Learning to Boost Disjunctive Static Bug-Finders

Yoonseok Ko
Meta
ysko@meta.com

Hakjoo Oh
Korea University
hakjoo_oh@korea.ac.kr

***Abstract*—We present a new learning-based approach for accelerating disjunctive static bug-finders. Industrial static bug-finders usually perform disjunctive analysis, differentiating program states along different execution paths of a program. Such path-sensitivity is essential for reducing false positives but it also increases analysis costs exponentially. Therefore, practical bug-finders use a state-selection heuristic to keep track of a small number of beneficial states only. However, designing a good heuristic for real-world programs is challenging; as a result, modern static bug-finders still suffer from low cost/bug-finding efficiency. In this paper, we aim to address this problem by learning effective state-selection heuristics from data. To this end, we present a novel data-driven technique that efficiently collects alarm-triggering traces, learns multiple candidate models, and adaptively chooses the best model tailored for each target program. We evaluate our approach with Infer and show that our technique significantly improves Infer's bug-finding efficiency for a range of open-source C programs.**

I. Introduction

Static bug-finders are increasingly used in industry [1], [2]. Infer [1], for example, is probably the most well-known bug-finding static analyzer based on academic research [3], which has been used at Meta to catch important issues such as memory safety errors in C/C++ codebases [4] and data races in Java programs [5]. Other large software companies such as Apple [6], Google [2], [7], and Microsoft [8] are also actively building static analysis tools in order to catch latent bugs early in their software development process [7], [9].

Bug-finding static analyzers usually perform disjunctive analysis [4], [6], [10]–[12]. A disjunctive static analyzer maintains a set of abstract states, each state representing an analysis result along a different execution path of a program. This path-sensitivity is an essential element of practical bug finding. It is well-known that path-sensitivity is critical for reducing false positives [13]–[16]. Also it is important for understanding and debugging reported bugs; without path-sensitivity, it is hardly possible to explain how bugs arise in terms of concrete execution paths.

However, modern disjunctive analyzers still suffer from low bug-finding efficiency. Since analyzing a non-trivial program in a fully path-sensitive manner is infeasible, practical bug-finders use various heuristics to balance the performance. A well-known method is to limit the number of disjuncts to be maintained (called "dropping disjuncts" in Incorrectness Logic [17]), which prunes states so that the number of states is always kept smaller than a given threshold. While such a state-selection heuristic is critical for the analysis performance, developing a good one still remains a challenge; as a result, even state-of-the-art static analyzers show unsatisfactory performance. For example, Infer employs a heuristic that takes the threshold value, denoted K, as an external parameter, allowing to control the trade-off between analysis coverage and cost. However, increasing K to detect more bugs quickly makes the analysis prohibitively expensive (Table I). Therefore, Infer is typically used with small K values in practice, significantly compromising analysis coverage to maintain scalability.

This Work. In this paper, we aim to improve the bug-finding efficiency of disjunctive static bug-finders, thereby allowing an existing analyzer to detect more bugs with the same or even smaller K values (hence within shorter analysis times).

We achieve this goal by learning a state-selection heuristic from data. The basic approach consists of two steps. Given a static analyzer and a set of training programs, we first augment the analyzer to collect alarm-triggering traces and learn a statistical model that predicts how likely states are involved in those traces. Our state-selection heuristic then uses the trained model to rank and select top-K states. However, this basic method faces two technical challenges: (1) naively using the augmented analysis is unlikely to collect diverse alarm-triggering traces, and (2) even with sufficient data, it is unlikely that "one-size-fits-all" models exist for a wide range of real programs. We address these challenges by developing an efficient trace-collection algorithm that learns to explore likely alarm-triggering traces, and a dynamic model selection technique that trains multiple models and adaptively chooses the best one tailored for each target program via pre-analysis.

Experimental results show that our technique can substantially improve the performance of industrial static analyzers. We applied our approach to Infer, replacing its original state-selection heuristic by ours learned from a dataset of 70 open-source programs. We then compared the performance of our data-driven Infer with the original Infer on 15 unseen programs (57–701 KLoC). The original Infer produced 1,415 alarms in 5,444 seconds with the default configuration $K = 20$, and 1,637 alarms in 39,684 seconds with $K = 60$. Our data-driven Infer, on the other hand, was able to report 1,668 alarms even with $K = 5$ and took 875 seconds, accelerating the original Infer with $K = 20$ and $K = 60$ by 6x and 45x, respectively.

Contributions. Our contributions are summarized as follows:

- We present a new method for learning state-selection heuristics for disjunctive static bug-finders. Key technical contributions are efficient trace-collection and adaptive model selection.

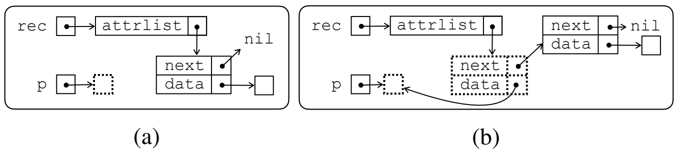

Fig. 1: Memory states after the call (a) `sdp_seq_alloc` and (b) `sdp_attr_replace`. The dotted boxes denote newly allocated memory blocks.

- We evaluate our approach with Infer, an industrial bug-finding static analyzer deployed within Meta, and show that its efficiency can be improved significantly.
- For open science, we make our source code publicly available.

II. Overview

A. Motivating Example

Bug Detection Example. With our technique, Infer successfully detected a memory-leak bug from open-source project `bluez-5.55`. The alarm report produced by Infer is given below:

```
Memory dynamically allocated at line 2007 by call
to sdp_seq_alloc(), is not freed after the last
access at line 2008, column 3.
```

The (simplified) code for lines 2007 and 2008 is as follows:

```
p = sdp_seq_alloc(...);          // line 2007
sdp_attr_replace(rec, ..., p);   // line 2008
```

where the pointer variable `rec` is an argument of the enclosing function, and it refers to a memory block accessible from the outside. The local variable `p` temporarily stores the result of `sdp_seq_alloc`, is passed to `sdp_attr_replace` as an argument, and is not used anymore after line 2008.

Executing the two lines above involves 14 different function calls, but for the sake of brevity, let us consider the following core executions related to the memory leak:

- When the function `sdp_seq_alloc` is called, it allocates a new memory block and returns the address of the block. This address is then assigned to the local variable `p` (Figure **??**).
- When the function `sdp_attr_replace` is called, it allocates a new linked-list memory block. It then assigns the address of the memory block pointed to by `p` to the `data` field of the newly allocated linked-list block, and inserts the block into the linked list designated by `rec->attrlist` (Figure **??**).

When `sdp_attr_replace` fails to allocate a memory block, the program state remains as Figure **??** and hence results in a memory leak. the local variable `p` is not used later and its referenced memory block is not freed.

Difficulty of Finding the Bug. To detect this bug, Infer needs to employ a carefully-tuned state-selection heuristic. Analyzing the above two lines of code path-sensitively produces more than two million distinct cases. The execution of the function `sdp_attr_replace` involves 8 different function calls, each of which has multiple branches; analyzing a series of function calls results in a combinatorial explosion of the number of possible execution cases, yielding more than 2,000 disjunctive summaries for the function `sdp_attr_replace`. Similarly, the analysis produces more than 1,000 summaries for the function `sdp_seq_alloc`. Among those two million $(2,000 \times 1,000)$ execution paths, only 7% of them are buggy paths that produce a memory leak. Our data-driven Infer could accurately prioritize these paths and succeeded to report the bug while the original Infer failed to do so.

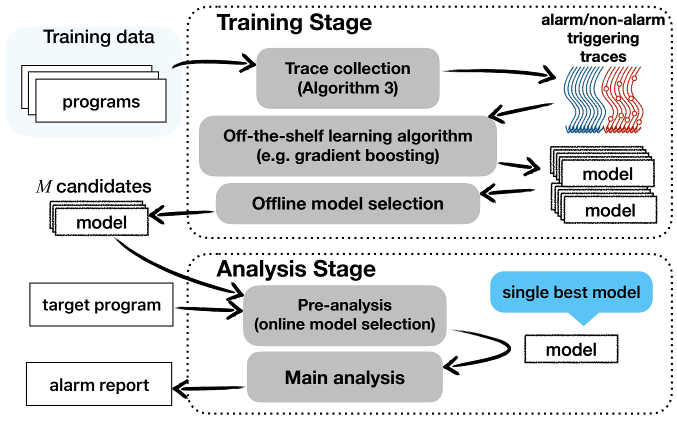

Fig. 2: Overview of our approach

B. Overall Workflow of Our Approach

Figure 2 provides an overview of our approach, which is divided into two main stages: the training stage and the analysis stage. The training stage generates a collection of models for state-selection heuristics. First, it collects alarm and non-alarm triggering traces from programs in the training set. For this step, we use an extended analysis producing a set of traces instead of summaries. Then, an off-the-shelf learning algorithm trains a collection of models that predict the probability of a state contributing to an alarm from the collected traces. Finally, we select M models that collectively perform the best in the validation set. The analysis stage first selects the best model for each target program from the M models, where we use a lightweight pre-analysis with a limited K_{pre} to identify the model that best fits the target program. Then, we proceed with the main analysis using K_{main} with the selected model.

