Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code

Qingkai Shi[1], Xiao Xiao[2], Rongxin Wu[1], Jinguo Zhou[2], Gang Fan[1], Charles Zhang[1]
[1]Hong Kong University of Science and Technology, Hong Kong, China
[2]Sourcebrella, China
{qshiaa,wurongxin,gfan,charlesz}@cse.ust.hk;{xx,jinguo}@sbrella.com

Abstract

When dealing with millions of lines of code, we still cannot have the cake and eat it: sparse value-flow analysis is powerful in checking source-sink problems, but existing work cannot escape from the "pointer trap" – a precise points-to analysis limits its scalability and an imprecise one seriously undermines its precision. 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 of only the necessary inter-procedural data dependence and path feasibility queries, which are then solved by a costly SMT solver. Experiments show that Pinpoint can check programs such as MySQL (around 2 million lines of code) within 1.5 hours. The overall false positive rate is also very low (14.3% - 23.6%). Pinpoint has discovered over forty real bugs in mature and extensively checked open source systems. And the implementation of Pinpoint and all experimental results are freely available.

CCS Concepts • **Software and its engineering → Software verification and validation**;

Keywords Sparse program analysis, error detection, path-sensitive analysis.

ACM Reference Format:
Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192418

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192418

1 Introduction

The analysis of value flows underpins many recent techniques in statically finding bugs such as null pointer deference [3, 30, 31, 54], memory leak [9, 46, 48, 53], use-after-free and double-free [9, 16, 21]. 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.

Techniques following the design of conventional data-flow analysis and symbolic execution, such as the IFDS framework [40], Saturn [54], and Calysto [3], propagate data-flow facts to all program points following control-flow paths. These "dense" analyses are known to have performance problems [9, 39, 48, 50]. For example, a recent report [3] observes that it takes 6 to 11 hours for Saturn [54] and Calysto [3] to check only one property (null pointer dereference, a typical source-sink problem) for programs of 685 KLoC.

Sparse value-flow analysis (SVFA) [9, 36, 44, 46, 48] 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. These techniques are referred to as the "layered" approaches due to the need of discovering data dependence as a priori through an independent points-to analysis. However, since a highly precise points-to analysis is difficult to scale to millions of lines of code [29], these "layered" SVFA techniques 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 [9] and Saber [48]. 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.

In this work, we make no claims of breakthroughs to the innate scalability limitations of points-to 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 points-to 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 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 by Cherem et al [9], we first build a global SVFG labeled with path conditions. To determine the value flow incurred by the expression $f = *ptr$ (Line 8 in function `foo`), a points-to analysis, whether exhaustive or demand-driven, is needed to discover that the pointer ptr can point to any of the five variables in the set $\{a, b, c, d, e\}$, resolving the corresponding calling contexts of `bar` and `qux`, as well as checking the satisfiability of the five path conditions for the points-to relations. After building the SVFG, to find the use-after-free bug, we need to traverse the SVFG starting from the vertex $\texttt{free}(c)$, generating a value-flow path, $\langle \texttt{free}(c), c, f, \texttt{print}(*f)\rangle$, 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 points-to relations, two calling contexts and six path conditions. However, if we take a "holistic" view across the layers of SVFA: points-to analysis, SVFG construction, and bug detection, it is easy to discover that, in this example, only the points-to 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.

In this work, we advocate a novel "holistic" approach to SVFA, in which, 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. We present Pinpoint, a tool that decomposes the cost of high-precision points-to 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. 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 light-weight points-to 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 [3, 9, 36, 48, 54], Pinpoint is soundy [35]. 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 paper makes the following contributions:

- An efficient approach to building precise data dependence without an expensive global points-to analysis (Section 3.1).
- A new type of SVFG, namely symbolic expression graph, which enables efficient path-sensitive analysis (Sections 3.2).
- A demand-driven and compositional approach to detecting bugs that can be modeled as value-flow paths (Section 3.3).
- An implementation and an experiment that evaluates Pinpoint's scalability, precision, and recall (Section 4 and Section 5).

2 Pinpoint in a Nutshell

To find the use-after-free vulnerability in Figure 1(a), we begin with an intra-procedural points-to analysis to analyze each function in a bottom-up manner, where we discover data dependence and function side-effects.[1] 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.

