A Survey of Parametric Static Analysis

JIHYEOK PARK, HONGKI LEE, and SUKYOUNG RYU, KAIST, Daejeon,
Republic of Korea

Understanding program behaviors is important to verify program properties or to optimize programs. Static analysis is a widely used technique to approximate program behaviors via abstract interpretation. To evaluate the quality of static analysis, researchers have used three metrics: performance, precision, and soundness. The static analysis quality depends on the analysis techniques used, but the best combination of such techniques may be different for different programs. To find the best combination of analysis techniques for specific programs, recent work has proposed *parametric static analysis*. It considers static analysis as black-box parameterized by *analysis parameters*, which are techniques that may be configured without analysis details. We formally define the parametric static analysis, and we survey analysis parameters and their parameter selection in the literature. We also discuss open challenges and future directions of the parametric static analysis.

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

Additional Key Words and Phrases: Static analysis, parametric analysis, analysis techniques, abstraction

ACM Reference format:
Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2021. A Survey of Parametric Static Analysis. *ACM Comput. Surv.* 54, 7, Article 149 (July 2021), 37 pages.
https://doi.org/10.1145/3464457

1 INTRODUCTION

Program comprehension is one of the most important topics in software maintenance. It helps developers understand program behaviors to refactor existing code for removing bugs or optimizing programs. However, it is time-consuming: More than 60 percent of manpower in software engineering is invested in program understanding [100]. Thus, researchers have proposed various ways to mechanically extract program behaviors such as static analysis and dynamic analysis.

Static analysis is based on abstract interpretation [13] and approximately estimates program behaviors using abstract semantics. While dynamic analysis [61] collects runtime behaviors of instrumented code, static analysis over-approximates all program behaviors without actually executing programs. The static analysis quality is often evaluated by three criteria:

Jihyeok Park and Hongki Lee contributed equally to the article.
Authors' addresses: J. Park, H. Lee, and S. Ryu, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea; emails: {jhpark0223, petitkan, sryu.cs}@kaist.ac.kr.

0360-0300/2021/07-ART149 $15.00
https://doi.org/10.1145/3464457

— *Performance* denotes how fast static analysis analyzes programs. The literature compared analysis time to measure the analysis performance. The less the analysis time, the more efficient the analysis is.
— *Precision* stands for accuracy of analysis results. Because static analysis performs over-approximation, its results may contain infeasible behaviors. Thus, static analysis may report false positives. To measure the analysis precision, the literature uses the ratio of true positives over the total alarms. The higher the ratio of true positives, the more precise analysis results.
— *Soundness* represents whether analysis results cover all possible program behaviors at runtime. Thus, a sound analysis should not report false negatives. However, practical static analyzers intentionally assume that several complex features, such as reflection in Java and dynamic code generation in JavaScript, do not exist in programs. In such cases, static analysis guarantees soundness only for certain behaviors. When a static analysis allows unsoundness for well-identified features, the analysis ensures *soundiness* [55].

Diverse analysis techniques have been proposed to enhance performance and precision of static analysis while preserving soundness. Several researchers configured them as *analysis parameters* to find the best analysis result depending on the analysis purpose. However, three criteria often compete with each other according to the selected analysis parameters. In most cases, analysis parameters are for the balance between performance and precision. For example, flow-sensitivity is one of the most popular analysis sensitivity techniques. A flow-insensitive analysis maintains a universal abstract state to represent the abstract semantics of the entire program. However, a flow-sensitive analysis stores a local abstract state for each program point. It usually increases the analysis precision but degrades the analysis performance because of abundant abstract states. While most analysis techniques configure the balance between performance and precision, some techniques sacrifice soundness for better precision or performance. When a static analysis approximates complex behaviors using the $\top$ abstract value to preserve soundness, it degrades both precision and performance due to the propagation of imprecise values. Thus, researchers have unsound analyzed real-world programs using features such as dynamic code generation or reflection.

To mechanically select analysis parameters depending on given programs and analysis purposes, researchers have proposed *parametric static analysis*. The parametric static analysis considers a static analyzer as a black box configurable with analysis parameters and focuses on how to select the best analysis parameter. For the parameter selection, existing techniques utilize extracted program properties and feedback from the previous or intermediate results of static analysis for the iterative strategy. In this article, we formally define parametric static analysis for the first time, and we survey the analysis parameters used for parametric static analysis and how existing research selects them depending on given programs and analysis purposes. In Section 2, we formalize the definition of parametric static analysis. We categorize analysis parameters and describe their effects during static analysis in Section 3, and we introduce how to select analysis parameters in Section 4. We discuss open challenges and future research directions in Section 5 and conclude in Section 6.

2 FORMAL DEFINITION OF PARAMETRIC STATIC ANALYSIS

In this section, we formally define which parts of static analysis are parameterizable and explain the overall structure of parametric static analysis.

2.1 Programs and Collecting Semantics

We represent a program $P = (\Sigma, \rightsquigarrow, \Sigma_\iota)$ as a state transition system. A state $\sigma \in \Sigma$ represents a status of the program and Σ_ι denotes the initial state set. The transition relation $\rightsquigarrow \subseteq \Sigma \times \Sigma$ describes how states are transformed to other states. The notation $\rightsquigarrow^*$ is zero or more repetitions of $\rightsquigarrow$.

$\bullet_{\ell_0}$ x = ? ; $\bullet_{\ell_1}$
if (x ≥ 0) $\bullet_{\ell_2}$ x = x;
else $\bullet_{\ell_3}$ x = −x; $\bullet_{\ell_4}$

Fig. 1. Conditional branch.

We define the set of reachable states of P as its *collecting semantics* $[\![P]\!] = \{\sigma \in \Sigma \mid \sigma_\iota \in \Sigma_\iota \wedge \sigma_\iota \rightsquigarrow^* \sigma\}$. One way to systematically evaluate it is to define a step-by-step iteration to collect next reachable states. For such iterations, we define a *concrete domain* $\mathbb{D}$ as a lattice whose elements are sets of states $\mathcal{P}(\Sigma)$, the partial order is the subset relation $\subseteq$, and the least upper bound and the greatest lower bound between sets of states are the set union $\cup$ and the set intersection $\cap$, respectively. Then, we define a *transfer function* $F : \mathbb{D} \to \mathbb{D}$ to gradually collect reachable states:

$$F(d) = d \cup \mathbf{step}(d),$$

where the *one-step execution* $\mathbf{step} : \mathbb{D} \to \mathbb{D}$ transforms each element of the given set of states d using the transition relation $\rightsquigarrow$: $\mathbf{step}(d) = \{\sigma' \mid \sigma \in d \wedge \sigma \rightsquigarrow \sigma'\}$. The transfer function F merges the given set of states d with the next reachable states $\mathbf{step}(d)$. Finally, we define the collecting semantics using the iteration of the transfer function F with the initial set of state $d_\iota = \Sigma_\iota$:

$$[\![P]\!] = \lim_{n\to\infty} F^n(d_\iota).$$

For example, consider the simple code with a conditional branch in Figure 1. In this case, we define states as pairs of control states and the values of the variable x: $\Sigma = \mathbb{L} \times \mathbb{Z}$. A control state $\ell \in \mathbb{L}$ denotes a program point and $\mathbb{Z}$ denotes the set of integers. The initial states are $d_\iota = \Sigma_\iota = \{(\ell_0, 0)\}$, which means that the program point is ℓ_0 and the variable x has the initial value 0. We use the trivial transition relation corresponding to each program instruction. The question mark denotes a random input of integers; it represents any integer. Thus, the first iteration becomes $F(d_\iota) = \{(\ell_0, 0)\} \cup \{(\ell_1, n) \mid n \in \mathbb{Z}\}$. After the third iteration $F^3(d_\iota)$, it converges to the following:

$$[\![P]\!] = \{(\ell_0, 0)\} \cup \{(\ell_1, n) \mid n \in \mathbb{Z}\} \cup \{(\ell_2, n) \mid n \geq 0\} \cup \{(\ell_3, n) \mid n < 0\} \cup \{(\ell_4, n) \mid n \geq 0\}.$$

The randomness of the question mark generates an infinite number of reachable states. In general, the lengths of iterations could be also infinite when programs have infinite loops. Thus, it is difficult to acquire finite representation of collecting semantics of complex programs.

2.2 Abstract Interpretation

Abstract interpretation [13] presents a way to over-approximate the collecting semantics $[\![P]\!]$ to an *abstract semantics* $[\![P]\!]^\sharp$ using an *abstract domain* $\mathbb{D}^\sharp$ and an abstract transfer function $F^\sharp$:

$$[\![P]\!]^\sharp = \lim_{n\to\infty} (F^\sharp)^n(d_\iota^\sharp).$$

We define a *state abstraction* using $\mathbb{D}^\sharp$ and a *concretization function* $\gamma : \mathbb{D}^\sharp \xrightarrow{\gamma} \mathbb{D}$. The abstract domain $\mathbb{D}^\sharp$ is a lattice whose elements are abstract states $d^\sharp \in \mathbb{D}^\sharp$ with a partial order $\sqsubseteq$ between abstract states. The join $\sqcup$ and the meet $\sqcap$ operators denote the least upper bound and the greatest lower bound, respectively. Each abstract state represents a set of states using $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$. The initial abstract state $d_\iota^\sharp \in \mathbb{D}^\sharp$ represents an abstraction of the initial state set; $d_\iota \subseteq \gamma(d_\iota^\sharp)$.

We define an abstract transfer function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ as $F^\sharp(d^\sharp) = d^\sharp \sqcup \mathbf{step}^\sharp(d^\sharp)$ with an *abstract one-step execution* $\mathbf{step}^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$. For a sound abstraction, the following conditions should hold:

- a sound join $\sqcup$: $\forall d_0^\sharp, d_1^\sharp \in \mathbb{D}^\sharp.\ \gamma(d_0^\sharp) \cup \gamma(d_1^\sharp) \subseteq \gamma(d_0^\sharp \sqcup d_1^\sharp)$,
- a sound abstract one-step execution $\mathbf{step}^\sharp$: $\forall d^\sharp \in \mathbb{D}^\sharp.\ \mathbf{step} \circ \gamma(d^\sharp) \subseteq \gamma \circ \mathbf{step}^\sharp(d^\sharp)$,

where $f \circ g$ denotes the composition of functions f and g. Then, for any step $n \in \mathbb{N}_0$, the proposition $F^n(d_\iota) \subseteq \gamma \circ (F^\sharp)^n(d_\iota^\sharp)$ holds, thus $[\![P]\!] \subseteq \gamma([\![P]\!]^\sharp)$.

Consider the state abstraction with abstract states $\mathbb{D}^\sharp = \{\top, \bot, \oplus, -\}$ and the concretization:

$$\gamma(\top) = \Sigma \qquad \gamma(\oplus) = \mathbb{L} \times \{n \in \mathbb{Z} \mid n \geq 0\} \qquad \gamma(-) = \mathbb{L} \times \{n \in \mathbb{Z} \mid n < 0\} \qquad \gamma(\bot) = \varnothing.$$

For an abstract transfer function $F^\sharp$, assume that it soundly abstracts a transfer function F in the most precise way. Under this abstract interpretation, the initial abstract state $d_\iota^\sharp$ is $\oplus$ because the initial value of the variable x is 0. The concretization $\gamma(\oplus)$ contains the state $(\ell_0, 0)$ whose next reachable states are $\{(\ell_1, n) \mid n \in \mathbb{Z}\}$. Thus, $\gamma(\mathbf{step}^\sharp(\oplus))$ should contain $\{(\ell_1, n) \mid n \in \mathbb{Z}\}$ and $\mathbf{step}^\sharp(\oplus)$ must be the top abstract state $\top$. Therefore, the first iteration becomes $F^\sharp(d_\iota^\sharp) = d_\iota^\sharp \sqcup \mathbf{step}^\sharp(d_\iota^\sharp) = \oplus \sqcup \top = \top$, which makes $[\![P]\!]^\sharp$ converge to the top abstract state $\top$. However, this analysis result contains many false positives such as $(\ell_3, 42)$. One option is to use analysis sensitivities to increase the precision.

2.3 Abstract Interpretation with Analysis Sensitivity

Abstract interpretation often uses *analysis sensitivity* techniques to achieve precise analysis results. We define a sensitive abstract domain $\mathbb{D}_\delta^\sharp$ using a *view abstraction* [42] δ and describe an abstract transfer function $F^\sharp$ with the analysis sensitivity. A view abstraction $\Pi \xrightarrow{\delta} \mathbb{D}$ provides multiple points of views for reachable states during static analysis. It maps a finite number of *views* Π to sets of states $\mathbb{D}$. Each view $\pi \in \Pi$ represents a set of states $\delta(\pi)$. A *sensitive state abstraction* $\mathbb{D}_\delta^\sharp \xrightarrow{\gamma_\delta} \mathbb{D}$ is defined with a given view abstraction δ. Its abstract domain $\mathbb{D}_\delta^\sharp = \Pi \to \mathbb{D}^\sharp$ maps views Π to a view-wise abstract domain $\mathbb{D}^\sharp$, and its operators $\sqsubseteq_\delta$, $\sqcup_\delta$, and $\sqcap_\delta$ are defined in a pointwise manner for each view $\pi \in \Pi$. The concretization function $\gamma_\delta : \mathbb{D}_\delta^\sharp \to \mathbb{D}$ is as follows:

$$\gamma_\delta(d_\delta^\sharp) = \{\sigma \in \Sigma \mid \forall \pi \in \Pi.\ \sigma \in \delta(\pi) \Rightarrow \sigma \in \gamma \circ d_\delta^\sharp(\pi)\}.$$

With analysis sensitivities, we define an abstract one-step execution $\mathbf{step}_\delta^\sharp : \mathbb{D}_\delta^\sharp \to \mathbb{D}_\delta^\sharp$ as follows:

$$\mathbf{step}_\delta^\sharp(d_\delta^\sharp) = \lambda \pi \in \Pi.\ \bigsqcup_{\pi' \in \Pi} [\![\pi' \to \pi]\!]^\sharp \circ d_\delta^\sharp(\pi'),$$

where $[\![\pi' \to \pi]\!]^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is the abstract semantics of a *view transition* from a view π' to another view π. It should satisfy the following condition for the soundness of the analysis:

$$\forall d^\sharp \in \mathbb{D}^\sharp.\ \mathbf{step}(\gamma(d^\sharp) \cap \delta(\pi')) \cap \delta(\pi) \subseteq \gamma \circ [\![\pi' \to \pi]\!]^\sharp(d^\sharp).$$

For example, one of the most popular sensitivity techniques is *flow-sensitivity*. We define it with a flow-sensitive view abstraction $\delta^{FS} : \mathbb{L} \to \mathbb{D}$. It discriminates states using their control states:

$$\forall \ell \in \mathbb{L}.\ \delta^{FS}(\ell) = \{\sigma \in \Sigma \mid \sigma = (\ell, \_)\}.$$

It partitions reachable states of the program in Figure 1 into five cases based on their control states: $\ell_0, \ldots, \ell_4$. We define the abstract semantics of each view transition in the most precise way. For instance, we define the abstract semantics of a view transition from ℓ_1 to ℓ_2 as follows:

$$\forall d^\sharp \in \mathbb{D}^\sharp.\ [\![\ell_1 \to \ell_2]\!]^\sharp(d^\sharp) = \begin{cases} \oplus & \text{if } d^\sharp \in \{\top, \oplus\}, \\ \bot & \text{if } d^\sharp \in \{\bot, -\}. \end{cases}$$

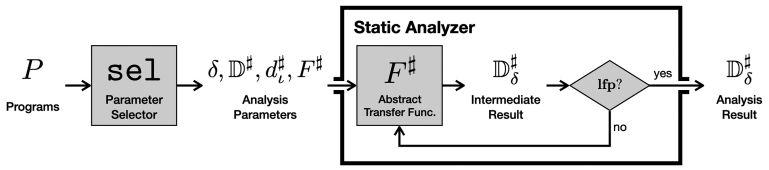


Fig. 2. Structure of parametric static analysis.

Since the view transition $\ell_1 \to \ell_2$ passes the condition $x \geq 0$, it refines $\top$ to the non-negative integer abstract state $\oplus$ and the negative integer abstract state $-$ to $\bot$ in $[\![\ell_1 \to \ell_2]\!]^\sharp$. With such abstract semantics of view transitions, the final analysis result of the program becomes:

$$[\![P]\!]^\sharp(\ell_0) = \oplus \qquad [\![P]\!]^\sharp(\ell_1) = \top \qquad [\![P]\!]^\sharp(\ell_2) = \oplus \qquad [\![P]\!]^\sharp(\ell_3) = - \qquad [\![P]\!]^\sharp(\ell_4) = \oplus.$$

It produces a more precise analysis result than without using the sensitivity.