III. Preliminaries

A. Disjunctive Bug-Finding Static Analysis

In this section, we describe a generic algorithm for disjunctive static bug-finders, which forms the core of industrial static analyzers such as Infer [1]. We note that soundness is not a primary concern of such an analyzer; instead, the analysis is designed with a focus on scalability and precision. For scalability, the analysis is designed to be summary-based and works in a bottom-up fashion [1], [10]. For precision, it is designed to be disjunctive, among others, separately maintaining program states along different execution paths.

Algorithm 1 Bottom-up analysis algorithm

```
Require: $P$: program, $\mathcal{H}^K$: state-selection heuristic
Ensure: $\mathcal{A}$: alarms (a set of potential errors)
1: procedure Analyzer($P, \mathcal{H}^K$)
2:     $Funcs \leftarrow$ functions of $P$ in reverse topological order
3:     $\mathcal{T} \leftarrow \lambda f.\emptyset$    ▷ initial function summaries
4:     $\mathcal{A} \leftarrow \emptyset$    ▷ initial alarms
5:     for each $f$ in $Funcs$ do    ▷ do bottom-up analysis
6:         $\langle S, A\rangle \leftarrow$ analyze_function($f, \mathcal{T}, \mathcal{H}^K$)
7:         $\mathcal{T} \leftarrow \mathcal{T}[f \mapsto S]$    ▷ update summaries
8:         $\mathcal{A} \leftarrow \mathcal{A} \cup A$    ▷ accumulate alarms
9:     return $\mathcal{A}$
```

Overall algorithm. The bottom-up analysis first analyzes leaf functions in the program and summarizes their behaviors. Functions that call leaf functions are then analyzed, where called functions are not re-analyzed but their summaries are instantiated at call-sites context-sensitively. This way, functions in the program are analyzed in the reverse topological order of the call graph. For simplicity, we do not consider recursive call cycles; in practice, recursion is ignored or summaries of the functions involved in a recursive call cycle are computed simultaneously via iterative fixpoint computation.

Algorithm 1 describes the bottom-up analysis algorithm. It takes as input a program to analyze (we ignore the second input, $\mathcal{H}^K$, for the moment) and as output it produces a set of alarms, e.g., potential memory leak errors. *Funcs* at line 2 stores the list of functions sorted in the reverse topological order of the call graph. At line 3, we initialize function summaries; $\mathcal{T}$ is a map from functions to (initially empty) summaries. $\mathcal{A}$ at line 4 denotes the set of alarms produced by the analysis, which is initially set to $\emptyset$. At line 5, the analysis considers each function f in *Funcs*. At line 6, function f is analyzed in isolation using the current summaries $\mathcal{T}$, where $\mathsf{analyze\_function}(f, \mathcal{T}, \mathcal{H}^K)$ is responsible for computing the function summary (S) for f and alarms A obtained from analyzing the body of f. At lines 7 and 8, summaries and alarms are stored in $\mathcal{T}$ and $\mathcal{A}$, respectively. Once all functions in *Funcs* are analyzed, the algorithm returns the set $\mathcal{A}$ of alarms accumulated so far.

Analysis of a function. Now we describe how a function is analyzed, i.e., analyze_function. Suppose the body of function f is given by a control-flow graph $\mathcal{G} = (N, \hookrightarrow, exit)$, where N is the set of nodes, $(\hookrightarrow) \subseteq N \times N$ a set of edges, and $exit \in N$ the (unique) exit node of the function. We assume each node n is associated with a command, denoted $\mathsf{cmd}(n)$. Commands include assignments, branches, and function calls:

$$c ::= x \;\texttt{:=}\; e \mid \texttt{assume}(x \;\texttt{=}\; b) \mid f\texttt{()} \mid \ldots$$

where e denotes an expression and b is a boolean constant. For simplicity, we only allow a boolean value in `assume` statements and do not consider function arguments.

A program state $s \in \mathbb{S} = PC \times \mathbb{M}$ is a pair of path condition (PC) and memory state ($\mathbb{M}$). A path condition $\pi \in PC$ is a conjunction of predicates, e.g., $x = \texttt{false} \wedge y = \texttt{true}$, which keeps track of the branch conditions taken by the current path being analyzed. A memory state $m \in \mathbb{M}$ maps (abstract) locations to (abstract) values, which we leave unspecified as the definition of memory states varies depending on the purpose of the analysis. For example, when the analysis aims to detect memory leaks, it would keep information about memory allocation and deallocation while discarding other aspects (e.g., numerical values) of the program.

The goal of analyze_function is to compute a set $S \subseteq \mathbb{S}$ of reachable program states at the exit node of f, one for each different path in f, as well as alarms representing potential errors that may occur in f. To compute reachable states (S), we analyze the function flow-sensitively. That is, we compute a table $X : N \to \wp(\mathbb{S})$ from program points to sets of output states, which is defined as a fixpoint, $\mathit{fix}F$, of the semantic function $F : (N \to \wp(\mathbb{S})) \to (N \to \wp(\mathbb{S}))$:

$$F(X) \overset{\mathsf{def}}{=} \lambda n \in N.\, \mathcal{H}^K\Big(\bigcup_{n' \hookrightarrow n} \bigcup_{s \in X(n')} [\![\mathsf{cmd}(n)]\!](s)\Big) \qquad (1)$$

where we assume $\mathcal{H}^K$ is the identity function at the moment (we explain $\mathcal{H}^K$ shortly in Section III-B). To analyze a node $n \in N$, the analysis takes its predecessor n' and its output state s, and uses the transfer function, denoted $[\![\mathsf{cmd}(n)]\!]$, to transform s into the output state of n. The transfer function $[\![\mathsf{cmd}(n)]\!] : \mathbb{S} \to \wp(\mathbb{S})$ is defined for each statement type. The semantics of assignment is to make a side-effect in the memory state:

$$[\![x \;\texttt{=}\; e]\!](\langle \pi, m\rangle) \overset{\mathsf{def}}{=} \{\langle \pi, \mathsf{assign}_{\mathbb{M}}(x, e, m)\rangle\}$$

for which we assume the domain of memory states comes with function $\mathsf{assign}_{\mathbb{M}}(x, e, m) \in \mathbb{M}$ that replaces the value of x in m by the value of e. At assume statements, the path condition π is updated by conjoining π with the branch condition if the assumption is satisfied:

$$[\![\texttt{assume}(x \;\texttt{=}\; b)]\!](\langle \pi, m\rangle) \overset{\mathsf{def}}{=} \{\langle \pi \wedge (x \;=\; b), m\rangle\}.$$

Function calls are analyzed using summaries:

$$[\![f\texttt{()}]\!](\langle \pi, m\rangle) \overset{\mathsf{def}}{=} \{\langle \pi \wedge \pi', \mathsf{instantiate}_{\mathbb{M}}(m, m')\rangle \mid \langle \pi', m'\rangle \in \mathcal{T}(f)\}.$$

We first look up the summary table, i.e., $\mathcal{T}(f)$, to obtain the output states of f. For each output state $\langle \pi', m'\rangle$, we produce a new state $\langle \pi \wedge \pi', \mathsf{instantiate}_{\mathbb{M}}(m, m')\rangle$ by merging path conditions and *instantiating* the output memory m' appropriately for the current call context (m). For this, we assume function $\mathsf{instantiate}_{\mathbb{M}}$ is given together with the domain of memory state ($\mathbb{M}$), so that we can design a range of bottom-up analyses by varying the definitions of $\mathbb{M}$, $\mathsf{assign}_{\mathbb{M}}$, and $\mathsf{instantiate}_{\mathbb{M}}$.

To generate alarms, static bug-finders use their own alarm-producing criteria. In this paper, we assume the function alarms of the following type is provided by the analyzer:

$$\mathsf{alarms} : Cmd \times \mathbb{S} \to \wp(\mathbb{A}) \qquad (2)$$

where $\mathbb{A}$ denotes the set of all alarms possibly generated from the program. Function alarms takes a command and a program state, and produces a set of alarms denoting errors that may occur at the command. For example, when the analysis aims to detect potential null dereferences, alarms produces alarms when the command involves pointer dereference $*p$ and p may point to null according to the input state.

We now define the result of $\mathsf{analyze\_function}(f, \mathcal{T}, \mathcal{H}^K)$ to be (S, A), where S is the set of states reachable at the exit of the function, $S \overset{\text{def}}{=} (\mathit{fix} F)(\mathit{exit})$, and A is the set of alarms generated from commands in f:

$$A \overset{\text{def}}{=} \bigcup_{n \in N} \bigcup_{n' \hookrightarrow n} \bigcup_{s \in (\mathit{fix} F)(n')} \mathsf{alarms}(\mathsf{cmd}(n), s).$$

B. State-Selection Heuristic

The analysis described in Section III-A is not yet practical because the number of summaries to maintain grows exponentially with the number of branches in the program. A well-known method to mitigate this issue is to use a state-selection heuristic [17] that compromises the soundness of the analysis in order to reduce costs. A state-selection heuristic, denoted $\mathcal{H}^K$, is a function that takes a set of program states and returns its subset of size K, where K is a pre-defined parameter: $\mathcal{H}^K : \wp(\mathbb{S}) \to \wp(\mathbb{S})$. For example, Infer uses $K = 20$ by default, which means that Infer computes up to 20 program states for each function.

