PARF: Adaptive Parameter Refining for Abstract Interpretation

Zhongyi Wang*
Zhejiang University
Hangzhou, China
wzygomboc@zju.edu.cn

Linyu Yang*
Zhejiang University
Hangzhou, China
linyu.yang@zju.edu.cn

Mingshuai Chen†
Zhejiang University
Hangzhou, China
m.chen@zju.edu.cn

Yixuan Bu
Zhejiang University
Hangzhou, China
yixuanbu@zju.edu.cn

Zhiyang Li
Zhejiang University
Hangzhou, China
misakalzy@zju.edu.cn

Qiuye Wang
Fermat Labs, Huawei Inc.
Dongguan, China
wangqiuye2@huawei.com

Shengchao Qin
Xidian University
Xi'an, China
shengchao.qin@gmail.com

Xiao Yi
Fermat Labs, Huawei Inc.
Hong Kong, China
yi.xiao1@huawei.com

Jianwei Yin
Zhejiang University
Hangzhou, China
zjuyjw@zju.edu.cn

ABSTRACT

Abstract interpretation is a key formal method for the static analysis of programs. The core challenge in applying abstract interpretation lies in the configuration of abstraction and analysis strategies encoded by a large number of external parameters of static analysis tools. To attain low false-positive rates (i.e., accuracy) while preserving analysis efficiency, tuning the parameters heavily relies on expert knowledge and is thus difficult to automate. In this paper, we present a fully automated framework called PARF to adaptively tune the external parameters of abstract interpretation-based static analyzers. PARF models various types of parameters as random variables subject to probability distributions over latticed parameter spaces. It incrementally refines the probability distributions based on accumulated intermediate results generated by repeatedly sampling and analyzing, thereby ultimately yielding a set of highly accurate parameter settings within a given time budget. We have implemented PARF on top of FRAMA-C/EVA – an off-the-shelf open-source static analyzer for C programs – and compared it against the expert refinement strategy and FRAMA-C/EVA's official configurations over the FRAMA-C OSCS benchmark. Experimental results indicate that PARF achieves the lowest number of false positives on 34/37 (91.9%) program repositories with exclusively best results on 12/37 (32.4%) cases. In particular, PARF exhibits promising performance for analyzing complex, large-scale real-world programs.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**.

KEYWORDS

Automatic parameter tuning, Static analysis, Program verification

ACM Reference Format:
Zhongyi Wang, Linyu Yang, Mingshuai Chen, Yixuan Bu, Zhiyang Li, Qiuye Wang, Shengchao Qin, Xiao Yi, and Jianwei Yin. 2024. PARF: Adaptive Parameter Refining for Abstract Interpretation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695487

1 INTRODUCTION

The theory of abstract interpretation – ever since its inception in the 1970s by Cousot and Cousot [12] – has witnessed significant applications in the field of static program analysis, which aims to identify potential runtime errors (RTEs) without executing the program. Due to its core mechanism of safely approximating the concrete program semantics, abstract interpretation-based static analysis features soundness, i.e., true alarms of RTEs will not be missed, yet not completeness, i.e., false alarms may be emitted. These false alarms do not induce RTEs and thus may be eliminated by conducting more accurate approximations at the cost of less efficient analysis. State-of-the-art static analysis tools, such as Astrée [18], FRAMA-C [20], GOBLINT [29], MOPSA [16], and SPARROW [26], integrate multiple abstraction and analysis strategies encoded by various external parameters, thereby enabling users to balance analysis accuracy and efficiency by tuning these parameters.

Albeit with the extensive theoretical study of abstract interpretation, the picture is much less clear on its *parameterization* front: It is challenging to find a set of high-precision parameters to achieve low false-positive rates within a given time budget. The main reasons are two-fold: (i) Off-the-shelf static analyzers often provide a wide range of parameters subject to a huge and possibly infinite joint parameter space. For instance, the parameter setting in Fig. 1 consists of 13 external parameters that are highly relevant to the accuracy and efficiency of FRAMA-C/EVA; (ii) The process of seeking highly accurate results typically requires multiple trials of parameter setting and analysis, which generates a large amount of intermediate information such as RTE alarms and analysis time. Nevertheless,