For instance, as illustrated in Figure 2(a), our points-to 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

[1] Side-effects here has a broader meaning, including both referencing and modifying non-local memory locations in a function.

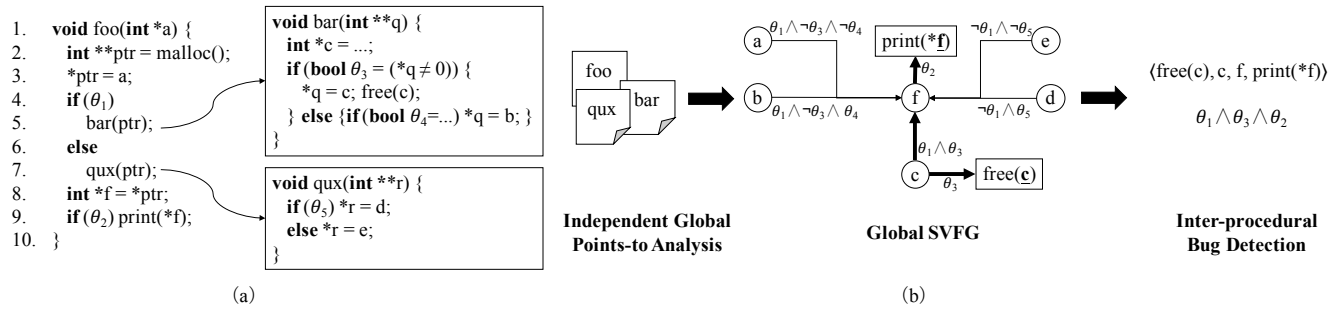

Figure 1. The "layered" design of SVFA. It builds five inter-procedural data-dependence relations between each in $\{a, b, c, d, e\}$ and f. However, only the dependence between c and f is related to the vulnerability.

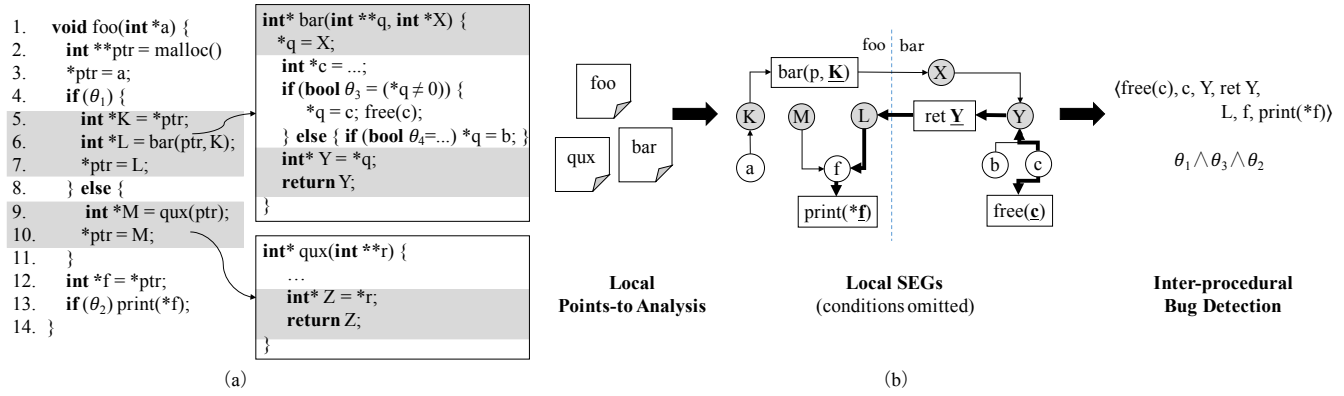

Figure 2. Pinpoint's "holistic" design. The design is "holistic" because, instead of using an independent global points-to analysis to find data dependence, we spread it across all stages of bug finding.

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 points-to analysis in `foo`.

Based on both the local points-to 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 2(b). For example, to build data dependence for the variable f at Line 12, we first obtain ptr's local points-to set, $\{(L, \theta_1), (M, \neg\theta_1)\}$.[2] 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 Section 3.

To detect the use-after-free vulnerability, we traverse the SEG in Figure 2(b) and obtain a complete value-flow path, $\langle \texttt{free}(c), c, Y, \text{return } Y, L, f, \texttt{print}(*f)\rangle$, 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-procedure analysis for bug detection.