Although the choice of state-selection heuristics critically affects the analysis performance, building an effective heuristic is nontrivial. As a result, even state-of-the-art static analyzers often rely on simple heuristics. For example, Infer uses a heuristic that selects top-K program states according to a lexicographic ordering of conjunctions of the path conditions. Depending on the syntactic structure of the program, it always delays the computation of particular branches, and eventually, program states essential for generating alarms are missed unless K is large enough to cover all the summaries from the remaining branches. Thus, Infer often fails to discover buggy program paths when K is small.

IV. Our Learning Approach

In this section, we present our approach for learning state-selection heuristics. In Section IV-A, we present a trace-collection algorithm that is used to generate labeled training data. In Section IV-B, we explain how to train models and select the best one tailored for each program. Throughout this section, we assume training and validation sets of programs are given, denoted $\vec{P}_{\mathit{train}} = \{P_{t_1}, P_{t_2}, \ldots\}$ and $\vec{P}_{\mathit{valid}} = \{P_{v_1}, P_{v_2}, \ldots\}$, respectively.

A. Trace Collection

To generate labeled training data, we use an augmented static analyzer to gather alarm-triggering and non-alarm-triggering traces from $\vec{P}_{\mathit{train}}$.

Notation. Given the set $\mathbb{S}$ of program states, we write $\mathbb{S}^*$ for the set of all finite sequences (traces) of states. Let ϵ be the empty sequence. Given a finite sequence $\sigma \in \mathbb{S}^*$ of states, we write σ_i for the ith state of σ, $\sigma_\dashv$ for the last state, and $|\sigma|$ for the length of the sequence, i.e., $\sigma_i = s_i$, $\sigma_\dashv = s_n$, and $|\sigma| = n$ when $\sigma = s_1 s_2 \cdots s_n$, and $|\epsilon| = 0$. Given a sequence $\sigma \in \mathbb{S}^*$ and a state $s \in \mathbb{S}$, we write $\sigma \cdot s$ for the sequence obtained by appending s to σ, i.e., $\sigma \cdot s = s_1 s_2 \cdots s_n s$ when $\sigma = s_1 s_2 \cdots s_n$. We also write $\sigma_1 \cdot \sigma_2$ for the concatenation of two sequences $\sigma_1, \sigma_2 \in \mathbb{S}^*$.

Algorithm 2 Trace-augmented analysis algorithm

Require: P: program, $\overline{\mathcal{H}}^K$: trace-selection heuristic
Ensure: Σ_{alm}: alarm-triggering traces, Σ_{all}: all analyzed traces

```
1: procedure $\overline{\mathsf{Analyzer}}(P, \overline{\mathcal{H}}^K)$
2:     $\mathit{Funcs} \leftarrow$ functions of $P$ in reverse topological order
3:     $\mathcal{T} \leftarrow \lambda f.\emptyset$                    ▷ initial function summaries
4:     $\Sigma_{\mathit{all}}, \Sigma_{\mathit{alm}} \leftarrow \emptyset, \emptyset$                    ▷ initial traces
5:     for each $f$ in $\mathit{Funcs}$ do                    ▷ do bottom-up analysis
6:         $\langle \Sigma_1, \Sigma_2 \rangle \leftarrow \overline{\mathsf{analyze\_function}}(f, \mathcal{T}, \overline{\mathcal{H}}^K)$
7:         $\mathcal{T} \leftarrow \mathcal{T}[f \mapsto \Sigma_1]$                    ▷ update summaries
8:         $\Sigma_{\mathit{all}} \leftarrow \Sigma_{\mathit{all}} \cup \Sigma_1$                    ▷ update summary traces
9:         $\Sigma_{\mathit{alm}} \leftarrow \Sigma_{\mathit{alm}} \cup \Sigma_2$                    ▷ update alarm traces
10:    return $\langle \Sigma_{\mathit{alm}}, \Sigma_{\mathit{all}} \rangle$
```

Given the set N of nodes, we write N^* for the set of all finite paths. We write 1-2-$\cdots$-x $\in N^*$ to denote a path from node 1, 2 to x where $1, 2, \cdots, \mathrm{x} \in N$. Given a sequence of states $\sigma \in \mathbb{S}^*$, we assume each state is associated with a node. We write $s^1 s^2 \cdots s^{\mathrm{x}}$ to denote the associated nodes, and the path of the sequence is 1-2-$\cdots$-x. We write $\mathsf{path}(\sigma)$ to denote the associated path of trace σ. For two paths $\phi_1, \phi_2 \in N^*$, we write ϕ_1-ϕ_2 for the concatenation of them. We write $\phi \dot{\leq} \phi'$ to denote ϕ is a non-strict prefix of ϕ'.

Trace-augmented analysis. To collect traces, we augment the static analysis in Algorithm 1 to deal with traces. The trace-augmented analysis in Algorithm 2 computes a set of traces as opposed to a set of states that the original analysis computes. As input, the analysis takes a program to analyze and a trace-selection heuristic, denoted $\overline{\mathcal{H}}^K$, which is a trace-lifted version of the state-selection heuristic $\mathcal{H}^K$:

$$\overline{\mathcal{H}}^K : \wp(\mathbb{S}^*) \to \wp(\mathbb{S}^*).$$

As output, the trace-augmented analysis computes two sets of traces, Σ_{alm} and Σ_{all}, where Σ_{alm} denotes the set of alarm-triggering traces and Σ_{all} denotes all analyzed traces. We say a trace σ is alarm-triggering if its last state $\sigma_\dashv$ triggers an alarm reporting, i.e., $\mathsf{alarms}(c, \sigma_\dashv) \neq \emptyset$ for the command c. If all the sub-traces of a trace σ is not alarm-triggering, σ is said to be non-alarm-triggering, i.e., $\forall 1 \leq i \leq |\sigma| : \mathsf{alarms}(c, \sigma_i) = \emptyset$ where c is the command that computed σ_i as an output.

The analysis iterates over the functions in the reverse topological order at lines 5–9. At line 6, each function f is isolation using the current summaries ($\mathcal{T}$), where a summary of a function is lifted to traces, i.e., $\mathcal{T} : \mathit{Funcs} \to \wp(\mathbb{S}^*)$. Analyzing a function produces as output Σ_1 and Σ_2 where Σ_1 denotes the trace-lifted summary for f (each last state of

Algorithm 3 Trace collection algorithm

```
Require: $\vec{P}$: programs, $K$: the number of states to select
Ensure: $\Sigma_{alm}$: alarm-triggering traces, $\Sigma_{non}$: non-alarm-triggering traces
 1: procedure CollectTraces($\vec{P}, K$)
 2:     $\Sigma_{alm}, \Sigma_{all} \leftarrow \emptyset, \emptyset$        $\triangleright$ alarm-triggering and all traces
 3:     repeat
 4:         for all $P \in \vec{P}$ do
 5:             $\overline{\mathcal{H}}^K_{exp} \leftarrow \mathsf{GenExpHeuristic}(\Sigma_{alm}, \Sigma_{all}, K)$
 6:             $\langle \Sigma'_{alm}, \Sigma'_{all} \rangle \leftarrow \overline{\mathsf{Analyzer}}(P, \overline{\mathcal{H}}^K_{exp})$
 7:             $\Sigma_{alm} \leftarrow \Sigma_{alm} \cup \Sigma'_{alm}$
 8:             $\Sigma_{all} \leftarrow \Sigma_{all} \cup \Sigma'_{all}$
 9:     until budget expires
10:     return $\langle \Sigma_{alm}, \Sigma_{all} \rangle$
```

which is a summary of f in Algorithm 1) and Σ_2 denotes the traces triggering alarms during the analysis of f. At line 7, the analysis updates the function summary traces. At lines 8–9, the analysis respectively accumulates the alarm-triggering traces and all traces. Once all functions are analyzed, the algorithm returns the alarm-triggering (Σ_{alm}) and the analyzed summary (Σ_{all}) traces at line 10.

