	redis	101	1,527
	17	python	434	3,619
	18	icu	537	27,046
	19	openssl	791	11,759
	20	mysql	2,030	79,263
			Total 4,381	**Avg.** 7,070

Like the previous work [3], we did not compare our implementation with other tools like Saturn [125] and Calysto [8]. This is because the comparison results will not make any sense due to a lot of different implementation details that may affect the runtime performance.

Our evaluation of Cheetah was over the standard SPEC CINT2000 benchmarks, which is commonly used in the literature on static analysis [115, 106]. We also include eight industrial-sized open-source C/C++ projects such as Python, OpenSSL, and MySQL. These real-world subjects are the monthly trending projects on Github that we are able to set up. Table 5.1 lists the evaluation subjects. The size of these subjects is more than four million lines of code in total, ranging from a few thousand to two million lines of code. The number of functions of these subjects ranges from tens to nearly eighty thousand functions, with about seven thousand on average.

We ran our experiments on a server with eighty "Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz" processors and 256GB of memory running Ubuntu-16.04. We set our initial number of threads to be 20 and add 20 for every subsequent run until the maximum number of available processors, i.e., 80. All the experiments were run with the resource limitation of 12 hours.

Table 5.2: Running time (seconds) and the speedup over the conventional method.

ID	# Thread = 20			# Thread = 40			# Thread = 80		
	Conv	Pipe	Speedup	Conv	Pipe	Speedup	Conv	Pipe	**Speedup**
1	60	28	2.1×	60	24	2.5×	60	20	**3.0**×
2	108	64	1.7×	96	40	2.4×	96	32	**3.0**×
3	168	76	2.2×	168	61	2.8×	168	56	**3.0**×
4	252	215	1.2×	168	120	1.4×	132	72	**1.8**×
5	264	192	1.4×	180	116	1.6×	144	76	**1.9**×
6	192	104	1.8×	168	76	2.2×	168	60	**2.8**×
7	168	132	1.3×	133	80	1.7×	122	56	**2.2**×
8	2568	2148	1.2×	1620	1192	1.4×	1128	708	**1.6**×
9	1728	860	2.0×	1524	648	2.4×	1476	545	**2.7**×
10	843	648	1.3×	698	374	1.9×	662	252	**2.6**×
11	1530	913	1.7×	1325	604	2.2×	1217	500	**2.4**×
12	1978	1573	1.3×	1486	926	1.6×	1235	613	**2.0**×
13	156	109	1.4×	132	68	1.9×	132	44	**3.0**×
14	876	468	1.9×	780	340	2.3×	768	288	**2.7**×
15	2940	1990	1.5×	2292	1248	1.8×	1980	908	**2.2**×
16	1332	1060	1.3×	984	628	1.6×	864	416	**2.1**×
17	5162	3022	1.7×	4276	2036	2.1×	3895	1605	**2.4**×
18	7.8hr	5.5hr	1.4×	5.8hr	3.4hr	1.7×	4.9hr	2.3hr	**2.1**×
19	2.8hr	2.2hr	1.2×	1.9hr	1.2hr	1.6×	1.6hr	0.8hr	**2.0**×
20	T/O	9.6hr	-	T/O	7.8hr	-	11.8hr	5.6hr	**2.1**×

T/O: Time Out (>12 hours)

5.5.1 Study of the Null Analysis

We first present the experimental results of the null analysis in detail and then briefly explain the experimental results of the taint analysis in the next subsection.

(1) Speedup. Table 5.2 lists the comparison results of the conventional parallel mechanism (Conv) and our pipeline strategy (Pipe) for the bottom-up program analysis. Each row of the table represents the results of a benchmark program, including the time costs (in seconds) and the speedup for these two kinds of parallel mechanisms. The speedup is calculated as the ratio of the time taken by Cheetah to that of the conventional parallel approach with the same number of threads.

We observe that the speedup achieved with 20 threads is 1.5× on average. However, as the number of threads is increased to 80, the observed speedup also increases, up to 3× faster. Using several typical examples, Figure 5.11 illustrates the relation between the number of threads and the speedup. The growing curves show that the speedup increases with the number of available threads, demonstrating that we can always achieve speedup and have higher parallelism than the conventional parallel approach.