3 The Holistic Design in Pinpoint

The key design goal of Pinpoint is to dodge the "pointer traps" incurred by the "layered" design in conventional SVFAs [9, 36, 44, 46, 48]. In this section, we first explain how we decompose the points-to 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 points-to (or data-dependence) relations.

[2] In condition θ_1, ptr points-to L, otherwise, it points-to M.

Language. To present our approach formally, we use the following simple call-by-value language similar to the previous work [17, 18]:

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

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 $k = 1$.

3.1 Decomposing the Cost of Data Dependence Analysis

Building precise SVFGs requires to resolve data dependence through expensive context- and path-sensitive points-to analysis. Our solution is to perform a "quasi" path-sensitive and intra-procedural points-to analysis to resolve both the local data dependence and the function side-effects (a.k.a. MOD-/REF sets [3, 54]). 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.

3.1.1 A Quasi Path-Sensitive Points-to Analysis

We first perform a local points-to 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 points-to 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,[3] denoted by $P(C)$ and $N(C)$, respectively, during the construction of a constraint C. If there exists some atomic constraint $a \in P(C) \cap N(C)$, it means C contains $a \wedge \neg a$ and, thus, is unsatisfiable. $P(C)$ and $N(C)$ are built using the following rules:

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

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 3.2.

3.1.2 A Connector Model for Inter-procedural Data Dependence Analysis

The outcome of the points-to 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 [3, 54], 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 [1].

We noticed that the IFDS/IDE approaches solve this problem much more efficiently [40], 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 2, the circles labeled 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 circles K and L are the call-site connectors for the call

[3] 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 $(x = y + 1) \wedge \neg z$.

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

(a)

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

(b)

Figure 3. Transformation rules. The code starting with ▶ is the target to transform and the result of each rule is the transformation result. (a) Transforming a function by inserting Aux formal parameters and Aux return values. (b) Transforming a call statement according to the signature change of the callee.

statement at Line 6. Then we can connect K to X and L to Y path- and context-sensitively if they are involved in building the inter-procedural data dependence. As described later in Section 3.3, this connector model is sufficient to run a standard value-flow analysis for checking the source-sink properties.

In Pinpoint, the input and output connectors are implemented by two kinds of auxiliary variables: the Aux Formal Parameter and the Aux Return Value.

Definition 3.1. An **Aux 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 **Aux return value** is defined similarly but the non-local memory location is modified.

Figure 3 defines the rules for inserting the auxiliary variables to represent the input and output connectors for functions and call sites. 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 2.

Summary. In summary, the quasi path-sensitive points-to analysis and the connector model enable a holistic design: the points-to 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 points-to analysis are delayed until the bug-finding phase. This holistic result cannot be achieved by an independent points-to analysis, whether exhaustive or demand-driven, in the conventional "layered" design. This is because, being unaware of the properties being checked, an independent points-to 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.

3.2 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:

1. It compactly and precisely encodes all the conditional and unconditional data dependence, as well as the control dependence (Section 3.2.1);
2. It enables the convenient query of the "efficient path conditions [44]" to provide the full support of path-sensitive analysis (Section 3.2.2);
3. It is separately built for each function, not only saving time costs, as described in Section 3.1, but also enabling the efficient compositional analysis (Section 3.3).

3.2.1 Definition

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.
- $\mathcal{E}_d \subseteq (\mathcal{V} \cup \mathcal{O}) \times (\mathcal{V} \cup \mathcal{O})$ 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.

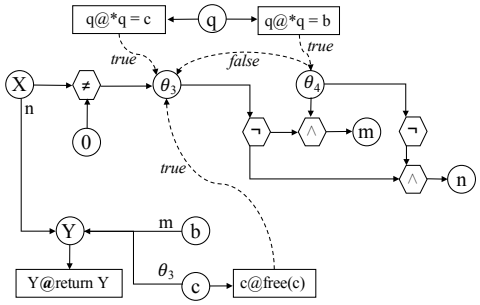

Figure 4. The complete SEG of function bar in Figure 2. Solid edges represent data dependence. The label *true* for unconditional data dependence is omitted. Dashed edges represent control dependence.

Following Definition 3.2, we build the SEG for each function. As an example, the SEG of the function bar in Figure 2 is shown in Figure 4.