Function $\overline{\mathsf{analyze\_function}}$, which corresponds to the lifted version of analyze_function in Algorithm 1, computes trace summaries for a function. Like the unaugmented version, it performs a flow-sensitive analysis to compute a fixpoint of a semantic function but the analysis results are traces, rather than states. That is, it computes a table $X : N \to \wp(\mathbb{S}^*)$ from nodes to a set of reachable traces, which is defined as a fixpoint of the function $\overline{F} : (N \to \wp(\mathbb{S}^*)) \to (N \to \wp(\mathbb{S}^*))$:

$$\overline{F}(X) \overset{\mathsf{def}}{=} \lambda n \in N.\, \overline{\mathcal{H}}^K\big(\bigcup_{n' \hookrightarrow n} \bigcup_{\sigma \in X(n')} \sigma \circ [\![\mathsf{cmd}(n)]\!](\sigma_{\dashv})\big)$$

where $(\circ) \in \mathbb{S}^* \times \wp(\mathbb{S}) \to \wp(\mathbb{S}^*)$ denotes the concatenation of a trace and a set of states, i.e., $\sigma \circ S \triangleq \{\sigma \cdot s \mid s \in S\}$. The analysis accumulates traces by concatenating the given trace σ and the state obtained from the execution of command with the last state $\sigma_{\dashv}$. Then, we define the output of $\overline{\mathsf{analyze\_function}}(f, \mathcal{T}, \overline{\mathcal{H}}^K)$ to be (Σ_1, Σ_2) where $\Sigma_1 = (\mathit{fix}\overline{F})(\mathit{exit})$ represents all the reachable traces at the end of the function and Σ_2 is the set of traces triggering alarms inside the function:

$$\Sigma_2 \overset{\mathsf{def}}{=} \bigcup_{n \in N} \bigcup_{n' \hookrightarrow n} \{\sigma \in (\mathit{fix}\overline{F})(n') \mid \mathsf{alarms}(\mathsf{cmd}(n), \sigma_{\dashv}) \neq \emptyset\}.$$

Naive trace-collection algorithm. A naive approach to collect traces is to repeatedly apply the trace-augmented analysis with a random trace-selection heuristic to the training programs $\vec{P}_{train}$. Let $\overline{\mathcal{H}}^K_{rand}$ be a random heuristic such that, given a set Σ of traces with $|\Sigma| > K$, $\overline{\mathcal{H}}^K_{rand}(\Sigma)$ returns a randomly-sampled subset $\Sigma' \subseteq \Sigma$ of size K. Then, we can iteratively invoke $\overline{\mathsf{Analyzer}}$ with programs in $\vec{P}_{train}$ and the random heuristic $\overline{\mathcal{H}}^K_{rand}$ to accumulate the resulting traces, Σ_{alm} and Σ_{all}, until a given time budget expires:

```
repeat
    for all $P \in \vec{P}_{train}$ do
        $\langle \Sigma_{alm}, \Sigma_{all} \rangle \leftarrow \overline{\mathsf{Analyzer}}(P, \overline{\mathcal{H}}^K_{rand})$
        accumulate $\Sigma_{alm}$ and $\Sigma_{all}$
until budget expires
```

This simple approach, however, is inefficient. Collecting a large number of alarm-triggering traces is a key to success in our approach, but relying on the random selection is unlikely to discover many different alarm-triggering cases.

Our algorithm. To improve the efficiency, our algorithm uses a so-called *exploratory trace-selection heuristic*, denoted $\overline{\mathcal{H}}^K_{exp} : \wp(\mathbb{S}^*) \to \wp(\mathbb{S}^*)$, to steer the search toward more profitable traces that are likely to trigger alarms. During the course of the algorithm, we continuously refine the heuristic based on the trace data collected, which in turn accelerates the trace-collection procedure and produces more data.

Algorithm 3 presents the workflow of our trace-collection algorithm. It repeatedly collects traces at lines 4–8 by running the trace-augmented analysis on the training programs until a given time budget expires. In each iteration, the exploratory heuristic $\overline{\mathcal{H}}^K_{exp}$ is reconstructed at line 5 (GenExpHeuristic) using the alarm-triggering traces Σ_{alm} and all collected traces Σ_{all}. The refined heuristic is then used by the analyzer ($\overline{\mathsf{Analyzer}}$) to find alarm-triggering traces more effectively. Below, we explain how we generate the heuristic $\overline{\mathcal{H}}^K_{exp}$ from Σ_{alm} and Σ_{all}.

The exploratory trace-selection heuristic $\overline{\mathcal{H}}_{exp}$ works in two steps. That is, it is defined by the composition of two sub-heuristics, namely repetition avoidance heuristic $\overline{\mathcal{H}}_{avoid}$ and adaptive selection heuristic $\overline{\mathcal{H}}^K_{adapt}$:

$$\overline{\mathcal{H}}^K_{exp}(\Sigma) = (\overline{\mathcal{H}}^K_{adapt} \circ \overline{\mathcal{H}}_{avoid})(\Sigma).$$

Given a set Σ of traces, we first use $\overline{\mathcal{H}}_{avoid}$ to discard traces in Σ that have already been considered in previous iterations. Next, we use $\overline{\mathcal{H}}^K_{adapt}$ to rank and select the top-K traces that are most likely to generate alarms.

Formally, the goal of the repetition avoidance heuristic ($\overline{\mathcal{H}}_{avoid}$) is to discard a trace σ in Σ if it is guaranteed that the current trace σ evolves to a complete trace σ' (denoted $\sigma \to_{\dashv} \sigma'$) that is already included in Σ_{all}:

$$\overline{\mathcal{H}}^{\Sigma_{all}}_{avoid}(\Sigma) \overset{\mathsf{def}}{=} \{\sigma \in \Sigma \mid \exists \sigma' \in \mathbb{S}^* : \sigma \to_{\dashv} \sigma' \Rightarrow \sigma' \notin \Sigma_{all}\}. \quad (3)$$

For a given trace $\sigma \in \Sigma$, to decide whether to discard it or not, note that we need to explore all the complete traces that follow σ, which is infeasible to compute efficiently.

Thus, we present an algorithm for approximately computing the set in (3). The idea is to maintain *covered pre-paths* Φ from the all analyzed traces. A path of a trace, denoted $\mathsf{path}(\sigma)$, is a covered pre-path of Σ_{all} if all its subsequent traces have been explored exhaustively. That is, for a given trace σ such that $\mathsf{path}(\sigma) \in \Phi$, $\forall \sigma' \in \mathbb{S}^* : \sigma \to_{\dashv} \sigma' \Rightarrow \sigma' \in \Sigma_{all}$ holds. We can express the definition (3) with Φ as follows:

$$\overline{\mathcal{H}}^{\Phi}_{avoid}(\Sigma) \overset{\mathsf{def}}{=} \{\sigma \in \Sigma \mid \mathsf{path}(\sigma) \notin \Phi\}.$$

Now we explain how to compute covered pre-paths from analyzed traces. For that, we need to know where we have yet to explore. Suppose, for instance, that there are no unexplored paths after a certain program node in the last analysis. It indicates that all subsequent traces after the program node have been exhaustively explored. In other words, a path from the start node to the program node is a covered pre-path.

We use annotated analyzed traces to find covered pre-paths. Let us mark a state with the underline, $\underline{s}$, to indicate where the analysis discarded a trace. The presence of this mark implies that an unexplored trace exists at the node. For instance, during the analysis, it discarded the trace $s^1 s^3 s^4$ and selected the trace $s^1 s^2 s^4$ at node 4, and it did not discard any trace at node 5. In this case, the annotated analysis result is $\{s^1 s^2 \underline{s}^4 s^5\}$.

From an annotated analyzed trace, because there is no unexplored path after the last annotated node, the path from the start node to the last annotated node is a covered pre-path and all subsequent paths are also covered pre-paths.

Example 1: Let $\Sigma_{all} = \{s^1 s^2 \underline{s}^4 s^5 \underline{s}^7 s^x\}$ be the all traces analyzed so far. The path 1-2-4-5-7 is a covered path because the analysis did not discard a trace at node x. However, 1-2-4 is not a covered path because there exists an unexplored trace at node 7 (as indicated by $\underline{s}^7$).

We use function cover : $\mathbb{S}^* \to \wp(N^*)$ that takes an annotated trace and computes covered pre-paths, and the function satisfies the following property:

$$\forall \phi \in N^* : \mathsf{pathLast}(\sigma) \dot{\leq} \phi \dot{\leq} \mathsf{path}(\sigma) \Rightarrow \phi \in \mathsf{cover}(\sigma).$$

where σ is an annotated trace, and $\mathsf{pathLast} : \mathbb{S}^* \to N^*$ takes an annotated trace and produces a path from the start node to the last annotated node. Finally, we compute the set of covered paths Φ from the annotated all analyzed traces Σ_{all} as follows:

$$\Phi = \bigcup_{\sigma \in \Sigma_{all}} \mathsf{cover}(\sigma)$$

Because cover does not consider other annotated traces, it computes a path that is not an actual covered path in terms of the whole analyzed result, and resulting heuristic may discard unexplored traces.

We extend the heuristic $\overline{\mathcal{H}}^K_{exp}$ and the trace-augmented analysis to create annotations on the all analyzed traces Σ_{all}. If the adaptive selection heuristic discards any trace during the analysis, the heuristic $\overline{\mathcal{H}}^K_{exp}$ marks the last state of the result. Note that the discarded traces by the repetition avoidance heuristic are not counted. In Algorithm 3, at line 6, the analysis produces the all analyzed traces Σ'_{all} with annotations.