2.4 Parametric Static Analysis

Static analysis techniques are often evaluated by three metrics: performance, precision, and soundness. Performance denotes how fast static analysis can analyze programs. Precision denotes how precisely static analysis can approximate program behaviors; if the analysis precision is low, then analysis results may contain many infeasible states resulting in many false alarms. In most cases, precision competes with performance. The more precise analysis performs, the more information it manipulates, which degrades the performance. For soundness, the abstract semantics computed by a static analysis should include the concrete semantics: $[\![P]\!] \subseteq \gamma([\![P]\!]^\sharp)$. However, it may not be because of the difficulties in safe approximation or real-world features such as dynamic code generation in JavaScript and reflection in Java.

Parametric static analysis considers a static analyzer as a black-box and focuses on how to improve analysis results by configuring its parameters. We dub such parameters that can be configured without knowing the details of static analysis *analysis parameters*. As illustrated in Figure 2, a parametric static analysis consists of two modules: a *parameter selector* and a *static analyzer*.

The parameter selector $\mathbf{sel} : \mathbb{P} \to \delta \times \mathbb{D}^\sharp \times d_i^\sharp \times F^\sharp$ selects analysis parameters. It utilizes the extracted program properties or feedback from previous or intermediate analysis results in the parameter selection process. We explain the detail of the parameter selector in Section 4.

The static analyzer produces an abstract semantics of the given program P via a static analysis based on the abstract interpretation framework. The abstract semantics depends on four analysis parameters selected via the parameter selector **sel**: a view abstraction δ, a state abstract domain $\mathbb{D}^\sharp$, an initial abstract state $d_i^\sharp$, and an abstract transfer function $F^\sharp$:

$$[\![P]\!]^\sharp = \lim_{n\to\infty} (F^\sharp)^n(d_i^\sharp) \text{ where } \begin{cases} d_i^\sharp \in \mathbb{D}_\delta^\sharp, \\ F^\sharp \in \mathbb{D}_\delta^\sharp \to \mathbb{D}_\delta^\sharp. \end{cases}$$

The initial abstract state $d_i^\sharp$ should be an element of the sensitive abstract domain $\mathbb{D}_\delta^\sharp$ and the abstract transfer function $F^\sharp$ should have the type $\mathbb{D}_\delta^\sharp \to \mathbb{D}_\delta^\sharp$.

3 ANALYSIS PARAMETERS

In parametric static analysis, we can specialize analyzers for a given program by selecting appropriate analysis parameters. In this section, we explain existing analysis techniques in terms of analysis parameters: analysis sensitivity using a view abstraction δ (Section 3.1), state abstraction

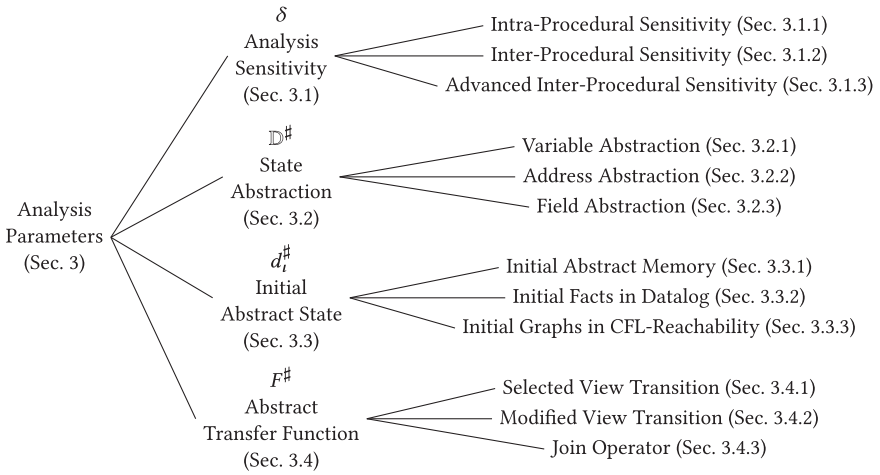


Fig. 3. Categories of analysis parameters utilized in parametric static analysis.

with an abstract domain $\mathbb{D}^\sharp$ (Section 3.2), initial abstract state $d_i^\sharp$ (Section 3.3), and abstract transfer function $F^\sharp$ (Section 3.4). We also discuss their effects on the quality of static analysis. Figure 3 depicts the overview of this section.

3.1 Analysis Sensitivity

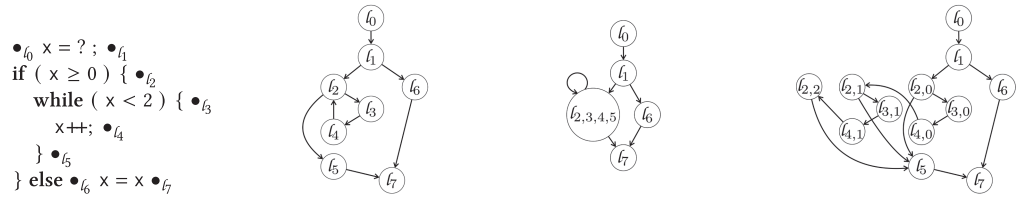
In static analysis, more sensitivities often provide opportunities to achieve more precise analysis and to verify stronger properties. As described in Section 2, a sensitivity is defined with a view abstraction $\delta : \Pi \to \mathbb{D}$, which divides states using a finite set of views Π. With a given view abstraction δ, a sensitive static analysis extends the original abstract domain $\mathbb{D}^\sharp$ to a sensitive abstract domain $\mathbb{D}^\sharp_\delta$, which maps views Π to the abstract domain $\mathbb{D}^\sharp$. Using views, it can enhance analysis precision by preventing merging different abstract states. However, more sensitive static analysis may lead to worse performance. To maintain more sensitivity, view abstractions become more fine-grained. The more fine-grained view abstractions, the more views static analysis should manage, which may degrade the analysis performance. To find a balance between precision and performance, researchers have sought for appropriate sensitivities for different program characteristics and analysis purposes. We explain *intra-procedural* (Section 3.1.1), *inter-procedural* (Section 3.1.2), and *advanced inter-procedural* (Section 3.1.3) sensitivities used in parametric static analysis.

3.1.1 Intra-procedural Sensitivity. One simple way to increase precision of intra-procedural analysis is to distinguish analysis results depending on control flows of a given program. For example, Figure 4(a) is a slight variant of the program in Figure 1 with a `while` loop. In the program, the variable `x` could be any integer, because the question mark denotes a random integer. However, depending on program points, we can get more information about the value of `x` than any integer. In the control point ℓ_2, because the condition $\mathtt{x} \geq 0$ holds, the value of `x` should be a non-negative integer. However, the value of `x` should be a negative integer in the control point ℓ_6.

To consider such flows, static analysis often uses *flow sensitivity*. We formally define it with a *flow sensitive view abstraction* $\delta^{FS} : \mathbb{L} \to \mathbb{D}$, which divides states based on their control states:

$$\forall \ell \in \mathbb{L}.\ \delta^{FS}(\ell) = \{\sigma \in \Sigma \mid \sigma = (\ell, \_)\}. \tag{1}$$

A flow-sensitive abstract domain $\mathbb{L} \to \mathbb{D}^\sharp$ is a map from control states $\mathbb{L}$ to abstract values of the basic abstract domain $\mathbb{D}^\sharp$. The example code in Figure 4(a) has eight control states: $\mathbb{L} = \{\ell_0, \ldots, \ell_7\}$.



(a) An example code (b) Flow-sensitivity (c) Partial flow-sensitivity (d) Loop sensitivity

Fig. 4. View abstractions for intra-procedural sensitivity.

Figure 4(b) illustrates the control states and control flows between them; a node denotes a control state and a directed edge denotes a control flow from a control state to another one. With flow-sensitivity, static analysis can distinguish different abstract values of x for different control states. Thus, we can verify that the value of x is non-negative in ℓ_2 and negative in ℓ_6, respectively.

One way to configure flow-sensitivity in intra-procedural analysis is a *partial flow-sensitivity*, which partially applies flow-sensitivity for specific control states. A partition of control states $\mathbb{L}/_{\equiv}$ represents which control states require flow- sensitivity. Thus, we define a *partial flow-sensitive view abstraction* $\delta^{FS[\mathbb{L}/_{\equiv}]} : \mathbb{L}/_{\equiv} \to \mathbb{D}$ as follows:

$$\forall X \in \mathbb{L}/_{\equiv}.\ \delta^{FS[\mathbb{L}/_{\equiv}]}(X) = \{\sigma \in \Sigma \mid \sigma = (\ell, \_) \wedge \ell \in X\}. \tag{2}$$

For example, if we do not want to precisely analyze the true branch in the previous example, then we could merge the control states inside the true branch but distinguish all the other control states using the partition: $\{\{\ell_0\}, \{\ell_1\}, \{\ell_6\}, \{\ell_7\}, \{\ell_2, \ell_3, \ell_4, \ell_5\}\}$. Figure 4(c) depicts partitioned control states and their control flows. Using partial flow-sensitivity, analysis becomes faster than using full flow-sensitivity with the cost of the precision loss in the true branch. Since the numbers of control states could be huge for large-scale programs, researchers have used partial flow-sensitivity. Wei and Ryder [111] apply flow-sensitivity only for heap-update statements in JavaScript, because they are critical parts that degrade the analysis precision in JavaScript points-to analysis.

Another way to achieve more precision while sacrificing performance is to use a more fine-grained sensitivity for loops. Park and Ryu [69] formally defined *loop sensitivity* to configure analysis sensitivity for loops without unrolling them. We extend their formalization with a finite set of loop context representations $\mathcal{L}$ and define a *loop sensitive view abstraction* $\delta^{LS} : \mathbb{L} \times \mathcal{L} \to \mathbb{D}$:

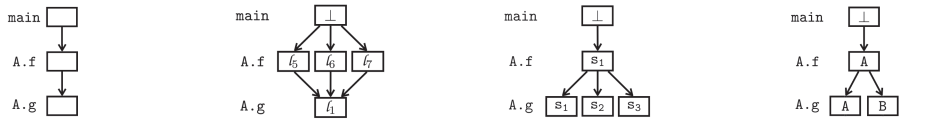
$$\forall \ell \in \mathbb{L}.\ \forall l \in \mathcal{L}.\ \delta^{LS}((\ell, l)) = \{\sigma \in \Sigma \mid \sigma = (\ell, \_) \wedge l = \texttt{getLoopCtxt}(\sigma)\}, \tag{3}$$

where loop context representations $\mathcal{L}$ are any features of loops and $\texttt{getLoopCtxt} : \Sigma \to \mathcal{L}$ returns a loop context representation for a given state. For example, Figure 4(d) depicts control flows of a loop-sensitive analysis where loop context representations are loop iteration numbers. In the graph, each node $\ell_{i,j}$ denotes the control point ℓ_i in the jth iteration of the `while` loop and ℓ_i denotes a control point without any loop contexts. Since `for-in` loops are JavaScript-specific features that degrade analysis precision, researchers have parameterized loop sensitivity to partially apply it to selected loop contexts. Sridharan et al. [90] identify correlated dynamic property accesses and apply iteration-based loop sensitivity only for loops that affect the correlations. Ko et al. [45, 46] syntactically define field-copy or transformation patterns and they apply loop sensitivity using field existence and variable values as representations only for loops that have such patterns.

3.1.2 Inter-procedural Sensitivity. Inter-procedural sensitivities are also called *context sensitivities,* because they distinguish analysis results depending on calling contexts of functions. Since a function has one definition site but multiple call sites, analysis often merges abstract values of

```
1. class A {                          7. main() {
2.   void f(A x) { •ℓ1 x.g(x); •ℓ2}   8. •ℓ4 A a = new A(); // s1
3.   void g(A y) { •ℓ3 }              9. •ℓ5 a.f(a);
4. };                                 10. •ℓ6 a.f(new A()); // s2
5.                                    11. •ℓ7 a.f(new B()); // s3
6. class B extends A;                 12. •ℓ8 }
```

(a) An example code



(b) context insensitivity (c) call-site sensitivity (d) object sensitivity (e) type sensitivity

Fig. 5. View abstractions for context sensitivity.

given arguments or calling contexts from different call sites, which makes function calls a root cause of analysis imprecision. For example, the code in Figure 5(a) has three calls of `A.f` on lines 9, 10, and 11, and the parameter `x` on line 2 receives three different objects. However, when using only flow-sensitivity as in Figure 5(b), the entry point of `A.f` has a unique view leading to precision loss.

To resolve such precision loss, static analysis generally uses context sensitivity. We define context sensitivity by extending states into tuples of control states, calling contexts, and memories (will be described in Section 3.2): $\Sigma = \mathbb{L} \times C \times \mathbb{M}$ where $C = \Sigma \uplus \{\bot\}$. A calling context $c \in C$ is a state right before calling the current function, or $\bot$ when the current state does not have any calling contexts. For example, the following table describes state transitions of the program in Figure 5(a):

Σ	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8	σ_9	σ_{10}	σ_{11}	σ_{12}	σ_{13}
$\mathbb{L}$	ℓ_4	ℓ_5	ℓ_1	ℓ_3	ℓ_2	ℓ_6	ℓ_1	ℓ_3	ℓ_2	ℓ_7	ℓ_1	ℓ_3	ℓ_2	ℓ_8
C	$\bot$	$\bot$	σ_1	σ_2	σ_1	$\bot$	σ_5	σ_6	σ_5	$\bot$	σ_9	σ_{10}	σ_9	$\bot$
$\mathbb{M}$	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}	m_{12}	m_{13}

The initial state σ_0 has $\bot$ as its calling context, because there is no calling context in the beginning. For the state transition $\sigma_1 \rightsquigarrow \sigma_2$, the calling context of σ_2 is σ_1, since the transition represents the function call on line 9. The state σ_4 also has the exactly same calling context σ_1. Because the number of calling contexts could be infinite, the *context sensitive view abstraction* $\delta^{CS} : R \to \mathbb{D}$ utilizes a finite number of context representations R as views instead of calling contexts C:

$$\forall r \in R.\ \delta^{CS}(r) = \{\sigma \in \Sigma \mid \sigma = (\_, c, \_) \wedge r = \rho(c)\}, \tag{4}$$

where a *representation extractor* $\rho : C \to R$ returns the representation of each calling context $c \in C$.

Moreover, *selective context sensitivity* allows to selectively apply context sensitivity for specific calling contexts. For context selection, it restricts the representation extractor ρ as $\rho\,|_{C_{sel}}$ with a selected set of contexts C_{sel}. If a calling context c is not in C_{sel}, then it is not selected and represented as the context insensitive view $\bot$. In general, calling contexts are selected based on callee functions:

$$C_{sel} = \{c \in C \mid \mathsf{Callee}(c) \in \mathcal{F}_{sel}\}, \tag{5}$$

where Callee : $\mathcal{C} \to \mathcal{F}$ takes a calling context and returns its callee function. In parametric static analysis, selective context sensitivity is commonly used for three basic context sensitivities: *call-site sensitivity*, *object sensitivity*, and *type sensitivity* with corresponding context representations.