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 [9, 46, 48]. Vertices for operators are used to represent symbolic expressions, as illustrated in Example 3.3. 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 3.3. As shown in Figure 4, the expression "$X \neq 0$" is explicitly presented by an operator vertex "≠" and two other vertices standing for its operands, i.e., "X" and "0".

Following the previous work [22], 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 points-to 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 [49]. Example 3.4 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 [22]. The control dependence of a statement is in the form v or $\neg v$ where v is a branch-condition variable. Example 3.5 shows a concrete example of control dependence in SEG.

Example 3.4. In Figure 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 3.3.

Example 3.5. In Figure 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*).

3.2.2 Querying "Efficient Path Conditions" on SEG

The design of SEG enables us to conveniently query the "efficient path condition [44]" of a value-flow path, which is much more succinct than those computed according to the definition of path condition [32]. 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 3.6. Based on the SEG in Figure 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 [32] 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 much more verbose and inefficient.

Given a value-flow path, π=$\langle v_1@s_1, \cdots, v_n@s_n\rangle$, 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. To be clear, for a given vertex, $v@s$, in SEG, we introduce two functions, DD($v@s$) (see Example 3.7) and CD($v@s$) (see Example 3.8), to compute the constraints that describe the data dependence and the control dependence, respectively. The path condition of π can be computed as following:

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

As shown in the above equation, a path condition includes following parts: (1) 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 3.7. Assume we are computing the data dependence of Y shown in Figure 4. 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,

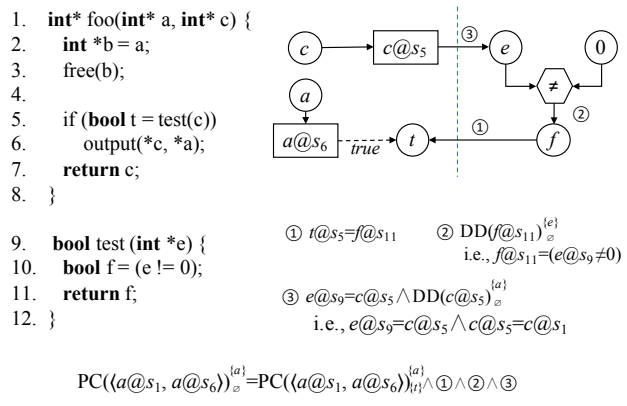

Figure 5. An example to illustrate our inter-procedural analysis. We use s_i to stand for the statement at Line i.

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 3.8. To compute the control dependence of $q@*q = b$, shown in Figure 4, $\text{CD}(q@*q = b)$ results in the constraint: $\theta_4 \wedge \neg\theta_3 \wedge \text{DD}(\theta_4) \wedge \text{DD}(\theta_3)$. 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.

3.3 Global Value Flow Analysis for Vulnerability Detection

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.

3.3.1 Demand-Driven Path- and Context-Sensitive Value Flow Analysis

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 $\text{PC}(\pi)$ (Equation (1) in Section 3.2.2) 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 $\text{PC}(\cdot)$, we rewrite it as $\text{PC}(\cdot)^P_R$ 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 3.9. In Figure 5, for the local value-flow path $\langle a@s_1, a@s_6\rangle$, according to Equation (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 $\text{PC}(\langle a@s_1, a@s_6\rangle)^{\{a@s_1\}}_{\{t@s_5\}} = (t@s_5 = true \wedge a@s_1 = a@s_6)$.

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 $\text{PC}(\cdot)^P_R$, so that it can be written as $\text{PC}(\cdot)^{P'}_{\emptyset}$. 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 3.10. Following the last example, we need to add the constraints of $t@s_5$ into $\text{PC}(\langle a@s_1, a@s_6\rangle)^{\{a@s_1\}}_{\{t@s_5\}}$, so that the dependence on the return-value receiver $t@s_5$ can be eliminated. 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.

In this way, we get the precise path condition, which can be recorded as $\text{PC}(\langle a@s_1, a@s_6\rangle)^{\{a@s_1, c@s_1\}}_{\emptyset}$.