The adaptive selection heuristic $\overline{\mathcal{H}}^K_{adapt}$ is simpler to define. Using the collected traces Σ_{alm} and Σ_{all}, it learns a statistical model that distinguishes between the alarm-triggering and non-alarm triggering traces. The heuristic is trained every iteration. The initial heuristic constructed with empty traces is equivalent to the random heuristic. As more traces are accumulated, however, the classifier is trained with more data and selects traces that are more likely to be alarm-triggering. Using the trained model, $\overline{\mathcal{H}}^K_{adapt}$ selects K likely alarm-triggering traces. We reuse the classifier learning algorithm and the trained classifier $C : \mathbb{S} \to [0, 1]$ in Section IV-B (explained shortly). The difference is that, for this on-the-fly heuristic construction, we use a specific hyper-parameter for the training algorithm, and do use the constructed model without model selection. The heuristic with a learned classifier C is defined as follows (when $|\Sigma| > K$; otherwise, $\overline{\mathcal{H}}^K_{adapt}(\Sigma) = \Sigma$):

$$\overline{\mathcal{H}}^K_{adapt}(\Sigma) \overset{\text{def}}{=} \underset{\Sigma' \subseteq \Sigma, |\Sigma'| = K}{\operatorname{argmax}} \sum_{\sigma_{\dashv} \in \Sigma'} C(\sigma_{\dashv})$$

The adaptive heuristic is similar to the state-selection heuristic $\mathcal{H}^K_C$, but it differs in that it computes ranking based on the last state ($\sigma_{\dashv}$) of the given traces.

Combining the two heuristics, $\overline{\mathcal{H}}_{avoid}$ and $\overline{\mathcal{H}}^K_{adapt}$, was essential for performance. For example, combining the repetition avoidance heuristic with the random selection heuristic is not effective because the random selection heuristic is inefficient on its own. Using the adaptive selection heuristic alone is definitely not good because the same traces are repeatedly selected at each iteration.

B. Training and Using Models

Once the traces Σ_{alm} and Σ_{all} are collected, we train statistical models that predict how likely a state belongs to an alarm-triggering trace. This learning procedure produces M models as output, where M is a user-provided parameter that controls the cost and coverage of our approach. When analyzing a new program, we choose the best model by running a pre-analysis M times with each model.

Generating training data. We generate training data by first computing non-alarm-triggering traces, denoted Σ_{non}, from Σ_{alm} and Σ_{all}, as follows:

$$\Sigma_{non} \overset{\text{def}}{=} \{\sigma \in \Sigma_{all} \mid \forall \sigma_1, \sigma_2 : \ \sigma = \sigma_1 \cdot \sigma_2 \Rightarrow \sigma_1 \notin \Sigma_{alm}\}$$

and then abstracting Σ_{alm} and Σ_{non} into alarm-contributing and non-alarm-contributing states, denoted S^+ and S^-, respectively. We say a state s is alarm-contributing if s appears in some alarm-triggering trace. Similarly, we say s is non-alarm-contributing if s does not appear in any alarm-triggering traces. S^+ and S^- are defined as follows:

$$S^+ = \alpha(\Sigma_{alm}), \qquad S^- = \alpha(\Sigma_{non}) \setminus \alpha(\Sigma_{alm})$$

where $\alpha : \mathbb{S}^* \to \wp(\mathbb{S})$ abstracts a trace into a set of states by ignoring the linkage between states: $\alpha(\Sigma) = \bigcup_{\sigma \in \Sigma} \{\sigma_i \mid 1 \leq i \leq |\sigma|\}$. We then generate the labeled data D as follows:

$$D = \{(\Pi(s), 1) \mid s \in S^+\} \cup \{(\Pi(s), 0) \mid s \in S^-\}.$$

Here, we assume a set of features $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ is given, where a feature $\pi_i : \mathbb{S} \to \mathbb{B}$ is a predicate on states. The feature vector of state s is denoted by $\Pi(s)$ and defined by $\Pi(s) = \langle \pi_1(s), \pi_2(s), \ldots, \pi_n(s) \rangle$. At the end of this section, we describe the state features used in our implementation.

Training models. Next, we train statistical models that distinguish between S^+ and S^-. For this purpose, we use an off-the-shelf classification algorithm. Let $\mathsf{TrainClassifier}_\lambda$ be a black-box algorithm for learning probabilisitc classifiers (e.g., gradient boosting). Classification algorithms typically have hyper-parameters, denoted λ, that are used to tune their performance on specific datasets. For example, gradient boosting classifiers include the maximum depth of a tree, the maximum depth of leaf nodes in a tree, and so on, as hyper-parameters. We assume a finite set Λ of hyper-parameter configurations is fixed. Given a hyper-parameter setting $\lambda \in \Lambda$ and training data D, the classification algorithm produces a probabilistic classifier C, i.e., $C = \mathsf{TrainClassifier}_\lambda(D)$, where the learned classifier $C : \mathbb{S} \to [0,1]$ is a function that takes a state and computes as output the probability that the state is alarm-contributing (we assume the classifier internally uses the features Π to transform a state into a feature vector). Let $\vec{C} = \{C_1, C_2, \dots, C_{|\Lambda|}\}$ be the set of all classifiers learned from D, one for each hyper-parameter $\lambda \in \Lambda$:

$$\vec{C} = \{\mathsf{TrainClassifier}_\lambda(D) \mid \lambda \in \Lambda\}.$$

Note that the behavior of the learned classifier varies depending on the hyper-parameter λ.

Offline model selection. We choose M candidate models from $\vec{C}$ by evaluating their performance on the validation set $\vec{P}_{valid}$. Given M ($\leq |\Lambda|$), we select M classifiers

$$\vec{C}^* = \{C'_1, C'_2, \dots, C'_M\} \subseteq \vec{C}$$

that collectively achieve the best performance when we run the analysis on the validation set $\vec{P}_{valid}$:

$$\vec{C}^* = \underset{\vec{C}' \subseteq \vec{C} \ s.t. \ |\vec{C}'| = M}{\operatorname{argmax}} \sum_{P \in \vec{P}_{valid}} \max_{C \in \vec{C}'} |\mathsf{Analyzer}(P, \mathcal{H}_C^{K_{main}})|$$

where K_{main} denotes the number of states maintained during the main analysis and the state-selection heuristic $\mathcal{H}_C^K$ with a learned classifier C is defined as follows:

$$\mathcal{H}_C^K(S) = \begin{cases} S & \text{if } |S| \leq K \\ \operatorname{argmax}_{S' \subseteq S \ s.t. \ |S'| = K} \sum_{s \in S'} C(s) & \text{if } |S| > K \end{cases}$$

As a special case, when $M = 1$, note that we choose the single classifier that performs the best on the validation programs:

$$C^* = \underset{C \in \vec{C}}{\operatorname{argmax}} \sum_{P \in \vec{P}_{valid}} |\mathsf{Analyzer}(P, \mathcal{H}_C^{K_{main}})|.$$

Example 2: Suppose $\vec{P}_{valid} = \{P_1, P_2, P_3, P_4\}$ and $\vec{C} = \{C_1, C_2, C_3\}$. Assume each (P_i, C_j) entry of the following table (left) stores the number of alarms obtained by running the analyzer on P_i using C_j, i.e., $|\mathsf{Analyzer}(P_i, \mathcal{H}_{C_j}^{K_{main}})|$:

	C_1	C_2	C_3
P_1	5	8	3
P_2	3	1	6
P_3	5	4	3
P_4	4	8	5

combination	max # of alarms
C_1, C_2	24
C_1, C_3	21
C_2, C_3	26

When $M = 2$, we select C_2 and C_3 because this combination maximizes the number of alarms (see the table on the right).

Online model selection via pre-analysis. Given M models $\vec{C}^* = \{C'_1, C'_2, \dots, C'_M\}$, we analyze a new, unseen program P by adaptively selecting the best model via pre-analysis. The aim of the pre-analysis is to estimate the behavior of models on P, and we simply do so by analyzing the program with a small K. Let K_{pre} be a pre-defined value such that $K_{pre} < K_{main}$. Then, we estimate the best model for P, denoted C_P^* as follows:

$$C_P^* = \underset{C \in \vec{C}^*}{\operatorname{argmax}} |\mathsf{Analyzer}(P, \mathcal{H}_C^{K_{pre}})|.$$

With C_P^*, the main analysis is run as $\mathsf{Analyzer}(P, \mathcal{H}_{C_P^*}^{K_{main}})$.

In summary, the performance of our analysis equipped with the learned heuristic is controlled by the three parameters:

1) K_{main}: the number of states to be maintained during the main analysis (higher is better but more expensive).
2) K_{pre}: the number of states to be maintained during the pre-analysis (higher is better but more expensive).
3) M: the number of models to be considered in online model selection (higher is better but more expensive).

State Features. Though our approach described so far is generally applicable to the class of static analyzers in Section III-A, the set Π of state features needs to be manually provided for each analysis instance. However, we found that many useful features may be already available in the implementation of abstract states of bug-finders, so that we can reuse those features without significant feature engineering effort.