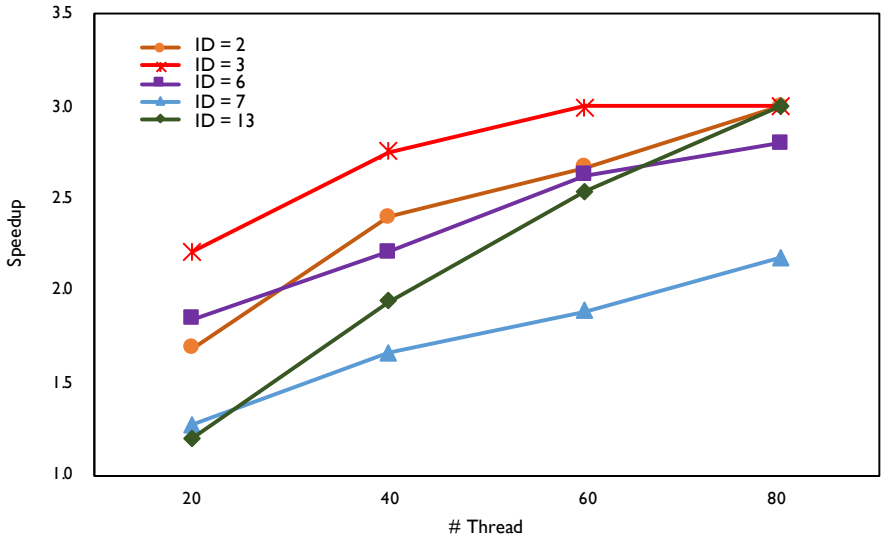

Figure 5.11: Speedup vs. The number of threads.

It is noteworthy that such 2×-3× speedup is significant enough to make many overly lengthy analyses useful in practice. For example, originally, it takes more than 10 hours to analyze MySQL (ID = 20, Size = 2 MLoC, typical size in industry). Such a time cost cannot satisfy the industrial requirement of finishing analysis in 5 to 10 hours [79]. With the pipeline strategy, it saves more than 6 hours, making the bug finding task acceptable in the industrial setting.

(2) CPU Utilization Rate The speedup over conventional parallel design is due to the higher parallelism achieved by the pipeline strategy. To quantify this effect, we profile the CPU utilization rates for both the conventional parallel design and the pipeline method. Figure 5.12 demonstrates the CPU utilization rates against the elapsed running time. Due to the page limit, we only show several typical ones for some of the programs running with 80 threads. In the figure, the solid line represents the CPU utilization rate of our pipeline method while the dashed line represents that of the conventional parallel design.

We can observe that, for each project, in the initial phase of the analysis, the CPU utilization rates for both parallel designs are similar, almost occupying all available CPUs. This is because the call graph of a program is usually a tree-like data structure. In the bottom half of the call graph, it usually has enough independent functions that we can analyze in parallel. Thus, both parallel designs can sufficiently utilize the CPUs.

Our pipeline strategy unleashes its power in the remaining part of the analysis, where it apparently has much higher CPU utilization rates, thus finishing the analysis