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

$$\text{PC}(\pi)^{P'}_{\emptyset} = \text{PC}(\pi)^P_R \wedge \underbrace{\bigwedge_{v_i@s_i \in R} \underbrace{v_i@s_i = \mathbb{M}(v_i@s_i)}_{①} \wedge \underbrace{\mathbf{DD(\mathbb{M}(v_i@s_i))^{Q_i}_{\emptyset}}}_{②}}_{} \wedge \underbrace{\bigwedge_{v_j@s_j \in Q_i} v_j@s_j = \mathbb{M}(v_j@s_j) \wedge \text{DD}(\mathbb{M}(v_j@s_j))^{P_j}_{\emptyset}}_{③} \tag{2}$$

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. $\text{DD}(\cdot)^{P'}_{\emptyset}$ can be converted from $\text{DD}(\cdot)^P_R$ 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=$\langle v_1@s_1, \cdots, v_n@s_n\rangle$ and π_2=$\langle u_1@r_1, \cdots u_n@r_n\rangle$, 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.

$$\begin{aligned}\mathrm{PC}(\pi_1\pi_2)_{\emptyset}^{P} =& \mathrm{PC}(\pi_1)_{\emptyset}^{P_1} \wedge \mathbf{PC(\pi_2)_{\emptyset}^{P_2}} \wedge v_n@s_n = u_1@r_1 \wedge \\ & \bigwedge_{v_i@s_i \in P_2} v_i@s_i = \mathbb{M}(v_i@s_i) \wedge \mathrm{DD}(\mathbb{M}(v_i@s_i))_{\emptyset}^{Q_i}\end{aligned} \tag{3}$$

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 [34, 51]. 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.

3.3.2 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 [3, 54]. According to the computation of inter-procedural path condition in Section 3.3.1, whenever we analyze a function, we actually require two kinds of information from the callees (see the bold parts in Equations (2) and (3)). One is the data dependence, $\mathrm{DD}(v@s)_{\emptyset}^{P}$, where $v@s$ is a callee's return value. The other is $\mathrm{PC}(\pi)_{\emptyset}^{P}$ 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, $\mathrm{DD}(v@s)_{\emptyset}^{P}$, 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., $\mathrm{DD}(v@s)_{\emptyset}^{P}$.
- A subset P of the function's formal parameters that the constraint depends on.

As described by the path condition, $\mathrm{PC}(\pi)_{\emptyset}^{P}$, 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., $\mathrm{PC}(\pi)_{\emptyset}^{P}$.
- 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 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 5. In order to detect the use-after-free vulnerability, we create a VF3 summary containing the value-flow path $\langle a@s_1, b@s_2, b@s_3\rangle$, 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.

4 Implementation

Pinpoint is implemented on top of LLVM 3.6 [33] using Z3 [14] as the SMT solver. Its main architecture is shown in Figure 6. The implementation is publicly available.[4]

[4] https://whichbug.github.io/artifact.html

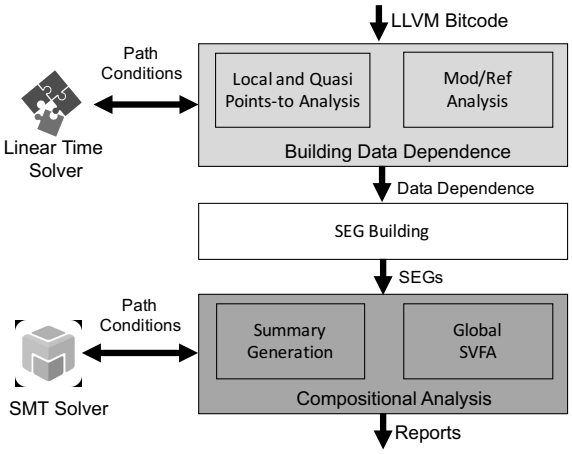

Figure 6. Pinpoint's architecture.

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.[5] 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*, ...) [41]. Similarly, a data transmission vulnerability may leak sensitive data to attackers.[6] 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*, ...) [41]. Similar to the previous taint analysis work [2], we have not modeled the sanitization operations in our taint-issue checkers.

4.2 Soundness

Pinpoint is soundy [35] as it shares the same "standard assumptions" with previous techniques that aim to find bugs rather than rigorous verification [3, 9, 36, 48, 54]. 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 [54], we currently have not modeled inline-assembly and function pointers, but we adopt a class hierarchy analysis to resolve virtual function calls [15]. 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 [52] 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`.

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 compare Pinpoint to the state-of-the-art techniques of SVFA, but also conduct comparison experiments on the tools using abductive inference (Facebook Infer[7]) and symbolic execution (CSA[8]). We also seek to evaluate other prominent static bug detection 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 use include the standard benchmark SPEC INT 2000, 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[9] 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[10] 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 [5, 37].