For example, in our implementation for Infer, we defined Π to be the basic features of Infer's abstract state ($\mathbb{S}$), which are easily obtained from the definition of $\mathbb{S}$. The abstract state of Infer consists of the five components:

$$\mathbb{S} = \mathit{Node} \times \mathit{PC} \times \mathit{Pre} \times \mathit{Post} \times \mathit{SkippedCalls}$$

where Node denotes the current node (program point) in the control-flow graph, PC the path condition, Pre the inferred pre-condition of the node, Post the post-condition of the node, and $\mathit{SkippedCalls}$ the set of function calls skipped so far. Both Pre and Post consist of Heap, Stack, and Attrs. Heap denotes the abstract heap, which is a map from abstract addresses to abstract values. Stack is the abstract stack, which is a map from abstract stack addresses to abstract values. Attrs denotes attributes of abstract addresses, which is a map from abstract addresses to attributes. For example, an abstract address may have attributes with AddressOfStackVariable, Allocated, and/or, WrittenTo. AddressOfStackVariable indicates that the abstract address designates the address of a stack variable. Also, it annotates an abstract address with inferred conditions such as MustBeValid. Allocated and WrittenTo respectively indicate whether the address is allocated or not, and that a value stored at the address is used to describe a buggy trace since it was updated at a certain program point.

We basically translated such attributes into state features. From Node, we used 49 features, which encode the instruction types and structural information (e.g., the number of successors/predecessors) of nodes, all of which are provided as predicates or functions in Infer. From PC, we used 2 features, the satisfiability and the size of the path condition. From *Pre*/*Post*, we used 14 features denoting the numbers of abstract locations belonging to each attribute (e.g., Allocated) and 1 feature for stack size. From *SkippedCalls*, we used the number of skipped calls as a feature.

V. Evaluation

We have implemented our approach on top of Infer and conducted experiments to answer the following questions:

- **(RQ1) Effectiveness of our approach**: How effective is our learned state-selection heuristic? How significantly does it improve the performance of original Infer?
- **(RQ2) Efficiency of trace collection algorithm**: How efficiently does our trace collection algorithm produce alarm-triggering traces compared to the naive approach?
- **(RQ3) Sensitivity on data**: How sensitive is our data-driven approach to the amount of training data?
- **(RQ4) Impact of learning**: How significantly is the learned heuristic better than the random heuristic?
- **(RQ5) Comparison of alarms**: How similar or different are the alarms produced by original and data-driven Infer?

Setup. We used 85 open-source C programs collected from GNU and GitHub. Among them, we used 15 largest programs in Table I for evaluating the heuristic learned from the remaining 70 programs. The 70 programs were further randomly classified into the training set $\vec{P}_{train}$ of 49 programs (70%) and the validation set $\vec{P}_{valid}$ of 21 programs (30%).

We implemented our approach on top of Infer.Pulse, an interprocedural analyzer for detecting memory safety bugs such as memory leak and null dereference. Infer.Pulse is being actively developed at Meta and we used v.1.0.0, the latest release at the time we started this project. When running Infer, we used options `-j 1` (i.e., no multi-threading) and `--pulse-only` (i.e., disabling other checkers).

We ran the trace collection algorithm (Algorithm 3) on $\vec{P}_{train}$ with the default K value, i.e., $K = 20$. The time budget was 20 hours. To learn classifiers from the collected traces, we used the gradient boosting algorithm available in the scikit-learn library [18] and trained 18 classifiers, one for each configuration in the following space Λ of hyper-parameters:

$$\Lambda = \{(l, e, d) \mid l \in \{0.01, 0.1, 1.0\}, e \in \{100, 200\}, d \in \{1, 2, 4\}\}$$

where l, e, and d denote the learning rate, the number of estimators (boosting stages), and the maximum depth of estimators, respectively. Other hyper-parameters were set to the default values provided by the library. We used the validation set $\vec{P}_{valid}$ to choose the best classifier (i.e., $M = 1$) and the best three classifiers (i.e., $M = 3$).

All experiments were done on a virtual machine running Ubuntu 18.04 with 4 CPUs and 32GB memory. The host machine was iMac powered by 3.8 GHz 8-core Intel Core i7 processor with 128GB memory.

A. Effectiveness of Our Approach

Performance of original Infer. Table I shows the performance of original Infer with various K_{main} values. The result shows that increasing K_{main} allows Infer to detect more alarms (potential bugs) but doing so slows down the analysis significantly. For example, with $K_{main} = 5$, Infer took 659 seconds to detect 1,160 alarms over the 15 programs. With $K_{main} = 60$, Infer was able to detect 1,637 alarms (41.1% increase) but it took 39,684 seconds (60x increase).

Performance of data-driven Infer. Table II shows the performance of data-driven Infer with various M, K_{main}, and K_{pre} values. With $M = 1$ (i.e., the single best model is used) and $K_{main} = 5$, the result shows that our data-driven Infer can already compete with original Infer with $K_{main} = 60$, detecting 1,668 potential bugs in 875 seconds; the numbers of alarms are similar (1,637 vs. 1,668) but ours is 45x faster than the original. By increasing K_{main} to 20 while maintaining $M = 1$, data-driven Infer can find out 2,293 alarms, 40% more than original Infer with $K_{main} = 60$, with much smaller analysis time (7,208 seconds).

The result shows that we can accelerate the analysis by using multiple models and pre-analysis. When we used three models ($M = 3$) and pre-analysis with $K_{pre} = 1$, our data-driven Infer with $K_{main} = 5$ detected 2,329 alarms in 1,777 seconds, reducing the cost of the analysis with $M = 1$ and $K_{main} = 20$ by 4.1x. When $K_{pre} = 1$, pre-analysis took 870 seconds. Note that, when $M = 3$, pre-analysis is run three times per program to select the best one out of three models. (Over the 15 programs, running pre-analysis with $K_{pre} = 1$ only takes about 290 (870/3) seconds.)

The result also shows that we can use a more precise pre-analysis to detect more alarms with reasonable cost increase. When $M = 3$ and $K_{main} = 5$, data-driven Infer with $K_{pre} = 3$ reports 17.6% more alarms (2,329 $\rightarrow$ 2,740) than analysis with $K_{pre} = 1$ while increasing the cost by 1.4x (1,777 $\rightarrow$ 2,443). This is because pre-analysis with $K_{pre} = 3$ is able to more precisely capture the behavior of the main analysis than pre-analysis with $K_{pre} = 1$.

Comparison of cost/bug-finding efficiency. Figure 3 shows that data-driven Infer remarkably improves the cost/bug-finding efficiency of original Infer, where the x-axis denotes the number of alarms detectable by each analysis and the y-axis is the analysis time in seconds. To depict Figure 3, for original Infer, we randomly selected 20 K_{main} values between 1 and 60, and measured the number of alarms and the analysis time for each sampled K_{main}. For data-driven Infer, we considered two cases separately: $M = 1$ and $M = 3$. When $M = 1$, we plotted the performance of five analyses with $1 \le K_{main} \le 5$. When $M = 3$, we randomly sampled 15 analyses from $1 \le K_{pre} \le 3$ and $3 \le K_{main} \le 20$ such that $K_{main} > K_{pre}$.

B. Efficiency of Trace Collection Algorithm

Our trace collection algorithm in Section IV-A was by far more effective than the naive algorithm based on random sampling. Figure 4 compares the efficiency of the two algorithms. With the budget of 20 hours, the random-sampling approach discovered about 50,000 unique alarm-triggering

TABLE I: Performance of original Infer(Pulse). Analysis time in seconds. K_{main}: the number of disjuncts to be maintained during the main analysis.

Program	KLOC	$K_{main} = 5$		$K_{main} = 10$		$K_{main} = 20$		$K_{main} = 40$		$K_{main} = 60$	
		alarms	time	alarms	time	alarms	time	alarms	time	alarms	time
gawk	57	68	9	67	22	67	67	68	294	69	497
redis	74	4	25	4	73	4	196	2	585	2	1,042
nasm	103	55	13	76	16	80	27	99	49	93	86
sqlite	117	2	42	2	232	2	835	2	3,178	8	3,237
gnucobol	123	10	28	10	68	11	201	12	666	11	1,717
gnuastro	151	47	48	51	56	52	78	63	201	66	320
DyLP	157	11	25	17	56	18	158	21	567	19	1,018
httpd	207	373	64	435	167	473	554	466	2,091	506	5,190
git	238	5	42	4	97	6	278	5	1,116	5	2,047
freeipmi	318	10	36	12	77	13	197	16	674	16	1,178
vim	342	1	67	1	187	1	926	1	3,725	1	7,448
bluez	366	59	24	98	55	113	164	130	508	158	880
cpython	367	10	97	10	207	12	645	12	2,595	12	5,200
openssl	414	412	99	422	238	425	841	468	3,767	492	7,314
gettext	701	93	39	96	92	138	277	161	1,408	179	2,510
TOTAL	3,735	**1,160**	659	**1,305**	1,642	**1,415**	5,444	**1,526**	21,425	**1,637**	39,684

TABLE II: Performance of data-driven Infer(Pulse). M: the number of learned models (from which the best one is used for each program). K_{main}: the number of disjuncts to be maintained during the main analysis. K_{pre}: the number of disjuncts to be maintained during the pre-analysis. Total: total analysis time including pre- and main-analyse. Pre: pre-analysis time.