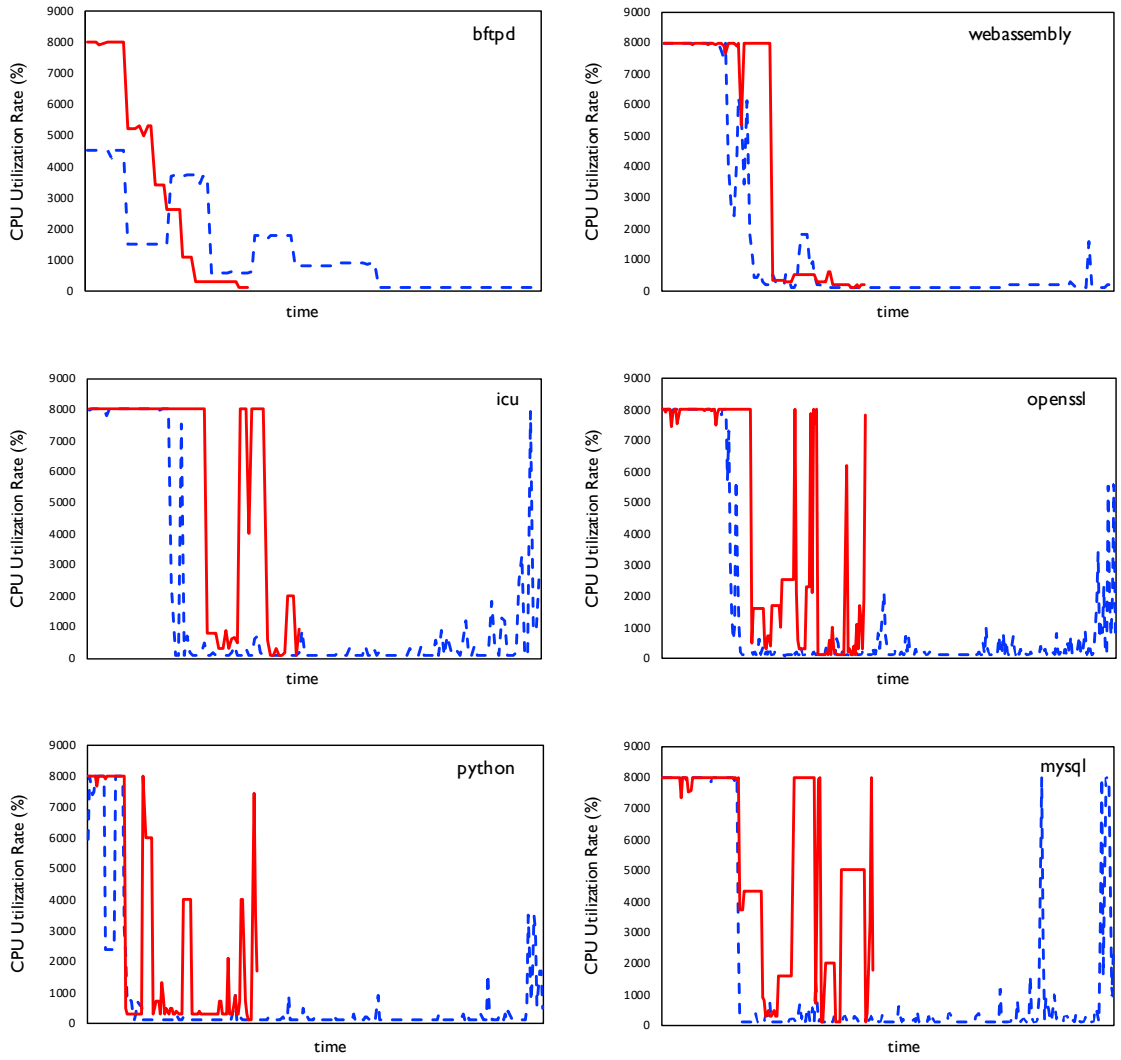

Figure 5.12: The CPU utilization rate vs. The elapsed time.

much earlier. This is because the top half of a call graph is much denser, where there are more calling relations than the bottom half. Since the conventional parallel design cannot analyze functions with calling relations in parallel, it cannot sufficiently utilize the CPUs. In contrast, our approach splits the analysis of a function into multiple parts and allows us to analyze functions with calling relations in parallel, thus being able to utilize more CPUs.

5.5.2 Study of the Taint Analysis

In order to demonstrate that our approach is generalizable to other analyses, we also conducted an experiment to see whether the pipeline approach can improve the scalability of taint analysis. Since the result of taint analysis are quite similar to

Table 5.3: Results of the taint analysis on MySQL.

	# Thread = 20			# Thread = 40			# Thread = 80		
	Conv	Pipe	Speedup	Conv	Pipe	Speedup	Conv	Pipe	**Speedup**
RPT	T/O	10.2hr	-	T/O	8.7hr	-	10.9hr	4.7hr	**2.3**×
TPR	9.3hr	6.6hr	1.4×	8.1hr	5.0hr	1.6×	6.1hr	2.8hr	**2.2**×

T/O: Time Out (>12 hours)

that of the null analysis, we briefly summarize the experimental results in Table 5.3, where the results of our largest benchmark program, MySQL, are presented. The results show that, with the increase of the number of threads, the speedup of Cheetah over the conventional approach also grows to >2× in analyzing both the *relative path traversal* (RPT) bug or the *transmission of private resources* (TPR) bug.

5.5.3 Discussion

There are two main factors affecting the evaluation results: the density of the call graph and the number of available threads.