5.1 Comparing with SVFA Techniques

We compare Pinpoint with the most recent and relevant work, SVF [47], 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

[5] https://cwe.mitre.org/data/definitions/23.html
[6] https://cwe.mitre.org/data/definitions/402.html
[7] http://fbinfer.com/
[8] https://clang-analyzer.llvm.org/
[9] https://scan.coverity.com/projects/
[10] https://cve.mitre.org/

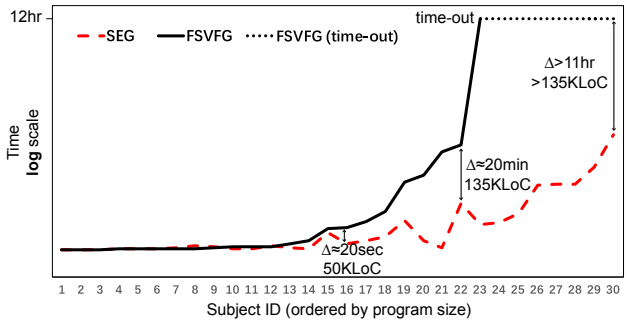

Figure 7. Time cost: building SEG vs. building FSVFG.

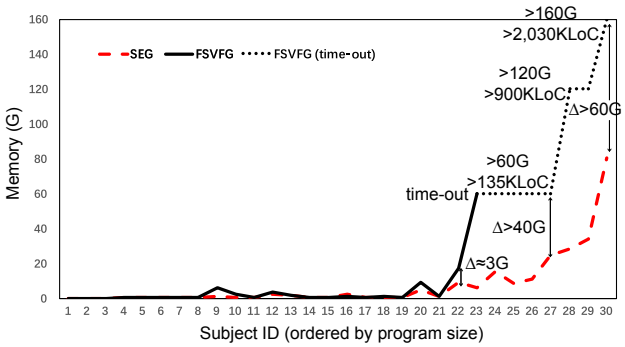

Figure 8. Memory cost: building SEG vs. building FSVFG.

quality of our tool. We do 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 [7]. This allows us to complete our quantification of bug finding capability within a reasonable period of time.

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 paper. Therefore, we compare 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 compare the false positive rates of both checkers. Since we cannot flood developers with all the warnings the tools report, we manually pre-screen 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 use Juliet Test Suite [6], 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 computer with two 20 core processors "Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz" and 256GB physical memory running Ubuntu-16.04.

5.1.1 Scalability

Figure 7 and Figure 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, however, takes less than an hour, up to >400X faster.

Table 1. Results of Use-after-Free Checkers

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.

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 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 9: for the subjects larger than 135 KLoC, SVF uses 10-30G additional memory compared to Pinpoint, while still unable to finish building FSVFG for these subjects.

We adopt the curve fitting approach [43] to study the observed time- and memory-complexity of Pinpoint. Figure 10

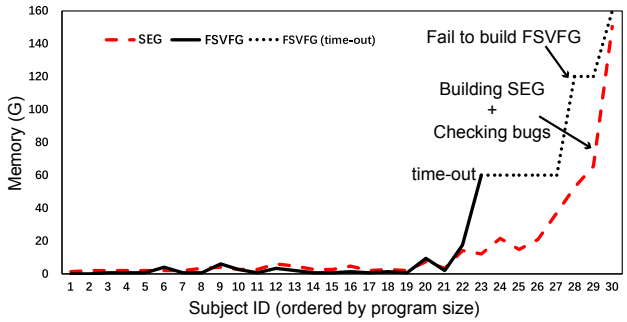

Figure 9. Memory cost: SEG- vs. FSVFG-based checkers.

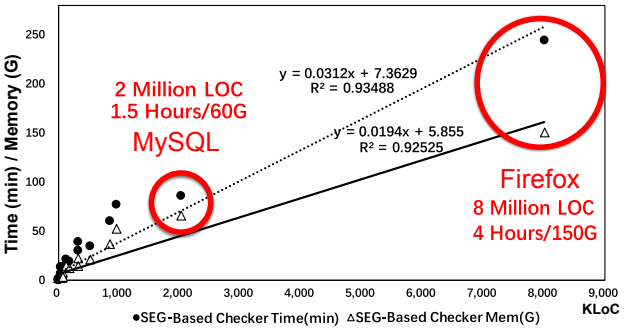

Figure 10. Scalability of an SEG-based checker. The x-axis stands for the number of lines in a project (KLoC) and the y-axis stands for the time cost (min) or the memory cost (G).