Program	$M = 1$				$M = 3$								
					$K_{pre} = 1$			$K_{pre} = 3$					
	$K_{main} = 5$		$K_{main} = 20$		$K_{main} = 5$			$K_{main} = 5$			$K_{main} = 10$		
	alarms	time	alarms	time	alarms	time		alarms	time		alarms	time	
						Total	Pre		Total	Pre		Total	Pre
gawk	76	24	82	205	75	33	15	75	47	30	78	70	29
redis	4	33	4	347	4	64	31	5	82	53	5	136	59
nasm	46	19	72	73	64	53	37	65	60	43	90	73	47
sqlite	8	66	8	649	8	98	42	8	156	99	8	256	92
gnucobol	10	27	12	400	10	73	33	10	108	72	10	182	78
gnuastro	54	52	68	182	54	200	137	64	225	159	71	281	166
DyLP	21	35	22	245	21	68	32	17	98	65	20	144	66
httpd	286	107	397	1,234	609	167	65	609	262	159	617	419	176
git	5	62	6	568	69	102	44	69	160	102	61	255	103
freeipmi	11	32	16	276	32	83	36	32	104	59	201	191	66
vim	167	101	398	1,005	167	166	67	525	232	151	535	403	162
bluez	139	40	167	243	77	73	39	113	94	63	148	148	69
cpython	16	131	17	853	15	259	142	15	336	228	17	498	254
openssl	669	80	842	495	968	198	74	968	296	170	757	439	177
gettext	156	67	182	431	156	141	75	165	184	123	186	268	132
TOTAL	**1,668**	875	**2,293**	7,208	**2,329**	1,777	870	**2,740**	2,443	1,576	**2,804**	3,764	1,674

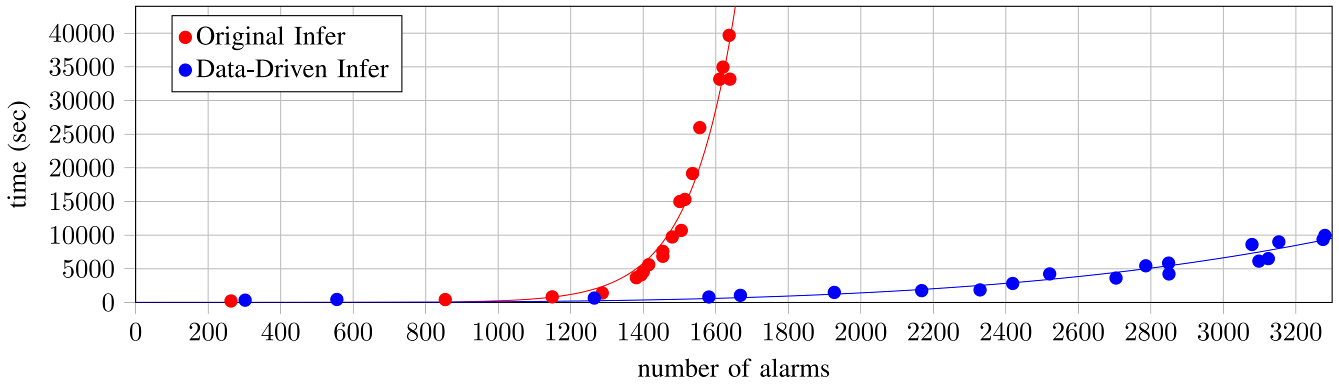

Fig. 3: Comparison of the cost/bug-finding efficiency between original and data-driven Infer.

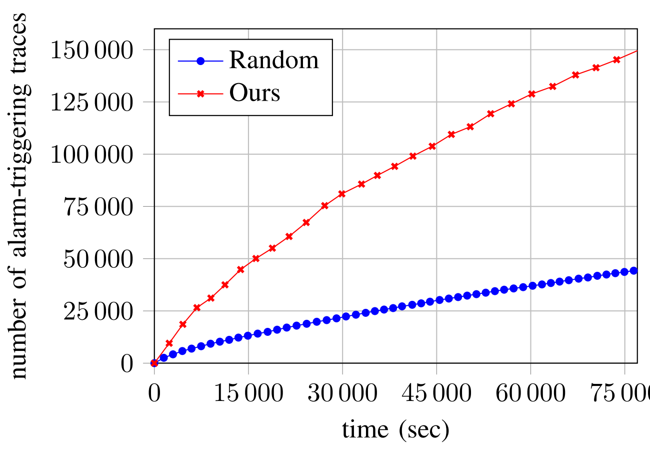

Fig. 4: Efficiency comparison between random and our trace-collection algorithms. The y-axis shows the number of unique alarm-triggering traces discovered by two algorithms.

traces. By contrast, our algorithm increased the number by 3x, discovering about 150,000 alarm-triggering traces.

C. Sensitivity on the Amount of Data

The effectiveness of our approach steadily increased as the underlying classifiers are trained with more data. Figure 5 compares the performance of data-driven Infer (with $K_{main} = 20$, $K_{pre} = 3$, and $M = 3$) when the classifiers are trained with different amounts of data (i.e., alarm-triggering and non-alarm-triggering traces). Our data-driven Infer reported 2,740 alarms when using 100% of the data collected over 20 hours. When the classifiers are trained on 50% of the data (sampled at random), over 10 trials, the number of alarms decreased to an average of 2,579 with a standard deviation of 130. Using 10% of the data, the average number was 2,398 and the standard deviation was 170.

D. Impact of Learning

Learning was essential for achieving the performance of data-driven Infer; simply using random state-selection heuristic without learning never achieved the performance. Figure 6 shows the performance of random Infer whose state-selection heuristic chooses K_{main} states at random (we set $K_{main} = 5$). Over 40 trials, random Infer produced 1068 alarms on average with a standard deviation of 208. The performance of original Infer was slightly better than random Infer (reporting 1,160 alarms). By contrast, our data-driven Infer was always better than random Infer. For example, when $M = 1$, ours reported 1,668 alarms, far outperforming random Infer.

E. Comparison of Reported Alarms

In principle, using our approach does not affect the false positive rate because original and data-driven analyses use the same abstract semantics (i.e, $[\![\mathsf{cmd}(n)]\!]$) and alarm-reporting criterion (i.e., alarms in (2)). However, the two analyses may produce different sets of alarms because they use different strategies (i.e., state-selection heuristics).

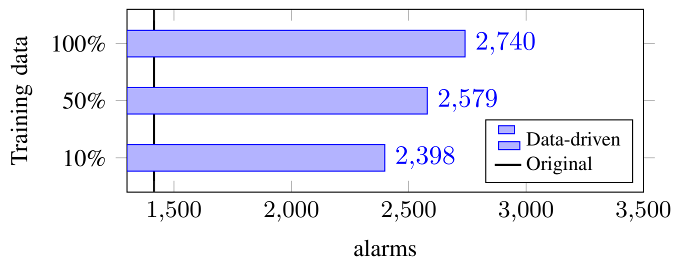

Fig. 5: Data-driven Infer sensitivity on data. Compared to the original Infer with $K_{main} = 20$.

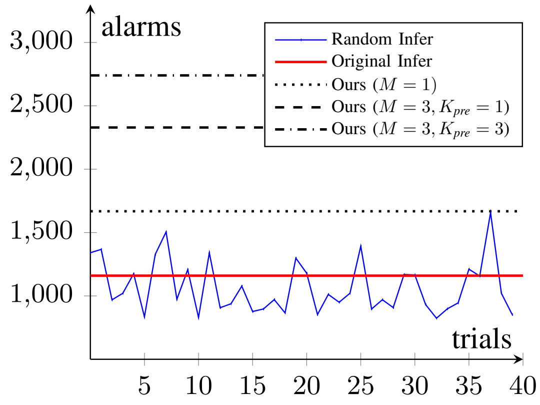

Fig. 6: Comparison with Random Infer (Infer with random state-selection heuristic). $K_{main} = 5$ for all analyses.

Table III compares the alarm sets produced by original and data-driven Infer. We ran original Infer with $K_{main} = 60$ and data-driven Infer with $M = 3$, $K_{main} = 10$, and $K_{pre} = 3$ to collect as many alarms as possible from both analyzers, and only compared memory-leak alarms since other types of alarms were not sufficiently reported. The result shows that 67% (1,024/1,525) of original Infer's alarms were reported by data-driven Infer as well. At the expense of missing 33%

TABLE III: Inclusion relationship between memory-leak alarms of original Infer (A) and data-driven Infer (B).

Programs	$\|A\|$	$\|B\|$	$\|A \cap B\|$	$\|A \setminus B\|$	$\|B \setminus A\|$
gawk	69	78	66	3	12
redis	0	0	0	0	0
nasm	93	90	57	36	33
sqlite	2	2	2	0	0
gnucobol	11	10	10	1	0
gnuastro	41	49	33	8	16
DyLP	9	12	8	1	4
httpd	502	614	342	160	272
git	4	6	2	2	4
freeipmi	15	200	13	2	187
vim	0	534	0	0	534
bluez	115	111	73	42	38
cpython	3	5	3	0	2
openssl	487	753	329	158	424
gettext	174	181	86	88	95
TOTAL	1525	2645	1024	501	1621

(501/1,525) of original alarms, data-driven Infer discovered additional 1,621 alarms.