- **Call-site Sensitivity:** One of the most widely used context sensitivities is *call-site sensitivity* [81] that represents calling contexts as their call-sites:

  $$\forall c \in \mathcal{C}.\ \rho(c) = \ell \text{ where } c = (\ell, \_, \_).$$

  For example, Figure 5(c) shows call-site sensitivity of the program in Figure 5(a). The `main` function does not have any calling context $\bot$, `A.f` has three call-sites ℓ_5, ℓ_6, and ℓ_7, and `A.g` has one call-site ℓ_1. The call-site sensitivity distinguishes three call sites of `A.f` and the variable `x` has a single object in each view precisely. However, the variable `y` still has an imprecise abstract value, because three different calling contexts of `A.g` are merged to one single call-site ℓ_1. Researchers [25, 26, 41, 67, 85] selectively apply the call-site sensitivity for specific callee functions to balance between precision and performance.
- **Object Sensitivity:** Another approach to represent calling contexts is *object sensitivity* [58], which uses abstract addresses of receiver objects as calling contexts:

  $$\forall c \in \mathcal{C}.\ \rho(c) = a^\sharp \text{ where } a^\sharp \in \mathbb{A}^\sharp \text{ is the abstract address of the receiver object in } c.$$

  Thus, it depends on abstraction of the addresses of receiver objects, and we explain abstract addresses $\mathbb{A}^\sharp$ in Section 3.2.2. In object sensitivity, the *allocation-site abstraction* (will be explained in Section 3.2.2 in detail), which divides addresses using their allocation sites, is the most used one. For example, the program in Figure 5(a) generates three objects and they all have different allocation sites: $\mathtt{s}_1$, $\mathtt{s}_2$, and $\mathtt{s}_3$. Because receiver objects of `A.f` at three call sites are all $\mathtt{s}_1$, their calling contexts are all merged to the representation $\mathtt{s}_1$, as shown in Figure 5(d). Thus, the variable `x` has an imprecise abstract value. However, since three function calls of `A.g` has three different receiver objects created at $\mathtt{s}_1$, $\mathtt{s}_2$, and $\mathtt{s}_3$, the variable `y` of each view has a precise object in each context view. Smaragdakis et al. [41, 50, 85] selectively apply object sensitivity for Java program analysis.
- **Type Sensitivity:** The *type sensitivity* is a variant of the object sensitivity. It uses types of receiver objects instead of their allocations sites:

  $$\forall c \in \mathcal{C}.\ \rho(c) = \tau \text{ where } \tau \text{ is the type of the receiver object in } c.$$

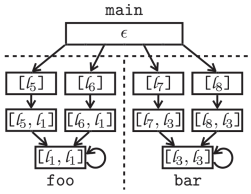
  Compared to object sensitivity, type sensitivity reduces the number of context representations without a huge loss of precision for type information. In Figure 5(e), the type sensitivity divides states into four cases: $\bot$ for `main`, `A` for `A.f`, and `A` and `B` for `A.g`. Thus, objects created at $\mathtt{s}_1$ and $\mathtt{s}_2$ are merged to a single view `A`, and the object created at $\mathtt{s}_3$ is represented as `B` in the type sensitivity. Smaragdakis et al. [41, 85] also selectively apply type sensitivity for Java analysis.

Beyond receiver objects, several researchers utilize arguments as context representations. For JavaScript, Wei et al. [113] selectively apply 1st-argument sensitivity [112], which distinguishes calling contexts using the abstract values of first arguments instead of receiver objects. For Java, Thakur and Nandivada [103, 104] propose a **level-summarized relevant value-contexts (LSRV-contexts)** to which reduces the size of value-contexts by using pre-analysis.

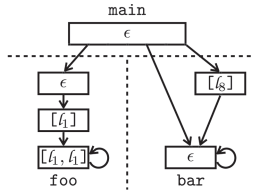
Furthermore, researchers proposed to use different context sensitivities for different callee functions. For example, consider static analysis of the example in Figure 5(a) that uses object sensitivity for `A.f` and type sensitivity for `A.g`. Then, the calling contexts of `A.f` are merged into a single object sensitive view $\mathtt{s}_1$ but calling contexts of `A.g` are divided into two different type sensitive views

```
1. void foo(int x) { if (?) •ℓ1 foo(x); •ℓ2 }
2. void bar(int y) { if (?) •ℓ3 bar(y); •ℓ4 }
3. int main() {
4. •ℓ5 foo(1); •ℓ6 foo(2);
5. •ℓ7 bar(3); •ℓ8 bar(4);
6. •ℓ9 }
```

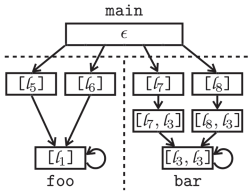
(a) An example code



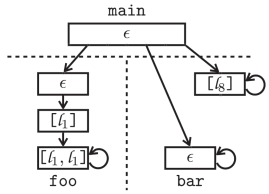
(b) 2-call-site sensitivity



(c) Selective 2-call-site sensitivity for $(\ell_1, \texttt{foo})$ and $(\ell_8, \texttt{bar})$



(d) k-call-site sensitivity with $k = 1$ for `foo` and $k = 2$ for `bar`



(e) Selective 2-call-site sensitivity for $(\ell_1, \texttt{foo})$ and $(\ell_8, \texttt{bar})$ with tunneling

Fig. 6. View abstractions for advanced context sensitivity.

A and B. Kastrinis and Smaragdakis [41] apply call-site sensitivity for Java static method calls but object/type sensitivity for the others. Wei and Ryder [112] select one of call-site, object, and 1st-parameter sensitivity for each function using ML-based heuristics for function characteristics in JavaScript static analysis. Li et al. [51] select object or type sensitivity for each function by predicting its analysis time with object allocation graphs for Java static analysis.

3.1.3 Advanced Inter-procedural Sensitivity. Several approaches utilize *k-context sensitivity* for parametric static analysis. Instead of abstracting the last call context, it abstracts k the recent calling contexts as a sequence of context representations. For the k-context sensitivity, the context sensitive view abstraction in Equation (4) is extended as follows:

$$\forall \overline{r} \in R^{\leq k}.\ \delta^{CS[k]}(\overline{r}) = \left\{\sigma \in \Sigma \mid \sigma = (\_, c, \_) \wedge \rho^k(c) = \overline{r}\right\}, \tag{6}$$

where the k-representation extractor $\rho^k : C \to R^{\leq k}$ is defined as follows:

$$\forall c \in C.\ \rho^k(c) = \begin{cases} \epsilon & \text{if } \rho(c) = \bot \vee k = 0, \\ \rho^{k-1}(c) :+ \rho(c) & \text{otherwise,} \end{cases} \tag{7}$$

where $a :+ b$ denotes a appended with b. Consider 2-call-site sensitivity, which is k-context sensitivity for call-site sensitivity with $k = 2$, for the code in Figure 6(a). For the function `foo`, it distinguishes calling contexts of `foo` in `main` using the call-sites ℓ_5 and ℓ_6 as in the basic call-site sensitivity. However, while the basic call-site sensitivity distinguishes calling contexts of `foo` in itself using the call-site ℓ_1, 2-call-site sensitivity divides them into three representations using two the recent call-sites: $[\ell_5, \ell_1]$, $[\ell_6, \ell_1]$, and $[\ell_1, \ell_1]$, as shown in Figure 6(b).

Moreover, it is possible to selectively apply context sensitivity for specific calling contexts by modifying the k-representation extractor ρ^k to $\rho^k_{C_{sel}}$ with a selective set of contexts C_{sel}:

$$\forall c \in C.\ \rho^k_{C_{sel}}(c) = \begin{cases} \epsilon & \text{if } c = \bot \vee k = 0 \vee c \notin C_{sel}, \\ \rho^{k-1}_{C_{sel}}(c) :+ \rho(c) & \text{otherwise.} \end{cases} \tag{8}$$

If a calling context c is not in C_{sel}, then it is not selected and produces just the empty sequence of context representation ϵ. For parametric static analysis, researchers have extended k-context sensitivity in three ways: *call-edge selection*, *different k per function*, and *context tunneling*.

- **Call-edge Selection:** Instead of selecting callee functions $\mathcal{F}_{sel}$ for selected calling contexts C_{sel} as described in Equation (5), we can select *call-edges* $\mathcal{E} = \mathcal{P}(R \times \mathcal{F})$ from context representations for callers to callee functions. This approach provides more fine-grained choices for selection of calling contexts via both callee functions and calling context representations of callers. It defines the selected calling contexts C_{sel} using *selected call-edges* $\mathcal{E}_{sel} \subseteq \mathcal{E}$ as follows:

  $$C_{sel} = \{c \in C \mid (\rho(c), \mathsf{Callee}(c)) \in \mathcal{E}_{sel}\}.$$

  For example, if we define $\mathcal{E}_{sel} = \{(\ell_1, \texttt{foo}), (\ell_8, \texttt{bar})\}$, then it distinguishes only calling contexts whose call-site and callee function are $(\ell_1, \texttt{foo})$ or $(\ell_8, \texttt{bar})$, as shown in Figure 6(c). Thus, calling contexts of `foo` at ℓ_5 and ℓ_6 are merged as ϵ and also calling contexts of `bar` at ℓ_1 and ℓ_7 are merged as ϵ. Researchers [23, 38, 54, 65, 66, 75, 114, 119] have adjusted the analysis precision and performance using selected call-edges $\mathcal{E}_{sel}$. While most works apply the technique for call-site sensitivity, several works [38, 54, 75] apply it for object or type sensitivities. In addition, Whaley and Lam [114] utilize **Binary Decision Diagrams (BDDs)** to indirectly merge calling contexts with same abstract states, which enhances the analysis performance without precision degradation.
- **Different k per Function:** Another way to parameterize k-context sensitivity is to assign different k for each function. It modifies Equation (8) with a *depth map* $K : \mathcal{F} \to \mathbb{N}$ from functions to their depths for k as follows:

  $$\forall c \in C.\ \rho^k_{C_{sel}}(c) = \begin{cases} \epsilon & \text{if } c = \bot \vee k' = 0 \vee c \notin C_{sel}, \\ \rho^{k'-1}_{C_{sel}}(c) :+ \rho(c) & \text{otherwise,} \end{cases}$$

  where $k' = \min(k, K \circ \mathsf{Callee}(c))$. For example, let us apply 1-callsite-sensitivity for `foo` and 2-callsite-sensitivity for `bar` to the code in Figure 6(a). Then, all calling contexts of `foo` at ℓ_1 is merged into a single view $[\ell_1]$ as shown in Figure 6(d) instead of three different views $[\ell_1, \ell_1]$, $[\ell_5, \ell_1]$, and $[\ell_6, \ell_1]$ in Figure 6(b). For Java static analysis, Jeong et al. [39] assign different k for each method using machine-learning algorithms. Moreover, Rama et al. [75] configure k for different context representations instead of functions using backward analysis.
- **Context Tunneling:** Another approach to selectively apply k-context sensitivity is *context tunneling*, which conveys the current context to unselected calling contexts:

  $$\forall c \in C.\ \rho^k_{C_{sel}}(c) = \begin{cases} \epsilon & \text{if } c = \bot \vee k = 0, \\ \rho^{k-1}_{C_{sel}}(c) & \text{if } c \notin C_{sel}, \\ \rho^{k-1}_{C_{sel}}(c) :+ \rho(c) & \text{otherwise.} \end{cases}$$

  For example, let us apply context tunneling to the sensitivity shown in Figure 6(c). Without context tunneling, the calling context of `bar` at ℓ_1 through ℓ_8 should be ϵ because $(\ell_1, \texttt{bar})$ is not in $\mathcal{E}_{sel}$. However, with context tunneling, it is represented as $[\ell_8]$ as shown in Figure 6(e), because the second most call-site ℓ_8 is conveyed. Tan et al. [101] first introduce a way to remove redundant contexts via **object allocation graph (OAG)** to improve the precision of k-object

```
1. class Obj { Int f; }
2. x = new Obj();
3. y = new Obj();
4. x.f = 42;
```

(a) An example code

$\mathbb{X}$	$\mathbb{V}$
x	a_1
y	a_2

(b) Environment after execution

$\mathbb{A}$	$\mathbb{O}$
a_1	f $\mapsto$ 42
a_2	f $\mapsto$ 0

(c) Heap after execution

Fig. 7. Concrete memory representation.

sensitivity for Java. Jeon et al. [38] formally define context tunneling and select target functions for context tunneling using machine learning algorithm.

3.2 State Abstraction

The second analysis parameter is a state abstraction with an abstract domain $\mathbb{D}^\sharp$. A state abstraction describes how to abstract a set of states. A program is a state transition system and each state represents the program's properties. Because the number of reachable states of a program could be infinite, computing the exact set of reachable states may be infeasible. To alleviate this problem, static analysis over-approximates them to an abstract state $d^\sharp \in \mathbb{D}^\sharp$. The abstract state domain $(\mathbb{D}^\sharp, \sqsubseteq)$ is a lattice and the concretization function $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$ defines the meaning of abstract states.

While state abstractions work for various data types such as numeric values, strings, and complex data structures, existing parametric static analysis has focused on configuring *memory abstractions*. A memory $m \in \mathbb{M}$ consists of an environment and a heap, $\mathbb{M} = \mathbb{E} \times \mathbb{H}$, where an environment $e \in \mathbb{E}$ is a finite mapping from variables to values, a heap $h \in \mathbb{H}$ is a finite mapping from addresses to objects, and each object $o \in \mathbb{O}$ is a finite mapping from fields to values:

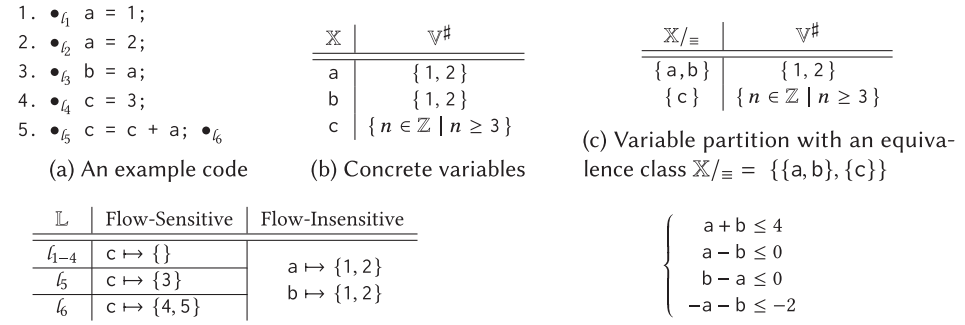
$$e \in \mathbb{E} = \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} \qquad h \in \mathbb{H} = \mathbb{A} \xrightarrow{\text{fin}} \mathbb{O} \qquad o \in \mathbb{O} = \mathbb{F} \xrightarrow{\text{fin}} \mathbb{V}.$$

For example, Figure 7 shows an example code and its environment and heap at the end of the example. At lines 2 and 3, the constructor calls of the class `Obj` at line 1 generate concrete objects with their corresponding addresses a_1 and a_2, respectively. As described in Figure 7(b), the environment has variables `x` that points to the concrete address a_1, and `y` with a_2. The assignment statement at line 4 updates the field `f` of the variable `x` to the integer value 42. Figure 7(c) depicts the heap structure at the end of the program; the addresses a_1 and a_2 point to objects that have the field `f` with the value 42 and 0, respectively.

Among various memory abstraction techniques, we study memory abstractions that parametric static analysis have used and categorize them into three kinds based on their abstraction targets: *variables* (Section 3.2.1), *addresses* (Section 3.2.2), and *fields* (Section 3.2.3).

3.2.1 Variable Abstraction. Environments $\mathbb{E} = \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}$ are finite maps from variables to their values. In programs, a variable denotes a memory location that points to its value. The variable assignment `x` = v updates the memory location of `x` to point to the new value $v \in \mathbb{V}$.

To abstract such environments, static analysis defines an abstract environment domain $\mathbb{E}^\sharp$. One of basic abstract environment domains is $\mathbb{E}^\sharp = \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}^\sharp$, which is a finite map from variables to their abstract values. Let us assume that an abstract value in $\mathbb{V}^\sharp$ is a set of concrete values. For example, Figure 8(a) shows a simple example code and Figure 8(b) presents its analysis result with the basic environment abstraction and flow-insensitivity. The abstract value of the variable `a` is $\{1, 2\}$ and the variable `b` has the same abstract value with `a`. The abstract value of the variable `c` initially is $\{3\}$, and it gets increased by adding the abstract value of the variable `a`. Because the variable `b` points to $\{1, 2\}$ and the analysis is flow-insensitive, `c` points to $\{n \in \mathbb{Z} \mid n \geq 3\}$.



(a) An example code

(b) Concrete variables

(c) Variable partition with an equivalence class $\mathbb{X}/_{\equiv} = \{\{a, b\}, \{c\}\}$

(d) Variable selection $X = \{c\}$ for flow-sensitive view abstraction

(e) Variable relation selection $\mathcal{R}_{\mathbb{X}\times\mathbb{X}} = \{(a, b)\}$ for the octagon domain

Fig. 8. Variable abstractions.