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.

5.1.2 Precision and Recall

Pinpoint reports 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. A stark contrast is that Pinpoint generates very few reports in total as shown by Table 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. While there is still subjectivity in this experiment as we can still miss true bugs found by SVF during pre-screening, we make the implementation of SVF-based use-after-free checker available[11] for interested readers to examine. 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.

5.2 Detected Real Vulnerabilities

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[12]), 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[13]), 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 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.

We have made an online list of all the vulnerabilities confirmed by the original software developers.[14] The list contains more than 40 vulnerabilities from about a dozen of open-source projects, including famous software systems like MySQL, FireFox, Python, Apache and OpenSSL, as well as fundamental libraries like LibSSH and LibICU.

5.3 Study of the Taint-Issue Checkers

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 4. The corresponding evaluation results are summarized in the Table 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 [2], 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%.

[11] https://github.com/whichbug/SVF-UAF

[12] https://bugs.mysql.com/bug.php?id=87203

[13] http://bugs.icu-project.org/trac/ticket/13301

[14] https://whichbug.github.io/reports.html

Table 2. Result Summary 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. Results of Infer and CSA

Program	Size (KLoC)	Infer		CSA	
		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		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	
swoole	54			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		1.4	0/0
wrk	340			2.5	0/0
libicu	537	3.3	8/8	2.6	0/0
php	863	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		54.0	5/5
Total			**35/35**		**24/26**

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

5.4 Comparing with Other Static Bug Detectors

To better understand its performance in comparison to other types of bug finding techniques, we run Pinpoint against two prominent and mature open-source static bug detection tools, Infer and CSA, on finding use-after-free vulnerabilities. The results are reported in Table 3. Our evaluation shows both CSA and Infer run faster compared to Pinpoint. The primary reason is that both Infer and CSA 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 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 twenty-six reports by CSA are true positives, which are also reported by Pinpoint.

6 Related Work

Existing techniques for static bug finding can be classified into two major categories by distinguishing the approach to tracking the flow of values: the ones tracking the flow of values via data dependence and the others via control flows.

To the best of our knowledge, all existing static bug-finding techniques utilizing data dependence rely on a pre-computed points-to analysis to build data dependence, which is referred to as a "layered" design. Because a precise points-to analysis is expensive [29], they usually adopt a flow-insensitive analysis to avoid getting stuck in the pre-computation phase [9, 13, 20, 36, 39, 44, 46, 48]. In contrast, the proposed "holistic" design can build precise data dependence efficiently and keep fully path-sensitive for bug finding.

In the other category, abstraction based approach like SLAM [4], BLAST [28], and SATABS [11] adopt abstract refinement to improve scalability. However, the scalability degrades with the refinement of abstraction. CBMC [11, 12] also suffers from the scalability issue because it feeds constraints to an SAT solver regardless of whether they are relevant or not. Cheetah [19], which is built on the IFDS framework [40], is similar to our approach as it does local analysis first and then gradually extends to the whole project. Magic [8], Saturn [17, 53, 54], Calysto [3], Compass [18], and Blitz [10] 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 unnecessary data-flow facts are tracked along control flows [9, 39, 48]. As a non-sparse analysis, they do not suffer from the "pointer trap" problem we attempt to address in this paper.

There are also techniques adopting client- or demand-driven points-to analysis to reduce redundancy in static bug finding techniques. Client-driven points-to analysis [24, 25, 38] 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 [2, 27, 42, 45, 55, 56] is in a fixed precision but computes only the necessary part of the solution. Existing approaches are not path sensitive and only aware of its immediate client. In contrast, the points-to analysis in Pinpoint is aware of the whole process, thus referred to as a "holistic" design. P/Taint [23] also integrates points-to analysis with value-flow analysis, but in a different manner: the value-flow analysis is implemented as an extension of points-to analysis while Pinpoint decomposes the cost of points-to analysis for value-flow analysis.