As discussed above, when the call graph is very sparse, the advantage of our approach is not very obvious. For instance, if functions are all independent on each other, all functions can be run in parallel. Thus, both approaches can always sufficiently utilize the available threads and, thus, have similar time efficiency. In practice, as demonstrated in our evaluation, the call graph is usually tree-like. Thus, our approach can present its power in the second half of the analysis and achieves up to 3× speedup in practice.

The number of threads is also a key factor affecting the observed speedup of our approach. For instance, if we only have one thread available, although our approach can provide more independent tasks, these tasks cannot be run in parallel. Thus, both of our approach and the conventional one will emit similar results. As illustrated by the evaluation, our approach can work better when we have more available threads. In the cloud era, we can expect that we have unlimited CPU resources and, thus, can expect more benefits from our approach in practice.

5.6 Conclusion

We have presented Cheetah, a pipeline parallelization strategy that enables to perform bottom-up data flow analysis in a faster way. The pipeline strategy relaxes

the calling dependence, which conventionally limits the parallelism of bottom-up analysis. The evaluation of our approach demonstrates higher CPU utilization rates and significant speedup over a conventional parallel design. In the multi-core era, we believe that improving the parallelism is an important approach to scaling static program analysis.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Although there have been many success stories for developers using static bug finding tools, we still observe the difficulty of applying them at industrial scale. In this thesis, we present our novel designs of sparse value-flow analysis to tackle a wide range of software bugs caused by improper value flows. The proposed approach is being commercialized and has been deployed in many of the global 500 companies in China, such as Tencent, Alibaba and others. It also has reported hundreds of real bugs for large-scale open-source software systems, which are frequently checked by commercial or free static code analyzers and, thus, are expected to have high quality. At the time of writing, many of the detected bugs have been assigned CVE IDs due to their severity and security impact.

Towards the scalability problem of building precise data dependence through a points-to analysis, this thesis proposes Pinpoint, a holistic design of SVFA. Instead of hiding points-to analyses behind points-to query interfaces, we create an analysis slice, including points-to queries, value flows, and path conditions, that is just sufficient for the properties to check. In this manner, we can escape from the pointer trap by precisely discovering local data dependence first and delaying the expensive inter-procedural data dependence analysis through symbolically memorizing the non-local data dependence relations and path conditions. At the bug detection step, only the relevant parts of these mementos are further "carved out" in a demand-driven way to go for a high precision. Experiments show that Pinpoint can check the use-after-free bugs or the taint issues in two million lines of code within 1.5 hours. The overall false positive rate is also very low, ranging from 14.3% to 23.6%.

Towards the extensional scalability problem caused by the rapid growth of the number of bug types, this thesis proposes Catapult, a new demand-driven and compositional SVFA with the precision of path-sensitivity. The key design of this technique is to leverage the inter-property awareness and to capture redundancies and inconsistencies when many value-flow properties are checked at the same time. In our analysis, the core static analysis engine shares the analysis results on path-reachability and path-feasibility among different properties to reduce redundant graph traversals and unnecessary invocations of the SMT solver. The experimental results demonstrate that Catapult is more than 8× faster than Pinpoint but consumes only 1/7 of the memory when checking twenty common value-flow properties together.

Towards the limit of parallelization in the conventional parallel design of bottom-up data flow analysis, this thesis proposes Cheetah, an approach to improving the parallelism by relaxing the calling dependence between functions. Our basic idea is to partition the analysis task of a function into multiple sub-tasks, so that we can pipeline the sub-tasks to generate function summaries. We formally prove the correctness of our approach and apply it to a null analysis and a taint analysis to show its generalizability. Overall, our pipeline strategy achieves 2× to 3× speedup over a conventional parallel design of bottom-up analysis. Such speedup is significant enough to make many overly lengthy analyses useful in practice.

6.2 Future Work

In the course of my research, I have noticed that existing static code analyzers still suffer from the scalability or precision issues when analyzing large-scale software systems. This is caused by many reasons and we summarize two significant ones that inspire my future research. First, few modern software systems are built from scratch. They rely on a large number of external libraries with various versions. Without the knowledge on the external code, it is challenging to analyze a program with high precision. Even with the knowledge, it is still challenging to analyze a program efficiently because the search space of the program, including a great deal of external code, is extremely large. Second, a large-scale system often consists of many components written in different programming languages, which have different memory models, different type systems, and so on. These differences make it challenging to perform a conventional data flow analysis, because we have a lack of uniformed methods to manage different language semantics. I envisage that my future research will start from addressing the above issues and, eventually, produce

the next-generation industrial-strength static security analyzer. In what follows, I briefly introduce my future plan.