Parametric static analysis has used three kinds of parametric variable abstraction:

- **Variable Partition:** While the basic environment abstraction uses variables themselves as keys of an abstract environment, several researchers [27, 78] use a partition of variables $\mathbb{X}/_{\equiv}$ instead of concrete variables $\mathbb{X}$:

$$\mathbb{E}^\sharp = \mathbb{X}/_{\equiv} \xrightarrow{\text{fin}} \mathbb{V}^\sharp,$$

  where $\mathbb{X}/_{\equiv}$ is a partition of $\mathbb{X}$ based on the equivalence relation $\equiv$. It is often called ***off-line variable substitution (OVS)*** [78], because it substitutes equivalent variables to an equivalence class representation before analysis. For example, Figure 8(c) shows an abstract environment based on flow-insensitive analysis with a variable partition $\mathbb{X}/_{\equiv} = \{\{a, b\}, \{c\}\}$, which divides the set of variables to two partitions for the example code in Figure 8(a). After applying this variable partition, we can get the same precision as the one without applying it while the number of keys of the abstract environment gets reduced. Therefore, since it reduces the number of abstract memory locations that static analysis manages, it can improve the performance of static analysis. However, note that inappropriate variable partitions may decrease the analysis precision. Assume that a variable partition divides variables to $\mathbb{X}/_{\equiv} = \{\{a\}, \{b, c\}\}$. Then, the abstract value of $\{b, c\}$ becomes $\{n \in \mathbb{Z} \mid n \geq 1\}$ while the analysis with concrete variables computes the abstract value of b as $\{1, 2\}$. Thus, finding an appropriate equivalence relation for given programs is important for variable partition. Hardekopf and Lin [27] extend OVS using ***Hash-based Value Numbering (HVN)*** with dereference, union, and location equivalence.
- **Variable Selection:** To balance the analysis performance and precision, an analysis can apply specific analysis techniques to only target variables $X \subseteq \mathbb{X}$. The most representative variable selection is selective flow-sensitivity [12, 25, 34, 56, 67]. For the example code in Figure 8(a), Figure 8(d) shows the selective flow-sensitivity for a set of selected variables $X = \{c\}$. Because it analyzes variables a and b in a flow-insensitive way, they have the same abstract values as in Figure 8(b). With a flow-sensitive analysis, the variable c does not exist from ℓ_1 to ℓ_4 and it is defined at ℓ_5 with the initial value 3. At the program point ℓ_6, the abstract value of c gets increased by adding the abstract value of the variable a, and c points to $\{4, 5\}$. Besides the selective flow-sensitivity, restricting must-alias sets of abstract states during type-state analysis [120] is also variable selection. To select a set of flow-sensitive variables, Guyer and Lin [25] use fast and low-precision pointer analysis as pre-analysis. Lu and Xue [56] select context-sensitive variables by reasoning about **context-free-language (CFL)** reachability at the level of variables

in programs using a new CFL-reachability formulation of k-object sensitivity. Guyer and Lin [26] use backward analysis, and Oh et al. [12, 34, 67] use machine learning to find variables that require precise analysis results in pointer analysis for C.

- **Variable Relation Selection:** Another approach is to selectively apply state abstraction techniques to specific variable relations $\mathcal{R}_{\mathbb{X}\times\mathbb{X}} \subseteq \mathbb{X}\times\mathbb{X}$ such as octagon packing [59]. The *octagon domain* is a relational abstract domain for numerical values defined with conjunction of invariants of the form $\pm\mathtt{x} \pm \mathtt{y} \leq c$ where $\mathtt{x}$ and $\mathtt{y}$ are variables and c is a constant. Because using the octagon domain for all variable relations is costly, octagon packing uses it selectively only for target variable relations. For example, Figure 8(e) uses the octagon domain only for the relations between a and b. It increases the analysis precision for a and b, since it excludes impossible cases such as (a, b) = (1, 2) and (a, b) = (2, 1) without much overhead. Blanchet et al. [10] first introduce *packing* for the octagon domain, which restricts variable relations on the octagon domain only between variables in the same packs. Miné [59] and Oh et al. [64] syntactically pack variables based on program blocks, and Heo et al. [33] utilize machine learning to cluster variables. Beyond octagon packing, several researchers directly configure the set of variable relations in the octagon domain in a more fine-grained way. Oh et al. [65, 66] perform impact analysis, and Chae et al. [12] utilize machine learning for variable relation selection.

3.2.2 Address Abstractions. Heaps $\mathbb{H} = \mathbb{A} \xrightarrow{\text{fin}} \mathbb{O}$ are finite maps from addresses to objects. A heap stores objects at their corresponding addresses.

To abstract heaps, static analysis defines abstraction of addresses. The most widely used address abstraction is *allocation-site abstraction*. A program point that creates objects has its own unique allocation-site $\mathtt{s} \in \mathbb{S}$, and the allocation-site abstraction uses them as abstract addresses:

$$\mathbb{H}^\sharp = \mathbb{A}^\sharp \to \mathbb{O}^\sharp \qquad\qquad \mathbb{A}^\sharp = \mathbb{S}.$$

Thus, in the allocation-site abstraction, a single allocation-site $\mathtt{s} \in \mathbb{S}$ represents all concrete addresses allocated at s. For example, Figure 9(b) shows the allocation-site abstraction for the example code in Figure 9(a). The code has three allocation-sites: $\mathtt{s}_1$ at top level, $\mathtt{s}_2$ in function f, and $\mathtt{s}_3$ in function g. While a single object is allocated at $\mathtt{s}_1$, two different objects are allocated at both $\mathtt{s}_2$ and $\mathtt{s}_3$ because of two function calls of f and g, respectively. Thus, addresses pointed by variables b and c are merged to a single abstract address $\mathtt{s}_2$ and addresses pointed by d and e are merged to $\mathtt{s}_3$. We show the abstract value of the field k of each abstract address $a^\sharp$ in Figure 9(b).

Parametric static analysis has used two kinds of parametric address abstraction:

- **Partitioned Allocation-site Abstraction:** Similarly for variable partition in the variable abstraction (Section 3.2.1), researchers also use partitions of allocation-sites as abstract addresses:

$$\mathbb{A}^\sharp = \mathbb{S}/_{\equiv}.$$

  For example, Figure 9(c) shows the partitioned allocation-site abstraction with the equivalence class $\mathbb{S}/_{\equiv} = \{\{\mathtt{s}_1, \mathtt{s}_2\}, \{\mathtt{s}_3\}\}$. It merges $\mathtt{s}_1$ and $\mathtt{s}_2$ to a single abstract address. Guyer and Lin [25] use fast and low-precision pointer analysis as pre-analysis to merge allocation sites. Naik et al. [62] and Zhang et al. [120] merge allocation-sites to two different partitions L for "thread-local" addresses and E for "possibly thread-escaping" addresses in parameterized thread-escape analysis. Tan et al. [102] selectively partition allocation-sites using their types.
- **Selective Heap Cloning:** To improve the analysis precision, *heap cloning* refines allocation-sites using calling contexts:

$$\mathbb{A}^\sharp = \mathbb{S} \times R_\perp,$$

```
1. class A { int k; }
2. a = new A { k = 1 }; // s1
3. A f(int x) { return new A { k = x }; } // s2
4. A g(int y) { return new A { k = y }; } // s3
5. •ℓ1 b = f(2);
6. •ℓ2 c = f(3);
7. •ℓ3 d = g(4);
8. •ℓ4 e = g(5);
```

(a) An example code

$\mathbb{A}^\sharp$	$a^\sharp$.k
s_1	{ 1 }
s_2	{ 2,3 }
s_3	{ 4,5 }

(b) Allocation-site abstraction

$\mathbb{A}^\sharp$	$a^\sharp$.k
{ s_1, s_2 }	{ 1,2,3 }
{ s_3 }	{ 4,5 }

(c) Partitioned allocation-site abstraction with $\mathbb{S}/_{\equiv} = \{\{ s_1, s_2 \}, \{ s_3 \}\}$

$\mathbb{A}^\sharp$	$a^\sharp$.k
(s_1, $\bot$)	{ 1 }
(s_2, ℓ_1)	{ 2 }
(s_2, ℓ_2)	{ 3 }
(s_3, ℓ_3)	{ 4 }
(s_3, ℓ_4)	{ 5 }

(d) Heap cloning with call-site sensitivity

$\mathbb{A}^\sharp$	$a^\sharp$.k
s_1	{ 1 }
s_2	{ 2,3 }
(s_3, ℓ_3)	{ 4 }
(s_3, ℓ_4)	{ 5 }

(e) Selective heap cloning with $S = \{ s_3 \}$ with call-site sensitivity

Fig. 9. Address abstractions.

where R is a set of context representations and $R_\bot = R \cup \{\bot\}$. For example, Figure 9(d) shows heap cloning with call-site sensitivity. The abstract addresses $\mathbb{A}^\sharp = \mathbb{S} \times \mathbb{L}_\bot$ are pairs of allocation-sites and either call-sites or $\bot$. Since s_1 does not have any calling contexts, it is refined to (s_1, $\bot$). On the contrary, since s_2 has two call-sites ℓ_1 and ℓ_2, it gets split to two different abstract addresses (s_2, ℓ_1) and (s_2, ℓ_2) }. Similarly, s_3 gets split to (s_3, ℓ_3) and (s_3, ℓ_4). *Selective heap cloning* [56, 85, 94] selectively applies heap cloning to target allocation-sites S:

$$\mathbb{A}^\sharp = (\mathbb{S} \setminus S) \cup (S \times R_\bot).$$

For example, Figure 9(e) shows selective heap cloning with target allocation-sites $S = \{ s_3 \}$ with call-site sensitivity. The allocation-site s_3 is divided to (s_3, ℓ_3) and (s_3, ℓ_4), but the other allocation-sites s_1 and s_2 are not affected by the heap cloning technique.

Another parameter of selective heap cloning is k for each function when using k-context sensitivity described in Section 3.1.3. Hassanshahi et al. [29] perform backward analysis to configure k for each allocation site.

$$\mathbb{A}^\sharp = \mathbb{S} \times R^{\leq k}.$$

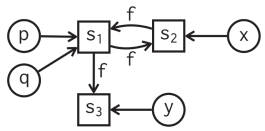
3.2.3 Field Abstraction. Static analysis with object abstraction often uses *field-sensitive abstraction*, which uses concrete fields as keys of abstract objects in memory abstractions. A concrete object $o \in \mathbb{O}$ is a finite map from fields $\mathbb{F}$ to their concrete values $\mathbb{V}$ and field-sensitive abstract objects $\mathbb{O}^\sharp$ map fields to their abstract values $\mathbb{V}^\sharp$:

$$\mathbb{O}^\sharp = \mathbb{F} \to \mathbb{V}^\sharp.$$

By using concrete fields as keys, it can precisely analyze points-to relations between addresses.

```
1. class A { A f; }
2. A create() { return new A }; // s1
3. p = create();
4. q = create();
5. x = new A; // s2
6. y = new A; // s3
7. q.f = y;
8. p.f = x;
9. x.f = p;
```

(a) An example code



(b) The flow-sensitive and field-sensitive points-to graph at the end of the program

$\mathbb{X} \times \mathbb{F}^{\leq 2} \to \mathbb{V}^\sharp$

$\mathbb{X}$	$\mathbb{V}^\sharp$	$\mathbb{X}.\mathbb{F}$	$\mathbb{V}^\sharp$	$\mathbb{X}.\mathbb{F}.\mathbb{F}$	$\mathbb{V}^\sharp$
p	$\{s_1\}$	p.f	$\{s_2\}$	p.f.f	$\{s_1\}$
q	$\{s_1\}$	q.f	$\{s_2, s_3\}$	q.f.f	$\{s_1\}$
x	$\{s_2\}$	x.f	$\{s_1\}$	x.f.f	$\{s_2\}$
y	$\{s_3\}$	y.f	$\varnothing$	y.f.f	$\varnothing$

(c) The 2-limiting access-path based memory abstraction with flow-sensitivity at the end of the program

Fig. 10. Field abstractions with access-path based memory abstraction.

In addition, static analysis with object abstraction also frequently uses flow-sensitivity to improve the precision of pointer analysis by performing *strong updates*. If an analysis finds that the abstract object of a field update denotes a single concrete address, then it can perform a strong update by replacing the old value with a new value. Otherwise, it cannot perform strong updates, and the analysis performs a *weak update*, which merges the old value and a new value to a single abstract value. Thus, weak updates may degrade the precision of pointer analysis.

For example, Figure 10 shows a simple code example and the points-to graph of flow-sensitive and field-sensitive pointer analysis at the end of the example code. The variables `p` and `q` have different concrete addresses but the same allocation-site s_1 in the function `create`. Thus, it points to the same abstract address represented by the allocation-site s_1. The variables `x` and `y` point to different abstract address s_2 and s_3, respectively. However, the field updates `q.f = y;` and `p.f = x;` should perform weak updates instead of strong updates, because the allocation site s_1 represents multiple concrete addresses. Such weak updates decrease the precision of static analysis by producing spurious points-to relations. For instance, the following three points-to relations are all spurious points-to relations but the analysis includes all of them:

$$\texttt{p.f} \mapsto s_3 \qquad \texttt{q.f} \mapsto s_2 \qquad \texttt{x.f.f} \mapsto s_3. \tag{9}$$

To improve the analysis precision by reducing weak updates, De and D'Souza [15] introduce an approach to configure abstract memories based on *access paths* [48]. An access path is a pair of a variable and a sequence of fields: $\texttt{x}(\texttt{.f})^* \in \mathbb{X} \times \mathbb{F}^*$. The *access-path based memory abstraction* abstracts concrete memories to a map from access-paths to their abstract values:

$$\mathbb{M}^\sharp = \mathbb{X} \times \mathbb{F}^* \to \mathbb{V}^\sharp.$$

It allows strong updates for field updates `p.f = x;` regardless of whether the variable `p` points to a single address or not. However, a program may have an infinite number of access paths because of recursive data structures or pointers. For example, objects pointed by variables `p` and `x` point

```
1. // If we're in Chrome or Firefox, provide a download link if not installed.
2. if (navigator.userAgent.indexOf('Chrome') > -1 &&
3.     navigator.userAgent.indexOf('Edge') === -1 ||
4.     navigator.userAgent.indexOf('Firefox') > -1) {
5.   ... // Download the React DevTools
6. }
```

Fig. 11. A browser-specific JavaScript code example in React.js.

to each other via the field `f`. It means that infinitely many access paths from them such as `x`, `x.f`, and `x.f.f` exist. Thus, a mechanism to make the number of access paths finite is necessary.

De and D'Souza [15] also utilize a k-limiting approach [47] for access-path based memory abstraction, which considers only access paths that consist of less than or equal to k fields:

$$\mathbb{M}^\sharp = \mathbb{X} \times \mathbb{F}^{\leq k} \rightarrow \mathbb{V}^\sharp.$$

Because it does not contain the points-to relations of access paths that consist of more than k fields ($\mathbb{X} \times \mathbb{F}^{>k}$), it can serve as an additional memory abstraction to increase the precision of another memory abstraction. For example, Figure 10(c) shows the abstract memory at the end of the program with a 2-limiting access-path based memory abstraction with flow-sensitivity. We use a flow-sensitive points-to analysis to supplement missing access paths. Under this memory abstraction, it removes two spurious points-to relations $\texttt{p.f} \mapsto \texttt{s}_3$ and $\texttt{x.f.f} \mapsto \texttt{s}_3$ that exist in the previous memory abstraction, which improves the analysis precision.

3.3 Initial Abstract State

The third analysis parameter is the initial abstract state $d_\iota^\sharp$, which is an abstraction of the set of the possible initial states Σ_ι. In general, $d_\iota^\sharp$ is the minimum abstraction of the possible set of concrete initial states Σ_ι. However, researchers have adjusted the initial abstract states (1) to increase precision by focusing on specific initial states while sacrificing soundness or (2) to increase performance by enduring slight precision degradation. The unsoundness in the second case is often called *soundiness* [55], which is mostly sound but with specific and well-identified unsound choices. For example, let us assume that the possible initial states are $\Sigma_\iota = \{3, 5, 7\}$. If we use the interval domain $\mathbb{D}^\sharp = \{[a, b] \mid a, b \in \mathbb{Z}\}$, then the most tight initial abstract state $d_\iota^\sharp$ is $[3, 7]$. For better performance, we may use over-approximated abstract states $[1, 10]$ as the initial abstract state. To focus on the initial input 5, we may use a *soundy* abstraction $[5, 5]$ as the initial abstract state.

We survey initial abstract states used as parameters of static analysis in the literature. In this section, we classify them into three categories based on their abstract domains: abstract memory (Section 3.3.1), facts in Datalog (Section 3.3.2), and graphs in CFL-reachability (Section 3.3.3).

3.3.1 Initial Abstract Memory. The first case is to configure the initial abstract states in abstract memory domains. As described in Section 3.2, a memory is a pair of an environment for variables and a heap for objects. At the beginning of program execution, the memory initialization depends on the execution contexts. While most static analysis aims to cover all the possible initial memories for sound abstraction, various dynamic features of programming languages often prohibit sound approximation of initial memories, which leads to excessive precision degradation in static analysis. Such dynamic features include reflection and native code interfaces in Java and dynamic code generation and browser-specific APIs in JavaScript.