The alarms, $B \setminus A$, exclusively found by data-driven Infer were not peculiar ones; they have chances of being found by original Infer with larger K_{main} values. For example, consider the memory-leak alarm produced by original Infer on `httpd`:

```
proxy/mod_proxy_fcgi.c:573: error: Memory Leak
memory allocated at line 572 by apr_brigade_create(),
is not freed after the last access at line 573.
```

where lines 572 and 573 are as follows:

```
572  ib = apr_brigade_create(r->pool, c->bucket_alloc);
573  ob = apr_brigade_create(r->pool, c->bucket_alloc);
```

and compare it with the following alarm found exclusively by data-driven Infer in a distant location of the same program:

```
filters/mod_ratelimit.c:119: error: Memory Leak
memory allocated at line 118 by apr_brigade_create()
is not freed after the last access at line 119.
```

where lines 118 and 119 are as follows:

```
118  c->tmpbb = apr_brigade_create(f->r->pool,ba);
119  c->holdingbb = apr_brigade_create(f->r->pool,ba);
```

Note that the root causes of the two alarms are the same, i.e., use of the user-defined allocator `apr_brigade_create`; the alarms are either true or false at the same time. Original Infer failed to report the latter simply because it could not analyze the corresponding program location.

VI. Related Work

Data-Driven Static Analysis. Developing effective heuristics for static analysis has been an active research area and various approaches have been proposed to balance competing factors such as precision, scalability, and soundness [19]–[40].

In particular, our work belongs to recent techniques called data-driven static analysis [32]–[40]. Because developing analysis heuristics manually for real-world programs is challenging, these data-driven approaches aim to generate analysis heuristics automatically from a corpus of programs using machine learning techniques. For example, Oh et al. [32] proposed a technique that uses Bayesian optimization to learn analysis heuristics for interval analysis, where the goal of the heuristics is to select subsets of program variables and procedures to apply flow-sensitivity and context-sensitivity, respectively. Heo et al. [41] developed a supervised learning algorithm to infer how to cluster program variables in relational analysis using the Octagon abstract domain. Singh et al. [38] used reinforcement learning to train a policy for choosing best abstract transformers in numerical analysis based on the Polyhedra domain. He et al. [36] presented an approach based on graph neural networks to selectively maintain constraints and remove redundant ones during numerical analysis. Heo et al. [35] used learning to control the soundness of analysis, where the learnt heuristic determines how differently to unroll loops.

Our work is different from the prior work on data-driven static analysis as follows. First, we tackle a new problem, i.e., learning a state-selection heuristic for industrial static analyzers; to our knowledge, no existing techniques have been developed for state-selection heuristics or industry-scale static bug-finders such as Infer [1]. Second, we present a new white-box learning algorithm specially designed for state-selection heuristics, which uses information about internal analysis states to reduce the learning cost. In contrast, most existing data-driven approaches are based on blackbox learning algorithms that only use the input and output behavior of static analysis as training data. The major shortcoming of such an algorithm is that learning is very expensive and therefore large programs cannot be used as training data. For example, Jeon et al. [34] used four small programs as training data due to the learning cost. Some techniques are based on white-box learning algorithms [33], [35] but they are not applicable to our problem. For example, the algorithm by Cha et al. [33] assumes an oracle whose construction requires to analyze codebases with full precision, which is obviously infeasible for path-sensitive analyses.

Scaling Disjunctive Analysis. Scaling disjunctive or path-sensitive analysis has been an important topic in static analysis [1], [10], [11], [16], [29], [30]. In particular, it is one of the major challenges to be overcome in modern static bug-finders used in industry [1], as industrial codebases are large and subject to frequent change. Our major difference from the prior work on this topic is that we address the problem using a data-driven approach while techniques used in prior work have been designed manually.

VII. Conclusion

Scaling disjunctive static bug-finders is one of the most important challenges in industrial settings. We demonstrated that using a data-driven technique to learn a state-selection heuristic can significantly improve the performance of a real-world static bug-finder. For a range of programs, our technique enabled Infer, a bug-finding static analyzer deployed within Meta, to effectively find alarms that are out of the reach of original Infer with a hand-crafted state-selection heursitic. To this end, we presented a learning framework that is generally applicable to disjunctive, bottom-up static analyzers.

Data Availability

The link below provides our source code that we used in our experiments.

https://github.com/facebookresearch/data_driven_infer

Acknowledgment

H. Oh was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51, Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair), the ICT Creative Consilience program(IITP-2023-2020-0-01819), and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944).

REFERENCES

[1] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, no. 8, p. 62–70, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3338112

[2] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[3] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *J. ACM*, vol. 58, no. 6, dec 2011. [Online]. Available: https://doi.org/10.1145/2049697.2049700

[4] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," *Proc. ACM Program. Lang.*, no. OOPSLA, 2022.

[5] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "Racerd: Compositional static race detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276514

[6] V. Bridgers, "Using the clang static analyzer to find bugs," 2020, presentation at 2020 LLVM Developers' Meeting.

[7] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 598–608.

[8] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 332–343. [Online]. Available: https://doi.org/10.1145/2970276.2970347

[9] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "Cloudbuild: Microsoft's distributed and caching build service," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 11–20. [Online]. Available: https://doi.org/10.1145/2889160.2889222

[10] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: Scalable path-sensitive memory leak detection for millions of lines of code," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 72–82. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00025

[11] Y. Xie, A. Chou, and D. Engler, "Archer: Using symbolic, path-sensitive analysis to detect memory access errors," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11. New York, NY, USA: Association for Computing Machinery, 2003, p. 327–336. [Online]. Available: https://doi.org/10.1145/940071.940115

[12] Y. Kim, "Using svace static analysis tool in samsung environments," http://0x1.tv/Using_Svace_static_analysis_tool_in_Samsung_environments_(Youil_Kim,_ISPRASOPEN-2019), 2019, presentation at ISPRASOPEN-2019.

[13] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, p. 115–125. [Online]. Available: https://doi.org/10.1145/1081706.1081728

[14] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 652–661.

[15] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 270–280. [Online]. Available: https://doi.org/10.1145/1375581.1375615

[16] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 693–706. [Online]. Available: https://doi.org/10.1145/3192366.3192418

[17] P. W. O'Hearn, "Incorrectness logic," *Proc. ACM Program. Lang.*, no. POPL, pp. 10:1–10:32, 2020.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] J. Park, H. Lee, and S. Ryu, "A survey of parametric static analysis," *ACM Comput. Surv.*, vol. 54, no. 7, Jul. 2021. [Online]. Available: https://doi.org/10.1145/3464457

[20] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 485–495. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594320

[21] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 423–434. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462191

[22] G. Xu and A. Rountev, "Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 225–236. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390658

[23] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *Static Analysis*, X. Rival, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 489–510.

[24] ——, "Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 278–291. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062360

[25] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 141:1–141:29, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276511

[26] ——, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 129–140. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236041

[27] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360574

[28] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective context-sensitivity guided by impact pre-analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 475–484. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594318

[29] H. Li, F. Berenger, B. E. Chang, and X. Rival, "Semantic-directed clumping of disjunctive abstract states," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, G. Castagna and A. D. Gordon, Eds. ACM, 2017, pp. 32–45. [Online]. Available: https://doi.org/10.1145/3009837.3009881

[30] B. Chimdyalwar and S. Kumar, "Selective path-sensitive interval analysis (wip paper)," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 146–150. [Online]. Available: https://doi.org/10.1145/3461648.3463855

[31] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis, "Making pointer analysis more precise by unleashing the power of selective context sensitivity," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485524

[32] H. Oh, H. Yang, and K. Yi, "Learning a strategy for adapting a program analysis via bayesian optimisation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 572–588. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814309

[33] S. Cha, S. Jeong, and H. Oh, *Learning a Strategy for Choosing Widening Thresholds from a Large Codebase*. Cham: Springer International Publishing, 2016, pp. 25–41. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-47958-3_2

[34] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 100:1–100:28, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133924

[35] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, pp. 519–529. [Online]. Available: https://doi.org/10.1109/ICSE.2017.54

[36] J. He, G. Singh, M. Püschel, and M. Vechev, "Learning fast and precise numerical analysis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1112–1127. [Online]. Available: https://doi.org/10.1145/3385412.3386016

[37] K. Chae, H. Oh, K. Heo, and H. Yang, "Automatically generating features for learning program analysis heuristics for c-like languages," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133925

[38] G. Singh, M. Püschel, and M. Vechev, "Fast numerical program analysis with reinforcement learning," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 211–229.

[39] M. Jeon, M. Lee, and H. Oh, "Learning graph-based heuristics for pointer analysis without handcrafting application-specific features," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428247

[40] K. Heo, H. Oh, and H. Yang, "Resource-aware program analysis via online abstraction coarsening," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 94–104. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00027

[41] ——, *Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 237–256. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-53413-7_12