(1) Data-Driven and Cloud-Based Program Security Analysis. To address the problem caused by external libraries, we need to persist the pre-analyzed results of the libraries in a database and query related analysis results whenever necessary. Since there are a huge number of external libraries to analyze and store, and each library also has many versions, I envision that it will be very challenging to build such a big-code warehouse with the capability of distributive storage and fast query of the analysis results. To the best of our knowledge, only a few existing studies attempt to manually model a very limited number of libraries, which is different from my future plan because they only handle limited data in an ad-hoc manner while I am aiming for a cloud-based big-code warehouse that support general program security analysis. With the big-code warehouse and an efficient query scheme, when encountering external library functions during a program security analysis, we can query the pre-analyzed results stored in the cloud so that the analysis can efficiently and precisely recognize the semantics of the library functions.

(2) Program Security Analysis for Complex Systems. My second future research direction is to study the analysis of a complex software system written in hybrid languages. Modern software, such as mobile applications, is usually written in multiple languages, such as Java, Javascript, C/C++, and so on. The memory of Java and Javascript is auto-managed while the memory of C/C++ has to be manually managed. Java and C/C++ are static-type languages while Javascript is a dynamic-type language. Java is a strong-type language while C/C++ and Javascript are weak-type languages. I plan to build uniform data structures to model the memory and the type systems so that data flow across different languages can be safely analyzed with high precision.

Publications

Thesis related publications

- **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 **PLDI 2018**: the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. Philadelphia, Pennsylvania, United States. June 2018.
- **Qingkai Shi**, and Charles Zhang. Pipelining Bottom-up Data Flow Analysis. In **ICSE 2020**: the 41st ACM/IEEE International Conference on Software Engineering. Seoul, South Korea. May 2020.
- **Qingkai Shi**, Rongxin Wu, Gang Fan, and Charles Zhang. Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks. In **ICSE 2020**: the 41st ACM/IEEE International Conference on Software Engineering. Seoul, South Korea. May 2020.

Other publications

- Gang Fan, Rongxin Wu, **Qingkai Shi**, Xiao Xiao, Jinguo Zhou, and Charles Zhang. SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. In **ICSE 2019**: the 41st ACM/IEEE International Conference on Software Engineering. Montreal, QC, Canada. May 2019. (**ACM SIGSOFT Distinguished Paper Award**)
- Heqing Huang, Peisen Yao, Rongxin Wu, **Qingkai Shi**, and Charles Zhang. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In **Oakland 2020**: the 41st IEEE Symposium on Security and Privacy. San Francisco, CA, United States. May 2020.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

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

[3] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani. Parallelizing top-down interprocedural analyses. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 217–228. ACM, 2012.

[4] N. Allen, P. Krishnan, and B. Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP '15, pages 13–18. ACM, 2015.

[5] 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*, pages 349–361. ACM, 2016.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269. ACM, 2014.

[7] 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*, ECOOP '16, pages 2:1–2:25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[8] D. Babic and A. 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.

[9] 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*, POPL '02, pages 1–3. ACM, 2002.

[10] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7):68–76, 2011.

[11] J. Barnat, L. Brim, and J. Stříbrná. Distributed ltl model-checking in spin. In *International SPIN Workshop on Model Checking of Software*, pages 200–216. Springer, 2001.

[12] P. Beanie, H. Kautz, and A. Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI '03, pages 1194–1201. Morgan Kaufmann Publishers Inc., 2003.

[13] 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*, 53 (2):66–75, 2010.

[14] F. E. Boland Jr and P. E. Black. The juliet 1.1 c/c++ and java test suite. *Computer*, 45(10), 2012.

[15] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262. ACM, 2009.

[16] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. ACM, 2012.

[17] G. Cabodi and S. Nocco. Optimized model checking of multiple properties. In *2011 Design, Automation, and Test in Europe Conference*, DATE '11, pages 1–4. IEEE, 2011.

[18] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *Journal of ACM*, 58(6):26:1–26:66, 2011.

[19] 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*, pages 313–325. Springer, 2014.

[20] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. *IEEE Transactions on Software Engineering*, 30 (6):388–402, 2004.

[21] 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*, PLDI '07, pages 480–491. ACM, 2007.