For example, Figure 11 presents a JavaScript code excerpt from React.js, which is one of the popular JavaScript libraries for building user interfaces. The code detects a user's browser and performs browser-specific actions. The global variable `navigator` is a web API and it represents the status and the identity of a user agent. The property `navigator.userAgent` is a getter of

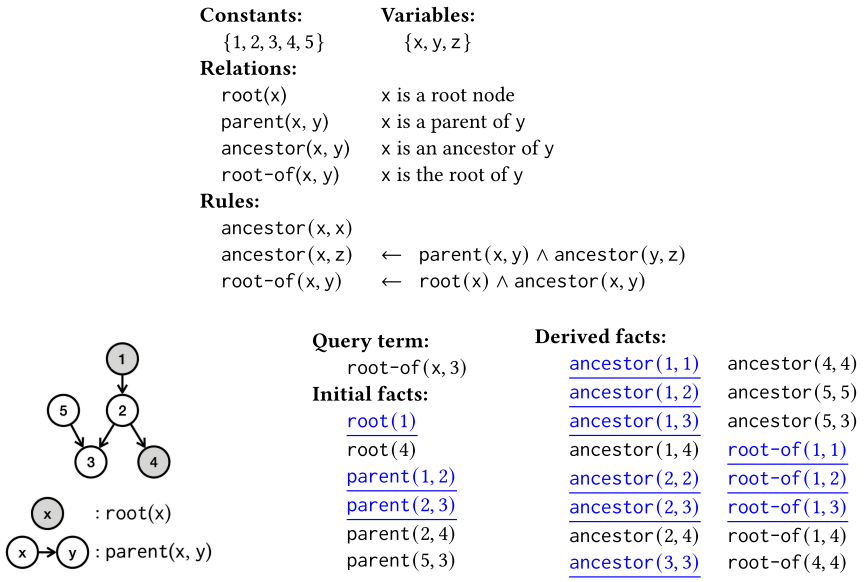


Fig. 12. Datalog program that computes all root nodes that the node 3 is reachable from.

navigator, which returns a string that contains the current browser information. JavaScript developers frequently use navigator.userAgent when they need a client's browser information as presented in Figure 11. Because the structure of navigator depends on specific browsers, soundly approximating its structure for all the browsers with reasonable analysis precision is unrealistic.

To address such a low precision problem, Park et al. [72] parameterize the initial abstract memory only for specific browsers to analyze JavaScript programs in a *soundy* manner. They first extract the concrete memory structure from a target browser using dynamic analysis and then abstract it as the initial abstract memory. For example, assume that the initial abstract memory is constructed from a Chrome browser. Then, the property navigator.userAgent has a string value containing "Chrome". If the analyzer uses the flat string domain that has concrete strings with $\bot$ and $\top$ as abstract strings, then the analyzer decides that the condition of the branch on line 2 is always true. Therefore, it successfully enhances the analysis performance by eliminating spurious control flows when programs are executed in a Chrome browser.

3.3.2 Initial Facts in Datalog. Another target of configurable initial abstract states is the abstract domain for Datalog, which is one of the most widely used programming languages to implement static analysis in a declarative style. Datalog has three basic components: *constants* like $3 \in C$, *variables* like $x, y \in \mathbb{X}$, and *relations* like $\texttt{root} \in \mathcal{R}$. A *term* t is $\texttt{r}(a_1, \ldots, a_n)$ consisting of a relation r with zero or more arguments $a_1, \ldots a_n$ where a_i is either a variable or a constant. A term whose arguments are all constants is a *fact*. A Datalog program takes a set of initial facts and computes all the possible facts based on the *rules* between facts. A rule $t \leftarrow t_1 \wedge \cdots \wedge t_n$ defines the derivation of a target term t from a set of source terms $t_1, \ldots, t_n$. Thus, a Datalog interpreter is like abstract interpretation that computes sets of possible facts (abstract semantics $[\![P]\!]^\sharp$) from the initial facts (initial abstract state $d_I^\sharp$). For example, consider the example in Figure 12. It starts with six facts from root(1) to parent(5, 3) and derives 16 new facts from ancestor(1, 1) to root-of(4, 4) using two rules for ancestor and one rule for root-of.

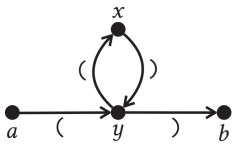


Fig. 13. A simple graph for CFL-reachability.

Liang and Naik [53] present a technique to prune unnecessary initial facts to answer a given query. In the example Datalog program, the query is represented as a term `root-of(x,3)`. To answer this query, only three underlined facts in the initial facts are enough to answer the query. With that three initial facts, the Datalog program produces only nine new facts that are underlined, which can accelerate the analysis without breaking soundness. Liang and Naik performed pre-analysis with more coarse-abstractions to quickly find soundly removable initial facts and repeatedly applied this technique with a sequence of abstractions ordered by granularity. Pruning initial facts is generally applicable to any static analysis implemented in Datalog.

3.3.3 Initial Graphs in CFL-reachability. Reps [77] introduces a new perspective of points-to analysis by converting them to **Context-Free-Language reachability (CFL-reachability)** problems. CFL-reachability is a variant of the traditional graph reachability problem. For a directed graph G whose edges are labeled with alphabets A and a context-free language L over A, CFL-reachability is to determine whether there exists an L-*path* between two given nodes, where a path p is an L-path if $s(p)$, the sequence of alphabets over p, is in the language L. For instance, consider the graph in Figure 13 and the context-free language L that satisfies the grammar $S = SS \mid (S) \mid \epsilon$ over alphabets $A = \{(,)\}$. Then, there exists an L-path from the node a to b because the string of the path $p = a \xrightarrow{(} Y \xrightarrow{)} b$ is in the language L: $s(p) = () \in L$. However, no L-paths exist from x to b or from y to b. For each non-terminal S, a path p is an S-path if and only if the non-terminal S generates the string $s(p)$.

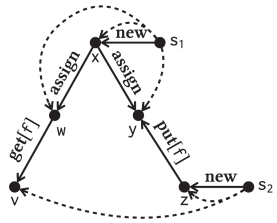
The core idea to convert points-to static analysis to CFL-reachability is to represent field updates and field accesses as open and close parentheses, which shows aliases between pointers as matched parentheses. Thus, *graphs* are abstract states of the CFL-reachability based points-to static analysis. It starts with a given initial graph (initial abstract state $d_\iota^\sharp$) and performs the fixed-point algorithm that converges to a graph (abstract semantics $[\![P]\!]^\sharp$) with additional edges representing analysis results. For example, the points-to information of the code in Figure 14(a) is represented as the graph in Figure 14(b). The **new** edge from $\mathtt{s}_1$ to `x` denotes an object allocation `x = new A();` where $\mathtt{s}_1$ stands for the allocation site of the newly allocated object. The **assign** edge from `x` to `y` denotes a variable assignment `y = x;`, and the **put**[f] and **get**[f] edges from `w` to `v` and from `z` to `y` parameterized by a field name `f` denote a field update `v.f = w;` and a field access `v = w.f;`, respectively. In addition, graphs also use reversed edges labeled with the over-line notation. For instance, the label of an edge from `x` to $\mathtt{s}_1$ is $\overline{\textbf{new}}$, because the label of its corresponding original edge from $\mathtt{s}_1$ to `x` is **new**. Then, we define a context-free language L with three non-terminals: *alias*, *flowsTo*, and $\overline{\textit{flowsTo}}$:

$$\begin{aligned} \textit{alias} &::= \overline{\textit{flowsTo}}\ \textit{flowsTo} \\ \textit{flowsTo} &::= \textbf{new}\ (\textbf{assign} \mid \textbf{put}[\mathtt{f}]\ \textit{alias}\ \textbf{get}[\mathtt{f}])^* \\ \overline{\textit{flowsTo}} &::= (\overline{\textbf{assign}} \mid \overline{\textbf{get}[\mathtt{f}]}\ \textit{alias}\ \overline{\textbf{put}[\mathtt{f}]})^*\ \overline{\textbf{new}} \end{aligned} .$$

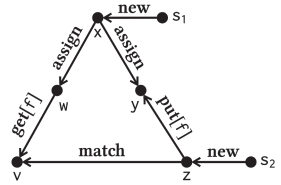
A *flowsTo*-path from a variable `x` to an allocation-site `s` means that the variable `x` points-to `s`. An *alias*-path is a concatenation of a $\overline{\textit{flowsTo}}$-path and a *flowsTo*-path where $\overline{\textit{flowsTo}}$ denotes the reverse of *flowsTo*. Thus, Figure 14(b) shows the final graph of the CFL-reachability based static

```
1. class A { A f; }
2. x = new A(); // s1
3. z = new A(); // s2
4. w = x;
5. y = x;
6. y.f = z;
7. v = w.f;
```

(a) An example code



(b) A graph for the code with dashed edges denoting *flowsTo*



(c) A modified graph with **match** edges

Fig. 14. An example points-to static analysis using CFL-reachability.

analysis where dashed arrows denote *flowsTo*-paths, that is, the newly added edges during the analysis.

Sridharan et al. [91] introduce a technique to configure the initial graph of CFL-reachability using **match** edges. The following revised grammar with **match**:

$$\begin{aligned} \textit{flowsTo} &::= \textbf{new}\ (\textbf{assign} \mid \textbf{put}[\mathtt{f}]\ \textit{alias}\ \textbf{get}[\mathtt{f}] \mid \textbf{match})^* \\ \overline{\textit{flowsTo}} &::= (\overline{\textbf{assign}} \mid \overline{\textbf{get}[\mathtt{f}]}\ \textit{alias}\ \overline{\textbf{put}[\mathtt{f}]} \mid \textbf{match})^*\ \overline{\textbf{new}} \end{aligned}$$

enables selectively drawing **match** edges between variables that update and access values of the same field name. For example, let us analyze the example code with the modified graph in Figure 14(c) and the modified context-free language. Then, an *alias*-path from z to v becomes:

$$\mathtt{z} \xrightarrow{\overline{\textbf{new}}} \mathtt{s}_2 \xrightarrow{\textbf{new}} \mathtt{z} \xrightarrow{\textbf{match}} \mathtt{v},$$

while its corresponding original *alias*-path is:

$$\mathtt{z} \xrightarrow{\overline{\textbf{new}}} \mathtt{s}_2 \xrightarrow{\textbf{new}} \mathtt{z} \xrightarrow{\textbf{put}[\mathtt{f}]} \mathtt{y} \xrightarrow{\overline{\textbf{assign}}} \mathtt{x} \xrightarrow{\overline{\textbf{new}}} \mathtt{s}_1 \xrightarrow{\textbf{new}} \mathtt{x} \xrightarrow{\textbf{assign}} \mathtt{w} \xrightarrow{\textbf{get}[\mathtt{f}]} \mathtt{v}.$$

Thus, using **match** edges makes static analysis efficient by reducing the lengths of *alias*-paths. However, abusing **match** edges may decrease analysis precision because of false relations. For example, consider removing the variable assignment `y = x;` on line 5. Then, the edge from x to y labeled with **assign** is removed and no *alias*-path exists from z to v. However, in the modified graph in Figure 14(c), an *alias*-path from z to v still exists because of the **match** edge, which is a false relation. Therefore, finding an appropriate balance between the analysis performance and precision caused by **match** edges is an important task. Sridharan et al. [89, 91] propose to start analysis with the initial graph having all **match** edges and to gradually refine them using previous analysis results. Xu et al. [115] optimize this approach using context-sensitive *must-not-alias* information by constructing Interprocedural Symbolic Points-to Graph. Dietrich et al. [16] introduce **bridge** edges based on transitive-closure data-structures unlike field-based **match** edges.

Instead of over-approximation of the initial graph with **match** edges, researchers accelerate CFL-reachability-based points-to analysis without any precision loss. Vedurada and Nandivada [106] utilize caches called *batches* extracted from previous analysis with different queries. Among newly added edges during previous analysis with different queries, a subset of them relevant to the current query is loaded to the initial graph. However, Li et al. [52] introduce a graph simplification algorithm for Dyck-reachability, which includes CFL-reachability. Therefore, in addition to CFL-reachability, it supports static analysis based on other Dyck-reachability such as Synchronized Pushdown Systems reachability [87] and Linear-Conjunctive-Language reachability [118].

```
1. a = 1;
2. b = a + 2;
3. c = a - 3;
4. d = b - 2;
5. e = c + a;
6. assert(d < 42);
```

(a) An example code

```
1. a = 1;
2. b = a + 2;
3.
4. d = b - 2;
5.
6. assert(d < 42);
```

(b) Program slicing

```
1. a = 1;
2. b = a + 2;
3.
4. d = b - 2;
5.
6.
```

```
1. a = 1;
2.
3. c = a - 3;
4.
5. e = c + a;
6.
```

(c) Program clustering

Fig. 15. Selected view transitions.

3.4 Abstract Transfer Function

The last analysis parameter is the abstract transfer function $F^\sharp$. A program is a transition system and its collecting semantics $[\![P]\!]$ is computed by iterations over the concrete transfer function F. Static analysis over-approximates F to an abstract transfer function $F^\sharp$ to compute an abstract semantics $[\![P]\!]^\sharp$. The precision and performance of static analysis depend on how to define the abstract transfer functions. The abstract transfer function $F^\sharp$ consists of two orthogonal parts: an abstract one-step execution $\mathbf{step}^\sharp$ and the join operator $\sqcup$:

$$\forall d^\sharp \in \mathbb{D}^\sharp.\ F^\sharp(d^\sharp) = d^\sharp \sqcup \mathbf{step}^\sharp(d^\sharp). \tag{10}$$

Moreover, the abstract one-step execution $\mathbf{step}^\sharp_\delta$ under the analysis sensitivity δ is defined with the view transition $[\![\pi' \to \pi]\!]^\sharp$ for views π' and π:

$$\mathbf{step}^\sharp_\delta(d^\sharp_\delta) = \lambda\pi \in \Pi.\ \bigsqcup_{\pi' \in \Pi} [\![\pi' \to \pi]\!]^\sharp \circ d^\sharp_\delta(\pi'). \tag{11}$$

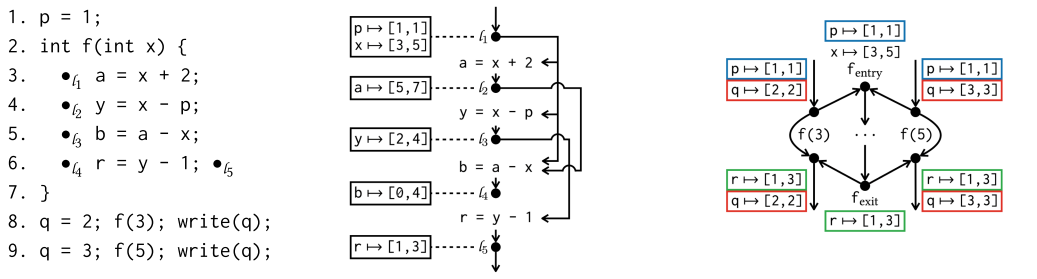
In this section, we describe how the existing research configures abstract transfer functions for parametric static analysis. We first explain how researchers configure the abstract semantics of view transitions to define abstract one-step executions in two ways: pruning specific view transitions (Section 3.4.1) and modifying the abstract semantics of view transitions (Section 3.4.2). Then, we survey the join operator $\sqcup$ for parametric static analysis (Section 3.4.3).

3.4.1 Selected View Transition. The abstract semantics of a view transition $[\![\pi' \to \pi]\!]^\sharp$ approximates all the possible behaviors of a view transition from π' to π. However, a sound abstraction of all program behaviors may not be the best solution for static analysis with specific goals. For example, the example code in Figure 15(a) includes an `assert` statement. If the goal of a static analysis is to prove the assertion, then it does not need to analyze some program parts. To focus on only relevant parts of a given program for static analysis, researchers have proposed refined abstract transfer functions by pruning out irrelevant view transitions:

$$[\![\pi' \to \pi]\!]^\sharp_{\mathtt{sel}}(d^\sharp) = \begin{cases} \bot & \text{if } \pi' \to \pi \in \mathcal{R}_{\mathtt{block}}, \\ d^\sharp & \text{if } \pi' \to \pi \in \mathcal{R}_{\mathtt{bypass}}, \\ [\![\pi' \to \pi]\!]^\sharp(d^\sharp) & \text{otherwise,} \end{cases}$$

where $\mathcal{R}_{\mathtt{block}}$ and $\mathcal{R}_{\mathtt{bypass}}$ are sets of selected view transitions to refine the abstract semantics by blocking or bypassing analysis flows, respectively. We categorize them to two approaches.