7 Conclusion

We have described Pinpoint, embodying a holistic design of sparse value-flow analysis that simultaneously achieves precision and observed linear scalability for millions of lines of code. Pinpoint has discovered over forty vulnerabilities, confirmed by developers of about a dozen of well-known systems and code libraries. Pinpoint is promising in providing industrial-strength capability in static bug finding.

8 Acknowledgments

We thank the anonymous reviewers for their insightful comments and Yulei Sui for his help on the SVF implementation. This work was partially funded by Hong Kong GRF16214515, GRF16230716, and ITS/368/14 grants.

References

[1] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2006. *The Saturn Program Analysis System*. Stanford University.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[3] D. Babic and A. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In *2008 ACM/IEEE 30th International Conference on Software Engineering (ICSE 2008)*. IEEE, 211–220.

[4] Thomas Ball and Sriram K. Rajamani. 2002. 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)*. ACM, 1–3.

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[6] Frederick E Boland Jr and Paul E Black. 2012. The Juliet 1.1 C/C++ and Java Test Suite. *Computer (IEEE Computer)* 45, 10 (2012).

[7] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. 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*. ACM, 133–143.

[8] Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2004. Modular verification of software components in C. *IEEE Transactions on Software Engineering* 30, 6 (2004), 388–402.

[9] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-flow Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 480–491.

[10] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. 2013. Blitz: Compositional bounded model checking for real-world programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 136–146.

[11] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. 2004. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design* 25, 2 (2004), 105–127.

[12] Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*. ACM, 368–371.

[13] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, 57–68.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

[16] David Dewey, Bradley Reaves, and Patrick Traynor. 2015. Uncovering Use-After-Free Conditions in Compiled Code. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, 90–99.

[17] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 270–280.

[18] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 567–577.

[19] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 307–317.

[20] N. Dor, S. Adams, M. Das, and Z. Yang. 2004. 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)*. ACM, 12–22.

[21] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques* 10, 3 (2014), 211–217.

[22] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

[23] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 102:1–102:28.

[24] Samuel Guyer and Calvin Lin. 2003. Client-driven pointer analysis. *Static Analysis* (2003), 1073–1073.

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

[26] Brian Hackett and Alex Aiken. 2006. How is Aliasing Used in Systems Software?. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, 69–80.

[27] Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 24–34.

[28] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, 58–70.

[29] Michael Hind. 2001. 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*. ACM, 54–61.

[30] David Hovemeyer and William Pugh. 2007. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 9–14.

[31] David Hovemeyer, Jaime Spacco, and William Pugh. 2005. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGSOFT Software Engineering Notes*, Vol. 31. ACM, 13–19.

[32] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[33] Chris Lattner and Vikram Adve. 2004. 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*. IEEE, 75.

[34] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices* 42, 6 (2007), 278–289.

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

[36] V Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. *ACM SIGSOFT Software Engineering Notes* 28, 5 (2003), 317–326.

[37] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 554–564.

[38] Nomair A Naeem and Ondrej Lhoták. 2011. Faster Alias Set Analysis Using Summaries.. In *CC*. Springer, 82–103.

[39] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 229–238.

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 49–61.

[41] Wolf-Steffen Rödiger. 2011. *Merging Static Analysis and model checking for improved security vulnerability detection*. Ph.D. Dissertation. Master thesis, Dept. of Com. Sc. Augsburg University.

[42] Diptikalyan Saha and CR Ramakrishnan. 2005. 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*. ACM, 117–128.

[43] LA Sandra. 1994. PHB Practical Handbook of Curve Fitting.

[44] G Snelting, T Robschink, and J Krinke. 2006. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 4 (2006), 410–457.

[45] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 59–76.

[46] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 265–266.

[47] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, 265–266.

[48] Y. Sui, D. Ye, and J. Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122.

[49] Peng Tu and David Padua. 1995. Efficient building and placing of gating functions. *ACM SIGPLAN Notices* 30, 6 (1995), 47–55.

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

[51] John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 131–144.

[52] Robert P Wilson and Monica S Lam. 1995. *Efficient context-sensitive pointer analysis for C programs*. Vol. 30. ACM.

[53] Yichen Xie and Alex Aiken. 2005. Context-and path-sensitive memory leak detection. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 115–125.

[54] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, 351–363.

[55] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 155–165.

[56] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. *ACM SIGPLAN Notices* 43, 1 (2008), 197–208.