[22] C. Y. Cho, V. D'Silva, and D. Song. Blitz: Compositional bounded model checking for real-world programs. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, pages 136–146. IEEE, 2013.

[23] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic (TOCL)*, 12(1):7, 2010.

[24] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[25] 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*, pages 368–371. ACM, 2003.

[26] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design*, 25(2):105–127, 2004.

[27] P. Cousot and R. Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, CC '02, pages 159–179. Springer, 2002.

[28] 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*, pages 25–35. ACM, 1989.

[29] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[30] 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*, PLDI '02, pages 57–68. ACM, 2002.

[31] 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*, pages 337–340. Springer, 2008.

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

[33] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[34] N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Theory and Applications of Satisfiability Testing*, SAT '06, pages 36–41. Springer, 2006.

[35] D. Dewey, B. Reaves, and P. Traynor. Uncovering use-after-free conditions in compiled code. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 90–99. IEEE, 2015.

[36] K. Dewey, V. Kashyap, and B. Hardekopf. A parallel abstract interpreter for javascript. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*, pages 34–45. IEEE, 2015.

[37] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 270–280. ACM, 2008.

[38] I. Dillig, T. Dillig, A. Aiken, and M. 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.

[39] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 307–317. ACM, 2017.

[40] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 12–22. ACM, 2004.

[41] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 3–12. IEEE, 2007.

[42] M. Edvinsson, J. Lundberg, and W. Löwe. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 45–54. ACM, 2011.

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

[44] J. Feist, L. Mounier, and M.-L. Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3): 211–217, 2014.

[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[46] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008.

[47] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 143–152. ACM, 1990.

[48] D. Garbervetsky, E. Zoppi, and B. Livshits. Toward full elasticity in distributed static analysis: the case of callgraph analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, FSE '17, pages 442–453. ACM, 2017.

[49] E. Goldberg, M. Güdemann, D. Kroening, and R. Mukherjee. Efficient verification of multi-property designs (the benefit of wrong assumptions). In *2018 Design, Automation, and Test in Europe Conference*, DATE '18, pages 43–48. IEEE, 2018.

[50] N. Grech and Y. Smaragdakis. P/taint: Unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017.

[51] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 129–145. Springer, 2005.

[52] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187. ACM, 2011.

[53] S. Guyer and C. Lin. Client-driven pointer analysis. *Static Analysis*, pages 1073–1073, 2003.

[54] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2):83–114, 2005.

[55] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 226–238. ACM, 2009.

[56] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 289–298. IEEE, 2011.

[57] B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. Lu. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP '17, pages 13–18. ACM, 2017.

[58] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 24–34. ACM, 2001.

[59] 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*, POPL '02, pages 58–70. ACM, 2002.

[60] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[61] G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10): 659–674, 2007.

[62] 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*, pages 9–14. ACM, 2007.

[63] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 13–19. ACM, 2005.

[64] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 87–96. Morgan Kaufmann Publishers Inc., 1989.

[65] H. Jordan, P. Subotić, D. Zhao, and B. Scholz. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 327–339. ACM, 2019.

[66] 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*, pages 194–206. ACM, 1973.

[67] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[68] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE, 2004.

[69] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289. ACM, 2007.

[70] W. Le and M. L. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 272–282. ACM, 2008.

[71] Y.-f. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. In *Proceedings of the 6th International Conference on Supercomputing*, pages 236–247. ACM, 1992.

[72] J. K. Lenstra and A. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[73] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 343–353. ACM, 2011.

[74] B. Liu, J. Huang, and L. Rauchwerger. Rethinking incremental and parallel pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1):6, 2019.

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

[76] 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*, 58(2):44–46, 2015.

[77] N. P. Lopes and A. Rybalchenko. Distributed and predictable software model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 340–355. Springer, 2011.

[78] C. A. Martínez-Angeles, I. Dutra, V. S. Costa, and J. Buenabad-Chávez. A datalog engine for gpus. In *Declarative Programming and Knowledge Management*, pages 152–168. Springer, 2013.

[79] 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*, ESEC/FSE '13, pages 554–564. ACM, 2013.

[80] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object*

Oriented Programming Systems Languages and Applications, OOPSLA '10, pages 428–443. ACM, 2010.

[81] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116. ACM, 2012.