- **Program Slicing:** *Program slicing* techniques focus on analyzing specific parts of programs by using selection of view transitions. For example, Figure 15(b) shows a sliced program that is necessary to prove the assertion `assert(d < 42)`. Because the assertion uses only the variable `d`, data flow information only for the variable `d` is enough to prove the assertion. Thus, a program slicing technique can soundly remove the statements on lines 3 and 5, which do not affect the



(a) An example code (b) Localization for variables (c) Localization for function calls

Fig. 16. Modified view transitions.

value of d. Researchers have utilized concrete executions to filter target files [72], functions [17], and blocks [18, 107, 110] in a *soundy* manner. However, several researchers introduce statement-level program slicing for dataflow analysis using value flow graphs [1], pointer analysis using deductive reachability formulation [31], and set-based reasoning [84]. Moreover, Heo et al. [35] use machine learning to selectively skip library calls in a soundy way.

- **Program Clustering:** *Program clustering* aims for high performance by dividing a given program to multiple sub-programs and to analyze them in parallel or using lower sensitivity. For example, let us analyze the example code in Figure 15(a) using program clustering for parallel computation. Because the variable d depends on only the variables a and b, and e depends on only a and c, we can divide the code to two sub-programs shown in Figure 15(c). Even though it analyzes two sub-programs separately, its analysis result is exactly the same as the analysis result of a whole program analysis. Thus, by leveraging parallel computation, program clustering can accelerate static analysis without losing analysis precision. Kahlon [40] utilize *Steensgaard partitions* [92] to divide given programs and Zhang et al. [121] use a pre-processor to construct weighted inter-procedural CFGs of given binary programs and sample multiple paths to be analyzed in parallel. However, Yu et al. [116] propose a technique called *level-by-level strategy* to utilize flow-insensitive analysis with the same precision of flow-sensitive analysis. They first partition given programs based on *points-to levels* and flow-insensitively analyze them in decreasing order of their levels, which results in high performance of flow-insensitivity and high precision of flow-sensitivity.

3.4.2 Modified View Transition. Existing work on parametric static analysis configures abstract transfer functions not only by pruning out view transitions but also by modifying them. We categorize modified view transitions used in previous work to four approaches: *variable localization*, *function call localization*, *semantics refinement*, and *opaque function modeling*.

- **Variable Localization:** While flow-sensitivity enables precise static analysis by distinguishing states with view abstraction using control states $\mathbb{L}$, it may cause explosion of the number of abstract states by making each view have its own abstract state. To alleviate the problem, *variable localization* [21, 28, 64, 82, 95–97] prunes out abstract states and connects necessary ones using def-use relations. For example, Figure 16(a) shows an example code with a function f and Figure 16(b) depicts the main idea of variable localization for f. We use the interval domain to approximate numbers; an abstract value is an interval represented as $[a, b]$ for two numbers a and b. At the entry of the function f, the abstract state consists of mappings for the global variable p and the parameter x. The statement a = x + 2; produces only a mapping for the local variable a, a ↦ [5,7], which is not passed to the next statement y = x - p; but

directly passed to `b = x - a;` that is a use-site of the variable `a`. In a similar way, the mapping `y` $\mapsto$ `[2,4]` in the abstract state at ℓ_3 is directly passed to `r = y - 1;`. This technique significantly reduces the size of each abstract state and increases analysis performance. Researchers perform *Sparse Value-Flow Analysis* [95–97] to construct ***Sparse Value-Flow Graphs (SVFGs)*** for data dependency. In addition, Shi et al. [82] use *Symbolic Expression Graphs*, which are intra-procedural SVFGs, and Fan et al. [21] use *Use-Flow Graphs*, which encode both definitions and uses of heap objects.

- **Function Call Localization:** Localization techniques are also applicable to function calls. Figure 16(c) shows *function call localization* [63, 68] for calls of `f` on lines 8 and 9 of the example code. Because some information in abstract states like the mapping for the global variable `q` is not necessary to analyze the body of `f`, a function localization technique prunes out the mappings `q` $\mapsto$ `[2,2]` and `q` $\mapsto$ `[3,3]` at the call-sites of `f`, and directly passes them to their corresponding return-sites. At the entry of the function `f`, the abstract state consists of only the global variable `p` and the parameter `x` without `q`. Finally, the abstract value of `r` becomes `[1,3]` at the exit point of the function body, and it merges with `q` $\mapsto$ `[2,2]` and `q` $\mapsto$ `[3,3]`, respectively. Such localization for function calls not only reduces the sizes of abstract states of function bodies but also increases the analysis precision of function calls. Without call context sensitivity, two abstract values of `q` at two call-sites of `f` should be merged as `[2,3]`, which maps `q` to `[2,3]` at the return-sites, even though the body of `f` does not use `q`. Then, the arguments of two function calls of `write` have `[2,3]`. However, the localization technique for function calls directly passes the irrelevant parts of abstract states to the corresponding return-sites, which keeps the mapping for the variable `q` precisely. Now, the arguments of two function calls of `write` have more precise abstract values `[2,2]` and `[3,3]`, respectively. Oh et al. [63] perform function call localization using an efficient pre-analysis to estimate the set of locations that are to be accessed during the analysis of each code block. Oh and Yi [68] optimize the technique with *bypassing*, which localizes input memory states only with memory locations that the function directly accesses and bypasses the other locations to transitively called functions.
- **Semantics Refinement:** Researchers [4, 14, 22, 35, 37, 44, 79, 86, 88, 93, 109] have enhanced the performance or precision of static analysis by *semantics refinement*, which is a technique to restrict or refine a view transition in a various way:

$$\forall d^\sharp \in \mathbb{D}^\sharp.\ [\![\pi' \to \pi]\!]^\sharp_{\mathtt{ref}}(d^\sharp) \sqsubseteq [\![\pi' \to \pi]\!]^\sharp(d^\sharp).$$

Jensen et al. [37] present a technique to replace JavaScript `eval` call with a single concrete string value. Schäfer et al. [79] extend it to enhance the analysis precision by replacing *determinate* expressions, which always have the same value at a given program point, with their corresponding concrete values. In the example code, because `a`'s value is `x + 2`, `b`'s value is always 2. Thus, we can replace the statement `b = a - x;` with `b = 2;` and it increases the analysis precision at ℓ_4 from `b` $\mapsto$ `[0,4]` to `b` $\mapsto$ `[2,2]`. While they use dynamic information to detect determinacy in a *soundy* way, Andreasen and Møller [4] present a sound technique to detect them. Researchers also perform backward analysis before [9, 88] or during [93] static analysis to refine abstract values. Beyond abstract values, researchers refine control flows or abstract states. For control flows, Grech et al. [22] use dynamic information to refine code loaded at runtime, Heo et al. [35] leverage machine learning for unsound loop unrolling, Sotiropoulos and Livshits [86] refine asynchronous event loops via callback graphs, and Cyphert et al. [14] refine path expressions, which are regular expressions that recognize all feasible execution paths. For abstract states, Ko et al. [44] present a sound way to restrict abstract states for each control point using other analysis results. Wei et al. [109] apply Futamura projection to specialize abstract semantics using deterministic program parts, and He et al. [30] remove

```
1. •ℓ1 x = 0;
2. •ℓ2 while (x < 100) {
3. •ℓ3   x++;
4. } •ℓ4
```

(a) An example code

Iter	$d^\sharp$	$\mathbf{step}^\sharp(d^\sharp)$	$F^\sharp(d^\sharp)$
1	$[0,0]$	$[1,1]$	$[0,20]$
2	$[0,20]$	$[1,21]$	$[0,50]$
3	$[0,50]$	$[1,51]$	$[0,100]$
4	$[0,100]$	$[1,99]$	$[0,100]$

(b) Iteration of the abstract transfer function $F^\sharp$

Fig. 17. Widening with thresholds $\mathbb{Z}_{sel} = \{0, 20, 50, 100, +\infty\}$.

redundant constraints in polyhedra and octagon domains during view transitions via machine learning.

- **Opaque Function Modeling:** Real-world programs often use *opaque* libraries in binary or different language. For such opaque functions, static analysis often ignores them by sacrificing soundness, or uses manually constructed models to mimic their behaviors with excessive labors. To overcome these problems, researchers present techniques to automatically model their behaviors. One approach is to infer their behaviors using their uses [57] or types extracted from library specifications [6, 70]. Recent work leverages machine learning [8, 36], natural language processing [117], or dynamic information [7, 71, 105] to automatically model them.

3.4.3 Join Operator. As described in Section 2, with analysis sensitivities, an abstract one-step execution $\mathbf{step}^\sharp_\delta : \mathbb{D}^\sharp_\delta \to \mathbb{D}^\sharp_\delta$ merges abstract states via the *join operator* ($\sqcup$) for each control state after executing view transitions:

$$\mathbf{step}^\sharp_\delta(d^\sharp_\delta) = \lambda\pi \in \Pi. \bigsqcup_{\pi' \in \Pi} [\![\pi' \to \pi]\!]^\sharp \circ d^\sharp_\delta(\pi').$$

Analysis performance heavily depends on the order of views visited during join in each abstract one-step execution: *the traversal strategy* or *worklist algorithm*. Ramu et al. [76] selected an optimal traversal strategy for each program using analysis properties and control flow graph properties.

Also, the tight definition of the join operator in the abstract transfer function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$:

$$F^\sharp(d^\sharp) = d^\sharp \sqcup \mathbf{step}^\sharp(d^\sharp)$$

increases analysis precision, but it also incurs performance degradation. Consider the code in Figure 17(a) where the abstract values of the variable x in the interval domain are abstract states: $\mathbb{D}^\sharp = \{[a,b] \mid a, b \in \mathbb{Z}\}$. At ℓ_2, the initial abstract state is $d^\sharp = [0,0]$. After finishing the first iteration of the loop, the new abstract state becomes $\mathbf{step}^\sharp(d^\sharp) = [1,1]$ and it is merged with the old one: $F^\sharp(d^\sharp) = d^\sharp \sqcup \mathbf{step}^\sharp(d^\sharp) = [0,0] \sqcup [1,1] = [0,1]$. Because the join operator $\sqcup$ is the least upper bound of two abstract states, the fixpoint computation for the example code requires 100 iterations.

To address such a performance problem, the abstract transfer function can use a *widening operator* ∇, which returns a coarser upper bound of two abstract states than their least upper bound:

$$F^\sharp(d^\sharp) = d^\sharp \nabla \mathbf{step}^\sharp(d^\sharp)$$

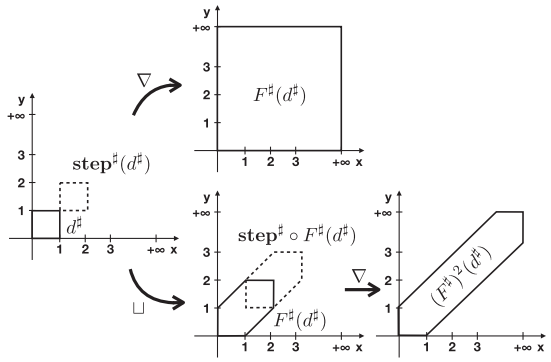
Parametric static analysis can use a widening operator as an analysis parameter in two ways:

- **Widening with Thresholds:** The first approach restricts the expressiveness of ∇ with a set of *thresholds* $\mathbb{Z}_{sel}$ [10, 11, 49, 83] for the interval domain. For an abstract state $[a,b]$ in the interval domain, thresholds $\mathbb{Z}_{sel}$ restrict the widening operator results as follows:

$$[a,b] \nabla [c,d] = [\max\{n \in \mathbb{Z}_{sel} \mid n \le \min(a,c)\}, \min\{n \in \mathbb{Z}_{sel} \mid \max(b,d) \le n\}].$$

```
1. assume 0 ≤ x ≤ 1 and 0 ≤ y ≤ 1
2. •ℓ1 while ( ? ) {
3. •ℓ2    x = x+1;
4. •ℓ3    y = y+1;
5. •ℓ4 } •ℓ5
```

(a) An example code



(b) Delayed widening for octagon domain after one iteration

Fig. 18. Delayed widening.

For example, Figure 17(b) describes the results of the widening operator with thresholds $\mathbb{Z}_{sel} = \{0, 20, 50, 100, +\infty\}$ for the interval domain of the variable x in the example code. For the first iteration, the old abstract state is $d^\sharp = [0, 0]$ and the next one is $\textbf{step}^\sharp(d^\sharp) = [1, 1]$ because of the statement x++;. Thus, the widening of two abstract states becomes $[0, 20]$, because 20 is the smallest threshold greater than or equal to 1. In the second and third iterations, the abstract states grow to $[0, 50]$ and $[0, 100]$. Then, the iteration converges to $[0, 100]$ and it takes only four iterations to reach the fixpoint. The performance and precision of the analysis using ∇ with thresholds highly depend on selection of the thresholds. Thus, researchers utilize machine learning to select thresholds of ∇. Cha et al. [11] leverage a Bayesian optimized machine learning algorithm to select thresholds of ∇ for the interval domain. Singh et al. [83] select thresholds of ∇ for the polyhedra domain with various constraint removal and merge strategies for approximations via **Reinforcement Learning (RL)**.

- **Delayed Widening:** Another approach delays the time to apply ∇ in the abstract transfer function $F^\sharp$. With a parameter n, we can refine $F^\sharp$ to delay ∇ until n iterations:

$$F^\sharp(d^\sharp) = \begin{cases} d^\sharp \sqcup \textbf{step}^\sharp(d^\sharp) & \text{if (\# of iterations)} \leq n, \\ d^\sharp \nabla \textbf{step}^\sharp(d^\sharp) & \text{otherwise.} \end{cases}$$

We dub this technique *delayed widening* and Gulavani and Rajamani [24] applied it for the relational domain. For example, Figure 18(a) shows an example code with assumptions on the variables x and y. An abstract state of the relational domain is a set of linear relations between variables and it represents the conjunction of the relations. The widening operator between two abstract states is not commutative but focuses on the left one:

$$d^\sharp \nabla d'^\sharp = \{r \in d^\sharp \mid r \text{ covers } d'^\sharp\}.$$

Thus, the widening operation removes several relations in the left abstract state $d^\sharp$ when they do not cover all the relations in the right abstract state $d'^\sharp$. For example, the upper part in Figure 18(b) describes the analysis result of the example code with the widening operator. In the first iteration, the old abstract state is $d^\sharp = \{x \geq 0, x \leq 1, y \geq 0, y \leq 1\}$ and the new one is $\textbf{step}^\sharp(d^\sharp) = \{x \geq 1, x \leq 2, y \geq 1, y \leq 2\}$. Among the relations in $d^\sharp$, $x \leq 1$ and $y \leq 1$ do not cover $\textbf{step}^\sharp(d^\sharp)$ such as $(x, y) = (2, 2)$ in $\textbf{step}^\sharp(d^\sharp)$. Therefore, the widening operator removes them and the result is the fixpoint of the analysis: $F^\sharp(d^\sharp) = d^\sharp \nabla \textbf{step}^\sharp(d^\sharp) = \{x \geq 0, y \geq 0\}$. Unfortunately, the analysis result is quite imprecise, because it does not provide any relations

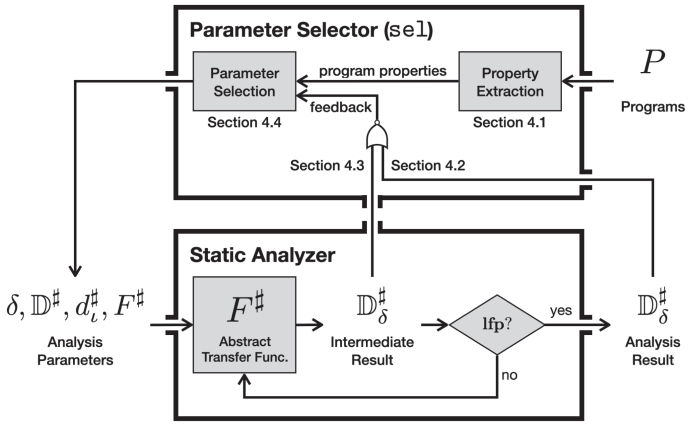


Fig. 19. Selection of analysis parameters in parametric static analysis.

between the variables x and y. On the contrary, the delayed widening technique can resolve this problem via the combination of the $\sqcup$ and ∇ operators. The lower part in Figure 18(b) depicts the analysis result of the example code with the delayed widening technique. The analysis applies ∇ after one iteration, which means that the first abstract transfer function uses $\sqcup$ instead of ∇: $F^\sharp(d^\sharp) = d^\sharp \sqcup \mathbf{step}^\sharp(d^\sharp) = \{x \geq 0, x \leq 2, y \geq 0, y \leq 2, y \leq x + 1, y \geq x - 1\}$. In the graph, the area tagged with $F^\sharp(d^\sharp)$ depicts this abstract state. In the next iteration, the new abstract state $\mathbf{step}^\sharp \circ F^\sharp(d^\sharp)$ becomes $\{x \geq 1, x \leq 3, y \geq 1, y \leq 3, y \leq x + 1, y \geq x - 1\}$. Now, the abstract transfer function uses ∇ and it removes only two relations $x \leq 2$ and $y \leq 2$. Thus, the result of the second iteration becomes $(F^\sharp)^2(d^\sharp) = F^\sharp(d^\sharp) \nabla \mathbf{step}^\sharp \circ F^\sharp(d^\sharp) = \{x \geq 0, y \geq 0, y \leq x + 1, y \geq x - 1\}$, which is the fixpoint of the analysis. Finally, we efficiently compute a more precise analysis result than simply using the join operator by delaying the widening operator.