[82] D. Monniaux. The parallel implementation of the astrée static analyzer. In *Asian Symposium on Programming Languages and Systems*, pages 86–96. Springer, 2005.

[83] V. O. Mordan and V. S. Mutilin. Checking several requirements at once by cegar. *Programming and Computer Software*, 42(4):225–238, 2016.

[84] N. A. Naeem and O. Lhotak. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 347–366. ACM, 2008.

[85] N. A. Naeem and O. Lhoták. Efficient alias set analysis using ssa form. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 79–88. ACM, 2009.

[86] N. A. Naeem and O. Lhoták. Faster alias set analysis using summaries. In *CC*, pages 82–103. Springer, 2011.

[87] V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 19–28. IEEE, 2013.

[88] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security '13)*, pages 543–558. USENIX Association, 2013.

[89] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for c-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 229–238. ACM, 2012.

[90] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. Eigencfa: Accelerating flow analysis with gpus. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 511–522. ACM, 2011.

[91] S. Putta and R. Nasre. Parallel replication-based points-to analysis. In *International Conference on Compiler Construction*, CC '12, pages 61–80. Springer, 2012.

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

[93] 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*, POPL '95, pages 49–61. ACM, 1995.

[94] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *International Static Analysis Symposium*, pages 284–302. Springer, 2005.

[95] W.-S. Rödiger. *Merging Static Analysis and model checking for improved security vulnerability detection*. PhD thesis, Master thesis, Dept. of Com. Sc. Augsburg University, 2011.

[96] J. Rodriguez and O. Lhoták. Actor-based parallel dataflow analysis. In *International Conference on Compiler Construction*, CC '11, pages 179–197. Springer, 2011.

[97] A. Rountev, M. Sharp, and G. Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, CC '08, pages 53–68. Springer, 2008.

[98] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 598–608. IEEE, 2015.

[99] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[100] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167 (1):131–170, 1996.

[101] D. Saha and C. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on principles and practice of declarative programming*, pages 117–128. ACM, 2005.

[102] L. Sandra. Phb practical handbook of curve fitting, 1994.

[103] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. On fast large-scale program analysis in datalog. In *International Conference on Compiler Construction*, CC '16, pages 196–206. ACM, 2016.

[104] J. Seib and G. Lausen. Parallelizing datalog programs by generalized pivoting. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 241–251. ACM, 1991.

[105] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *International Datalog 2.0 Workshop*, pages 165–176. Springer, 2012.

[106] 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*, PLDI '18, pages 693–706. ACM, 2018.

[107] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[108] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(4):410–457, 2006.

[109] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76. ACM, 2005.

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

[111] R. E. Strom. Mechanisms for compile-time enforcement of security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 276–284. ACM, 1983.

[112] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[113] Y. Su, D. Ye, and J. Xue. Parallel pointer analysis with cfl-reachability. In *2014 43rd International Conference on Parallel Processing*, pages 451–460. IEEE, 2014.

[114] Y. Sui and J. Xue. Svf: Interprocedural static value-flow analysis in llvm. In *International Conference on Compiler Construction*, CC '16, pages 265–266. ACM, 2016.

[115] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[116] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.

[117] P. Tu and D. Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 47–55. ACM, 1995.

[118] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 389–404. ACM, 2017.

[119] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[120] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144. ACM, 2004.

[121] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12. ACM, 1995.

[122] O. Wolfson and A. Ozeri. A new paradigm for parallel and distributed rule-processing. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 133–142. ACM, 1990.

[123] O. Wolfson and A. Silberschatz. Distributed processing of logic programs. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 329–336. ACM, 1988.

[124] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '05, pages 115–125. ACM, 2005.

[125] 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*, POPL '05, pages 351–363. ACM, 2005.

[126] 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*, ISSTA '11, pages 155–165. ACM, 2011.

[127] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 327–337. IEEE, 2018.

[128] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification*, pages 385–398. Springer, 2008.

[129] M. Yang, A. Shkapsky, and C. Zaniolo. Scaling up the performance of more powerful datalog systems on multicore machines. *The International Journal on Very Large Data Bases*, 26(2):229–248, 2017.

[130] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 221–234. ACM, 2008.

[131] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208. ACM, 2008.

[132] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, G. H. Xu, L. Wang, and X. Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *Proceedings of the 14th European Conference on Computer Systems*, EuroSys '19, pages 38:1–38:17. ACM, 2019.