4 PARAMETER SELECTION

In parametric static analysis, the *parameter selector* selects analysis parameters for analysis purposes using program properties, as shown in Figure 19. We describe ways to extract program properties (Section 4.1) to obtain feedback from previous analysis results (Section 4.2) or intermediate analysis results (Section 4.3), and to select parameters using properties and feedback (Section 4.4).

4.1 Property Extraction

In this section, we explain how to extract program properties used in parametric static analysis.

Syntactic Analysis. Various techniques extract properties via *syntactic analysis*. Block-level octagon packing [10, 59] applies octagon packing to variables in the same program blocks. Correlation-tracking by Sridharan et al. [90] and composite abstraction by Ko et al. [45, 46] introduce their own syntactic patterns to adjust loop sensitivity. Wei and Ryder [111] apply partial flow-sensitivity to syntactically determined state-update statements. Kastrinis and Smaragdakis [41] use call-site sensitivity for static method calls and object or type sensitivity for the others. Zhang et al. [121] extract weighted inter-procedural control-flow graphs from binary programs using syntactic analysis.

Dynamic Analysis. Another approach to extract program properties is *dynamic analysis*. Researchers [17, 18, 107, 110] utilize multiple concrete execution traces to focus on or to exclude program parts covered by them in static analysis. Execution traces are also useful to construct

modeling of opaque functions [7, 71, 105]. During concrete execution, collecting dynamically loaded code such as dynamic file loading [72] in Node.js or Java reflections [22] also lessens the soundness problem of static analysis. In addition, Naik et al. [62] keep whether each address is thread-local or thread-escaping during dynamic analysis to use the information when partitioning allocation sites. Schäfer et al. [79] detect determinacy using concrete execution, and Wei and Ryder [111] use dynamic analysis as well as syntactic analysis to detect state-update statements.

Specification Analysis. Researchers extract behaviors of opaque functions from their specifications. Bae et al. [6] introduce a technique to automatically model behaviors of vendor-specific Web APIs using their types in specifications written in Web IDL, and Park [70] uses type information in TypeScript declaration files. Zhai et al. [117] leverage natural language processing techniques to extract behaviors of Java API functions from their documentation written in a natural language.

Access Analysis. Access analysis identifies which variables or paths are accessible in a given program. Function call localization techniques [63, 68] perform static analysis only for accessible variables in each function call via access analysis. De and D'Souza [15] pre-calculate access paths of fields to limit their lengths in field abstraction with a pre-defined upper-bound k.

Dataflow Analysis. Dataflow analysis tracks value flows through control flows. *Sparse value-flow analysis* tracks value flows sparsely through def-use chains or SSA forms. Variable localization techniques [21, 28, 64, 82, 95–97] utilize it to refine intra-procedural view transitions using value flows of variables. Adams et al. [1] slice programs in a statement level by using value-flow graphs constructed by flow-insensitive dataflow analysis. Li et al. [50] detect value flow patterns of imprecise context-insensitive pointer analysis to select functions to analyze with context sensitivity.

Equivalence-based Analysis. Steensgaard [92] uses equivalence constraints to divide variables to *Steensgaard partitions* instead of subset constraints in Andersen's points-to analysis to enhance analysis performance. Kahlon [40] divides a given program based on variable partitions and performs Andersen's points-to analysis for each sub-program. Yu et al. [116] and Sui et al. [98] extend it to assign a *points-to level* for each variable partition and flow-insensitively analyze sub-programs in decreasing order of their levels while preserving the precision of flow-sensitivity. Another use of equivalence-based analysis is variable substitutions [27, 78] or statement-level program slicing [84].

Impact Analysis. Oh et al. [65, 66] first perform *impact analysis* of a given context sensitivity using a simple abstract domain quickly. Using the pre-analysis results, they perform main analysis using a more complex interval domain with high sensitivity for functions with high impacts.

Points-to Analysis. Points-to analysis infers relations between program components. Object allocation graphs represent relations between object allocations for call-edge selection [101] or different context sensitivity per function [51]. Tan et al. [102] construct **field points-to graphs (FPGs)** and check the equivalence between FPGs to partition allocation sites. The sizes of points-to sets relevant to functions, variables, and locations are also useful [29, 85]. Ko et al. [44] use the result of another points-to analyzer as an upper bound of its main analysis to provide better performance.

Graph Analysis. Researchers often use CFL-reachability analysis as a pre-analysis. It obtains must-not-alias information [115] to find imprecision of context-insensitive pointer analysis [56] or to add bridge edges to another CFL-reachability analysis [16]. Ramu et al. [76] perform graph analysis to detect cyclicity of control flow graphs when selecting a traverse strategy for the join operator. Li et al. [52] perform graph simplification algorithms to reduce the sizes of the initial abstract states in CFL-reachability analysis.

4.2 Feedback from Previous Analysis Results

Iterative parametric static analysis repeatedly performs static analysis by updating analysis parameters using previous analysis results until a specific condition is satisfied.

Client-driven Feedback. Researchers use backward analysis to localize program parts parts that cause failures of client analysis such as variables [25, 120], allocation sites [88], and function calls [26, 75]. CFL-reachability based analysis [89, 91] repeatedly remove **match** edges to increase analysis precision when client analysis uses **match** edges. Gulavani and Rajamani [24] localize the widening operator ∇ relevant to failures of client analysis and replace it with the join operations $\sqcup$. For Datalog-based analysis, researchers localize call-sites maximally relevant to failed parts of client analysis based on a heuristic model [119] and a learnt probabilistic model [23].

Refinement of Initial Abstract States. Datalog-based analysis refines the initial abstract states of subsequent analysis using previous analysis results. When analysis does not use some initial facts, its subsequent analysis does not use them either [53]. In addition, Vedurada and Nandivada [106] use previous analysis results as caches to avoid redundant analysis in subsequent analysis. They construct caches called *batches* extracted from the previous analysis with different queries in CFL reachability-based analysis and lift up the initial graphs the next analysis using batches.

4.3 Feedback from Intermediate Analysis Results

To avoid restarting static analysis multiple times, researchers propose techniques to configure analysis parameters *on-the-fly* using intermediate analysis results and continue the analysis iteration with different analysis parameters. One exception is Wei et al. [113], who restart the analysis with new analysis parameters when intermediate analysis results become too imprecise.

Optimization for Redundant Analysis. One approach to utilize intermediate analysis results is to reduce redundant analysis. Whaley and Lam [114] utilize BDDs to merge calling contexts of redundant function call analysis to improve the analysis performance without any precision loss.

Semantics Refinement. Semantics refinement techniques may get feedback from intermediate analysis results. Several researchers collect determinacy information during JavaScript static analysis to modify `eval` function calls with concrete string values [37] and to modify determinate expressions with their corresponding values [4]. Heintze and Tardieu [31] use deductive reachability to slice programs, Stein et al. [93] perform on-demand backward analysis to refine abstract values, and Sotiropoulos and Livshits [86] modify callback graphs for asynchronous event loops during static analysis. For algebraic program analysis, Cyphert et al. [14] refine path expressions in intermediate analysis results. Wei et al. [109] specialize abstract semantics using deterministic parts in intermediate analysis results inspired by partial evaluation using Futamura projection.

Opaque Function Modeling. Madsen et al. [57] use intermediate analysis results to model the behaviors of opaque functions automatically. They detect use patterns of opaque functions during static analysis and automatically infer their modeling.

4.4 Parameter Selection

Parametric static analysis selects analysis parameters using program properties and feedback from analysis results. Because selecting and configuring the best combination of analysis parameters manually is difficult [25, 26, 56, 64–67, 72, 85, 89], recent researchers have taken data-driven approaches for selection of analysis parameters using various statistical learning or machine learning models. They view parametric static analysis as a *search problem* to find the *best* analysis parameter from a specific set of analysis parameters explained in Section 3. The score of analysis parameters is defined with a function $score : \Psi \to \mathbb{R}$ from parameters Ψ to real values depending on analysis

purposes. For example, when an analysis purpose is to increase analysis performance while preserving the analysis precision, the score $score(\psi)$ of a parameter ψ would be 0 when the analysis with ψ degrades analysis precision, otherwise, $1/t$ where t is the analysis time. For this search problem on a search space Ψ with the score function *score*, researchers utilize *guided random search*, *statistical classification*, *reinforcement learning*, and *learnt probabilistic model*.

4.4.1 Guided Random Search. A naïve solution is to exhaustively or randomly traverse candidates in a given search space. However, it is infeasible when the the search space size is excessively large: $|\Psi| \gg 1$. Thus, researchers perform *guided random search*. Heule et al. [36] perform a local search inspired by Markov chain Monte Carlo sampling and the Metropolis-Hastings algorithm [5] to find modeling of JavaScript built-in functions. They randomly search the space near the current analysis parameter ψ by randomly mutating it and pick the next parameter whose score is the highest:

$$\underset{\psi' \in M}{\text{argmax}}\ score(\psi'),$$

where $M = \{\psi'_1, \ldots, \psi'_n\}$ is a set of n analysis parameters mutated from ψ.

Another solution is to leverage *statistical learning* for *program components* $\Psi = \mathcal{P}(\mathbb{J})$ in analysis parameters. A component $j \in \mathbb{J}$ may be a function for partial context sensitivity in Section 3.1.2 or a variable for variable selection in Section 3.2.1. To refine analysis parameters, Liang et al. [54] first sample n analysis parameters $M = \{\psi'_1, \ldots, \psi'_n\}$ by randomly adding new components to the current parameter ψ with the selection probability α: $mutate(\psi) = \psi \cup \{j \mid \text{randomly select } j \in \mathbb{J} \setminus \psi \text{ with } \alpha\}$. Then, they insert the best component to ψ using the statistical information extracted from M:

$$\underset{j \in \mathbb{J} \setminus \psi}{\text{argmax}} \sum \{score(\psi') \mid \psi' \in M \wedge j \in \psi'\}.$$

Thus, it changes the search space from analysis parameters Ψ to program components $\mathbb{J}$.

Using *features* of components to indirectly find the best component by searching the best weight vector $\mathbf{w}$, the search space once more changes from components to weight vectors. A *feature vector* $f(j)$ of a component j is defined with n different feature functions $f = \langle f_1, \ldots, f_n \rangle$ that represent whether components have the features. For a given *weight vector* $\mathbf{w} \in [0,1]^n$, a component j is selected when its estimated score $f(j) \cdot \mathbf{w}$ is the highest. This approach is efficient, because it infers the weight vectors using information from different programs, and it selects analysis parameters without additional searching after learning weight vectors. Oh et al. [67] use *Bayesian optimization* in a random search algorithm for weight vectors, and Cha et al. [11] present an *oracle*-guided random search to utilize a standard gradient decent algorithm to select a threshold of ∇.

Boolean formulas defined over features can replace weight vectors. Since such Boolean formulas are non-linear while weight vectors are linear, the random search using Boolean formulas is more expressive than using weight vectors. Jeong et al. [39] use vectors of k Boolean formulas as learning targets to determine different k per function for k-context sensitivity. While they use a greedy algorithm to learn Boolean formulas, because target analysis parameters are monotone in terms of precision and performance, the context tunneling technique is not monotone. Thus, Jeon et al. [38] present a non-greedy algorithm for such non-monotone analysis parameters.

4.4.2 Statistical Classification. Statistical classification is a supervised learning that classifies given input vectors. A classifier $classify : \{0,1\}^k \to \{0,1\}$ takes a feature vector $f(j)$ for each component $j \in \mathbb{J}$ and classifies whether it is selected or not for the analysis parameter ψ: $classify(f(j)) = 1 \Leftrightarrow j \in \psi$ and $classify(f(j)) = 0 \Leftrightarrow j \notin \psi$.

Decision tree learning is one of the most popular classification techniques. It constructs a decision tree using training data and classifies inputs according to the constructed decision tree. Wei

and Ryder [112] first use a C4.5 classifier [74] to learn relationships between function features and precision of context-sensitivity and design heuristics based on the constructed decision tree to select different context sensitivity per function. Bielik et al. [8] extend ID3 [73] algorithm to automatically infer abstract semantics of JavaScript built-in functions. To select variable relations for octagon domain, Heo et al. [33] utilize *logistic regression* [60], which is a statistical model that uses logistic functions to model binary dependent variables. They use the impact analysis for octagon domain introduced by Oh et al. [65] to generate labels with 30 features for relations of two variables. For selectively unsound analysis with harmless unsoundness, Heo et al. [35] leverage **One-Class Support Vector Machine (OC-SVM)** classifier [80]. They select loops for unsound unrolling and library function calls to skip using the OC-SVM classifier learned with 22 loop features and 15 library function call features. He et al. [30] utilize **Graph Convolutional Network (GCN)** [43] to over-approximate the join operator for polyhedra and octagon domains. After join operation, they construct graphs with constraints in abstract states as nodes and relationships between them as edges. Over the graphs, they train the model using GCN to decide which constraints to remove.

Unfortunately, the quality of parametric static analysis using classification is highly dependent on the quality of features. To tackle this problem, Chae et al. [12] automate the feature design process by using reduced and abstracted programs as features. They apply this technique to interval, pointer, and octagon analysis, and the average numbers of generated features are 38, 45, and 44.

4.4.3 Reinforcement Learning. RL [99] is a machine learning technique with an *agent* learning by interacting with its *environment* to reach a goal. The agent starts from an initial environment state $s_0 \in S$ where S is a set of environment states. At each time $t = 0, 1, 2, \ldots$, the agent performs an action $p(s_t) = a_t \in A$ at state s_t using the *policy* $p : S \to A$, which is a mapping from states to actions. Then, it moves to the next state s_{t+1} depending on the action a_t and receives a real-value reward $r(s_t, a_t, s_{t+1})$ via a black-box reward function $r : S \times A \times S \to \mathbb{R}$. The agent repeatedly performs actions until reaching the final state. The key concept of RL is to learn a policy that maximizes a cumulative reward for its actions. Instead of directly learning the policy, Q-learning [108] learns the quality function (Q-function) over state-action pairs. Using a long-term cumulative reward Q-function $Q : S \times A \to \mathbb{R}$, the policy function p is defined as follows:

$$p(s) = \underset{a \in A}{\operatorname{argmax}}\, Q(s, a).$$

Because explicitly computing Q-function is infeasible, Q-learning learns its approximation.

Singh et al. [83] leverage RL with Q-learning to find the best choices of a threshold, a splitting method, and a merge algorithm of ∇ in interval domain. A static analyzer becomes an agent, their actions are analysis parameters for ∇, and a reward function represents the precision and performance of the join operation. Heo et al. [34] present RL-based static analysis inspired by batch-mode reinforcement learning [20] to consider resource constraints during static analysis with interval domain. They reduce the number of selected variables for flow-sensitive views by continuous learning with analysis results.

4.4.4 Learnt Probabilistic Model. Grigore and Yang [23] propose a technique to utilize learnt probabilistic model to an abstraction refinement algorithm introduced by Zhang et al. [119]. They consider facts and rules in Datalog as nodes and edges of hypergraphs and define a probabilistic model that predicts how a hypergraph would change if a new and more precise analysis parameters are used. The probabilistic model is a variant of the Erdős-Rényi random graph model [19] defined on hypergraphs.

5 OPEN CHALLENGES AND FUTURE RESEARCH DIRECTIONS

Parameterizing Abstract Values. We introduced research on how to parameterize abstract states for parametric static analysis in Section 3.2. While they focus on configuring abstraction of memories such as variables, addresses, and fields, no work addresses parameterizing abstraction of values such as numbers or strings. A possible approach is to use different abstract number domains for different variables to adjust the balance between performance and precision. While it still has a challenge that the abstract semantics of operators for abstract values in different domains is not well defined yet, we believe that it is a promising research direction of parametric static analysis.

Using Different Opaque Function Modeling. As we explained in Section 3.4.2, several researchers have modified view transitions of opaque functions by automatically modeling their semantics in a soundy way. While all of them focus on how to automatically model their behaviors, no research studies use of different modeling depending on given programs. For example, consider three different modeling for JavaScript built-in functions: (1) type-guided automatic modeling [6, 70] has high performance but low precision; (2) ML-based modeling [36] has high precision but low performance; and (3) partial modeling using concrete execution [105] has both high precision and performance but it is only partially applicable. Instead of using only one modeling, using different modeling for different conditions may produce better results. When the third modeling is possible, it is the best option to choose. In other cases, we can select one of the first and second modeling while considering the balance between performance and precision.

Advanced Machine Learning-based Parametric Static Analysis. Recent studies on parametric static analysis take advantage of machine learning techniques in parameter selection, depending on given program properties and feedback from previous or intermediate analysis results, as explained in Section 4.4. Many of them perform analysis for a huge number of training data with various configuration of analysis parameters to identify relationships between given program properties (or feedback) and the best choice of analysis parameters. Thus, they are time-consuming to handle such big training data. Future research directions may include developing more efficient machine learning algorithms to reduce learning cost. Moreover, most of existing ML-based parametric static analysis exploits only restricted properties such as syntactic properties and sizes of points-to sets. Fortunately, recent studies on machine learning for programs [2, 3, 32] provide helpful directions to alleviate this problems. Allamanis et al. [2] present a technique to construct graphs from source code, Henkel et al. [32] use abstract symbolic traces of given programs as a representation for learning word embeddings, and Alon et al. [3] represent code snippets as single fixed-length code vectors. We believe that such learning-based code embedding approaches can enhance the quality of parameter selection for parametric static analysis.

6 CONCLUSION

We introduce research on parametric static analysis, which parameterizes static analysis to improve analysis quality. According to the parameterized components of static analysis, we classify the literature in four analysis parameters: analysis sensitivity, state abstraction, initial abstract state, and abstract transfer function. Parametric static analysis considers a static analyzer as a black-box and focuses on finding the best combination of analysis parameters in parameter selection. It utilizes the extracted program properties or feedback from previous or intermediate analysis results in the parameter selection process. Although parametric static analysis has addressed various challenges in balancing three metrics of the static analysis quality, we observed that there still remain several open challenges: (1) parameterizing abstract values, (2) using different opaque function modeling, and (3) advanced machine learning-based parametric static analysis.

We believe that parametric static analysis opens the gate to interesting research topics for future research directions.

REFERENCES

[1] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. 2002. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *Proceedings of the International Static Analysis Symposium*. 230–246.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. (2018).

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 3. 1–29.

[4] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jquery. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. 17–31.

[5] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I. Jordan. 2003. An introduction to MCMC for machine learning. *Mach. Learn.* 50, 1–2 (2003), 5–43.

[6] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API misuse detector for web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 507–517.

[7] Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. 2019. Eventually sound points-to analysis with specifications. In *Proceedings of the European Conference on Object-Oriented Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[8] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a static analyzer from data. In *Proceedings of the International Conference on Computer-aided Verification*. Springer, 233–253.

[9] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise refutations for heap reachability. In *Proceedings of the Conference on Programming Language Design and Implementation*, Vol. 48. 275–286.

[10] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the Conference on Programming Language Design and Implementation*. 196–207. DOI: https://doi.org/10.1145/781131.781153

[11] Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2016. Learning a strategy for choosing widening thresholds from a large codebase. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. 25–41.

[12] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically generating features for learning program analysis heuristics for c-like languages. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 1. 25. DOI: https://doi.org/10.1145/3133925

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*. 238–252. DOI: https://doi.org/10.1145/512950.512973

[14] John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps. 2019. Refinement of path expressions for static analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 3. 1–29.

[15] Arnab De and Deepak D'Souza. 2012. Scalable flow-sensitive pointer analysis for java with strong updates. In *Proceedings of the European Conference on Object-oriented Programming*. 665–687. DOI: https://doi.org/10.1007/978-3-642-31057-7_29

[16] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-scale exhaustive points-to analysis for Java in under a minute. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. 535–551. DOI: https://doi.org/10.1145/2858965.2814307

[17] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the International Symposium on Software Testing and Analysis*. 118–128. DOI: https://doi.org/10.1145/1273463.1273480

[18] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2008. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 59–70. DOI: https://doi.org/10.1145/1453101.1453111

[19] Paul Erdős and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* 5, 1 (1960), 17–60.

[20] Damien Ernst, Pierre Geurts, and Louis Wehenkel. 2005. Tree-based batch mode reinforcement learning. *J. Mach. Learn. Res.* 6, Apr. (2005), 503–556.

[21] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the International Conference on Software Engineering*. IEEE, 72–82.

[22] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps don't lie: Countering unsoundness with heap snapshots. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 1. 1–27.

[23] Radu Grigore and Hongseok Yang. 2016. Abstraction refinement guided by a learnt probabilistic model. In *Proceedings of the Symposium on Principles of Programming Languages*. 485–498. DOI: https://doi.org/10.1145/2837614.2837663

[24] Bhargav S. Gulavani and Sriram K. Rajamani. 2006. Counterexample driven refinement for abstract interpretation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 474–488. DOI: https://doi.org/10.1007/11691372_34

[25] Samuel Z. Guyer and Calvin Lin. 2003. Client-driven pointer analysis. In *Proceedings of the International Static Analysis Symposium*. 214–236.

[26] Samuel Z. Guyer and Calvin Lin. 2005. Error checking with client-driven pointer analysis. *Sci. Comput. Prog.* 58, 1–2 (2005), 83–114.

[27] Ben Hardekopf and Calvin Lin. 2007. Exploiting pointer and location equivalence to optimize pointer analysis. In *Proceedings of the International Static Analysis Symposium*. 265–280.

[28] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the International Symposium on Code Generation and Optimization*. 289–298.

[29] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the International Workshop on State of the Art in Program Analysis*. 13–18. DOI: https://doi.org/10.1145/3088515.3088519

[30] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2020. Learning fast and precise numerical analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. 1112–1127.

[31] Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. 24–34. DOI: https://doi.org/10.1145/378795.378802

[32] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 163–174.

[33] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In *Proceedings of the International Static Analysis Symposium*. 237–256.

[34] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware program analysis via online abstraction coarsening. In *Proceedings of the International Conference on Software Engineering*. IEEE, 94–104.

[35] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the International Conference on Software Engineering*. 519–529. DOI: https://doi.org/10.1109/ICSE.2017.54

[36] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing models for opaque code. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 710–720.

[37] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis*. 34–44. DOI: http://doi.acm.org/10.1145/2338965.2336758

[38] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 2. 1–29.

[39] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 1. 28. DOI: https://doi.org/10.1145/3133924

[40] Vineet Kahlon. 2008. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. 249–259. DOI: https://doi.org/10.1145/1375581.1375613

[41] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 423–434.

[42] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. 2018. A theoretical foundation of sensitivity in an abstract interpretation framework. *ACM Trans. Prog. Lang. Syst.* 40, 3 (2018), 1–44.

[43] Thomas N. Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[44] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically tunable static analysis framework for large-scale JavaScript applications. In *Proceedings of the International Conference on Automated Software Engineering*. 541–551. DOI: https://doi.org/10.1109/ASE.2015.28

[45] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2017. Weakly sensitive analysis for unbounded iteration over JavaScript objects. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 148–168.

[46] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2019. Weakly sensitive analysis for JavaScript object-manipulating programs. *Softw.: Pract. Exper.* 49, 5 (2019), 840–884.
[47] William Landi and Barbara G. Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the Conference on Programming Language Design and Implementation*, Vol. 27. 235–248.
[48] James R. Larus and Paul N. Hilfinger. 1988. Detecting conflicts between structure accesses. In *Proceedings of the Conference on Programming Language Design and Implementation*, Vol. 23. 24–31.
[49] Vincent Laviron and Francesco Logozzo. 2009. Refining abstract interpretation-based static analyses with hints. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. 343–358.
[50] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 2. 29. DOI: https://doi.org/10.1145/3276511
[51] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 129–140.
[52] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the Conference on Programming Language Design and Implementation*. 780–793.
[53] Percy Liang and Mayur Naik. 2011. Scaling abstraction refinement via pruning. In *Proceedings of the Conference on Programming Language Design and Implementation*. 590–601. DOI: https://doi.org/10.1145/1993316.1993567
[54] Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning minimal abstractions. In *Proceedings of the Symposium on Principles of Programming Languages*. 31–42. DOI: https://doi.org/10.1145/1926385.1926391
[55] 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 (Feb. 2015), 44–46. DOI: https://doi.org/10.1145/2644805
[56] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*.
[57] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 499–509. DOI: https://doi.org/10.1145/2491411.2491417
[58] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. DOI: https://doi.org/10.1145/1044834.1044835
[59] Antoine Miné. 2006. The octagon abstract domain. *Higher-order Symb. Computat.* 19, 1 (2006), 31–100.
[60] Kevin P Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. The MIT Press.
[61] Glenford J. Myers. 1976. *The Art of Software Testing*.
[62] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from tests. In *Proceedings of the Symposium on Principles of Programming Languages*. 373–386. DOI: https://doi.org/10.1145/2103656.2103701
[63] Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. 2011. Access analysis-based tight localization of abstract memories. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*. 356–370.
[64] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the Conference on Programming Language Design and Implementation*. 229–238. DOI: https://doi.org/10.1145/2254064.2254092
[65] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. 475–484. DOI: https://doi.org/10.1145/2594291.2594318
[66] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2015. Selective x-sensitive analysis guided by impact pre-analysis. *ACM Trans. Prog. Lang. Syst.* 38, 2 (2015), 1–45.
[67] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via Bayesian optimisation. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. 572–588. DOI: https://doi.org/10.1145/2814270.2814309
[68] Hakjoo Oh and Kwangkeun Yi. 2011. Access-based localization with bypassing. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. 50–65. DOI: https://doi.org/10.1007/978-3-642-25318-8_7
[69] Changhee Park and Sukyoung Ryu. 2015. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proceedings of the European Conference on Object-oriented Programming*. 735–756.
[70] Jihyeok Park. 2014. JavaScript API misuse detection by using typescript. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity (MODULARITY'14)*. 11–12. DOI: https://doi.org/10.1145/2584469.2584472
[71] Joonyoung Park, Alexander Jordan, and Sukyoung Ryu. 2019. Automatic modeling of opaque code for JavaScript static analysis. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, 43–60.

[72] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with false positives in static analysis of JavaScript web applications in the wild. In *Proceedings of the International Conference on Software Engineering Companion*. 61–70.
[73] J. Ross Quinlan. 1986. Induction of decision trees. *Mach. Learn.* 1, 1 (1986), 81–106.
[74] J Ross Quinlan. 2014. *C4.5: Programs for Machine Learning*.
[75] Girish Maskeri Rama, Raghavan Komondoor, and Himanshu Sharma. 2018. Refinement in object-sensitivity points-to analysis via slicing. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 2. 27. DOI: https://doi.org/10.1145/3276512
[76] Ramanathan Ramu, Ganesha B. Upadhyaya, Hoan Anh Nguyen, and Hridesh Rajan. 2020. BCFA: Bespoke control flow analysis for CFA at scale. In *Proceedings of the International Conference on Software Engineering*. IEEE.
[77] Thomas Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11 (Dec. 1998), 701–726. DOI: https://doi.org/10.1016/S0950-5849(98)00093-7
[78] Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. 47–56. DOI: https://doi.org/10.1145/349299.349310
[79] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. 165–174. DOI: https://doi.org/10.1145/2491956.2462168
[80] Bernhard Schölkopf, John C. Platt, John Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. 2001. Estimating the support of a high-dimensional distribution. *Neural Computat.* 13, 7 (2001), 1443–1471.
[81] Micha Sharir and Amir Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.
[82] 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 the Conference on Programming Language Design and Implementation*. 693–706.
[83] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast numerical program analysis with reinforcement learning. In *Proceedings of the International Conference on Computer-aided Verification*. Springer, 211–229.
[84] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based pre-processing for points-to analysis. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. 253–270. DOI: https://doi.org/10.1145/2509136.2509524
[85] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the Conference on Programming Language Design and Implementation*. 485–495.
[86] Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static analysis for asynchronous JavaScript programs. In *Proceedings of the European Conference on Object-oriented Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[87] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 3. 1–29.
[88] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *Proceedings of the European Conference on Object-oriented Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[89] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*. 387–400. DOI: https://doi.org/10.1145/1133981.1134027
[90] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-oriented Programming*. Springer, 435–458.
[91] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. 59–76. DOI: https://doi.org/10.1145/1094811.1094817
[92] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. 32–41. DOI: https://doi.org/10.1145/237721.237727
[93] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static analysis with demand-driven value refinement. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 3. 1–29.
[94] Yulei Sui, Yue Li, and Jingling Xue. 2013. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the International Symposium on Code Generation and Optimization*. 1–11. DOI: https://doi.org/10.1109/CGO.2013.6494978
[95] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 460–473. DOI: https://doi.org/10.1145/2950290.2950296

[96] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*. 254–264. DOI: https://doi.org/10.1145/2338965.2336784

[97] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Softw. Eng.* 40, 2 (2014), 107–122.

[98] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. 155–171. DOI: https://doi.org/10.1007/978-3-642-25318-8_14

[99] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. The MIT Press.

[100] T. A. Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Syst. J.* (1989), 294–306.

[101] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *Proceedings of the International Static Analysis Symposium*. 489–510. DOI: https://doi.org/10.1007/978-3-662-53413-7_24

[102] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In *Proceedings of the Conference on Programming Language Design and Implementation*. 278–291.

[103] Manas Thakur and V. Krishna Nandivada. 2019. Compare less, defer more: Scaling value-contexts based whole-program heap analyses. In *Proceedings of the 28th International Conference on Compiler Construction*. 135–146.

[104] Manas Thakur and V. Krishna Nandivada. 2020. Mix your contexts well: Opportunities unleashed by recent advances in scaling context-sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction*. 27–38.

[105] John Toman and Dan Grossman. 2019. Concerto: A framework for combined concrete and abstract interpretation. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 3. 1–29.

[106] Jyothi Vedurada and V. Krishna Nandivada. 2019. Batch alias analysis. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 936–948.

[107] V. Vipindeep and Pankaj Jalote. 2005. Efficient static analysis with path pruning using coverage data. In *Proceedings of the International Workshop on Dynamic Analysis*. 1–6. DOI: https://doi.org/10.1145/1082983.1083257

[108] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Mach. Learn.* 8, 3–4 (1992), 279–292.

[109] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 3. 1–32.

[110] Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis*. 336–346. DOI: https://doi.org/10.1145/2483760.2483788

[111] Shiyi Wei and Barbara G. Ryder. 2014. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of the European Conference on Object-oriented Programming*. 1–26.

[112] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *Proceedings of the European Conference on Object-oriented Programming*. 712–734. DOI: https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

[113] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 487–498. DOI: https://doi.org/10.1145/2950290.2950338

[114] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. 131–144. DOI: https://doi.org/10.1145/996841.996859

[115] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the European Conference on Object-oriented Programming*. 98–122. DOI: https://doi.org/10.1007/978-3-642-03013-0_6

[116] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the International Symposium on Code Generation and Optimization*. 218–229. DOI: https://doi.org/10.1145/1772954.1772985

[117] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for Java API functions. In *Proceedings of the International Conference on Software Engineering*. IEEE, 380–391.

[118] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the Symposium on Principles of Programming Languages*. 344–358.

[119] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in datalog. In *Proceedings of the Conference on Programming Language Design and Implementation*. 239–248. DOI: https://doi.org/10.1145/2594291.2594327

[120] Xin Zhang, Mayur Naik, and Hongseok Yang. 2013. Finding optimum abstractions in parametric dataflow analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. 365–376. DOI: https://doi.org/10.1145/2491956.2462185

[121] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: Practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 3. 1–31.

Received October 2019; revised March 2021; accepted April 2021