*Both authors contributed equally to this research.
†Corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695487

```
[eva] Option -eva-precision 3 detected, automatic configuration of the analysis:
  option -eva-min-loop-unroll set to 0 (default value).
  option -eva-auto-loop-unroll set to 64.
  option -eva-widening-delay set to 2.
  option -eva-partition-history set to 0 (default value).
  option -eva-slevel set to 35.
  option -eva-ilevel set to 24.
  option -eva-plevel set to 70.
  option -eva-subdivide-non-linear set to 60.
  option -eva-remove-redundant-alarms set to true (default value).
  option -eva-domains set to 'cvalue,equality,gauges,symbolic-locations'.
  option -eva-split-return set to '' (default value).
  option -eva-equality-through-calls set to 'none'.
  option -eva-octagon-through-calls set to false (default value).
```

Figure 1: A typical parameter setting (under precision 3) of Frama-C/Eva [9] with different parameter types: integer, Boolean, string, and set-of-strings.

few static analyzers provide a fully automated approach to guiding the refinement of abstraction strategies based on such information. Therefore, the use of abstract interpretation-based static analysis tools still relies heavily on expert knowledge and experience.

Some advanced static analysis tools attempt to address the above challenges through various methods. Kästner et al. [18] summarize the four most important abstraction mechanisms in Astrée and recommend prioritizing the accuracy of related abstract domains, which amounts to narrowing down the parameter space, though Astrée currently does not support automatic parameter generation. Goblint [28, 30] implements a simple, heuristic autotuning method based on syntactical criteria, which can automatically activate or deactivate abstraction techniques before analysis. However, this method only generates an initial analysis configuration once and does not dynamically adapt to refine the parameter configuration. We defer a detailed survey of related work in this thread to Section 7.

This paper presents Parf – a fully automated framework to adaptively tune the external parameters of static analyzers. Parf models various types of parameters as random variables subject to probability distributions over latticed parameter spaces. Within a given time budget, Parf identifies a set of highly accurate parameter settings by incrementally refining the distributions based on accumulated intermediate results generated via repeatedly sampling and analyzing. The core components in Parf are the *representation of probability distributions* and the *strategy to refine the distributions*, which together guarantee that the refined joint distribution gives a parameter setting under which Parf either (i) yields more accurate analysis in expectation (i.e., *incrementality*), or (ii) in case of analysis failure, terminates with a higher probability (i.e., *adaptivity*).

We have implemented Parf on top of Frama-C/Eva and compared it against the expert refinement strategy (by trying out increasingly higher precisions) and Frama-C/Eva's official configurations over the Frama-C OSCS benchmark. Experimental results show that Parf achieves the highest accuracy on 34/37 (91.9%) program repositories with exclusively best results on 12/37 (32.4%) cases. In particular, Parf exhibits promising performance for analyzing complex, large-scale real-world programs. Moreover, we show that Parf can be generalized to improve the performance of other static analyzers such as Mopsa [16].

Contributions. Our main contributions are as follows:

- We present Parf, a new framework for adaptively tuning external parameters of abstract interpretation-based static analyzers. Parf is, to the best of our knowledge, the first *fully automated* approach that supports *incremental* refinement of such parameters. The technical novelty of Parf lies in the representation of distributions over a latticed parameter space and the incremental refinement strategy.
- We implement Parf and demonstrate its effectiveness and generality on standard benchmarks. We show that Parf outperforms state-of-the-art parameter-tuning strategies by discovering parameter settings leading to more accurate analysis, particularly for programs of a large scale.

2 BACKGROUND

2.1 Static Analysis via Abstract Interpretation

Static analysis is the process of analyzing a program without executing its source code. The goal of static analysis is to identify and help users eliminate potential *runtime errors* (RTEs) in the program, e.g., division by zero, overflow in integer arithmetic, and invalid memory accesses. Amongst others, *abstract interpretation* [12] is an established technique widely used for *sound* static program analysis: Instead of reasoning about potentially large sets of program states and behaviors, it provides a mechanism to soundly approximate the concrete semantics and thereby reasoning about the program against abstract properties over a certain abstract domain. The soundness nature guarantees that an abstract interpretation-based static analyzer is capable of certifying the absence of RTEs in case no RTE *alarm* is emitted. However, an alarm can be *false positive* due to the abstraction, which does not actually incur RTEs and thus may be excluded by conducting more accurate approximations.

Fig. 2 depicts an example of identifying potential runtime errors in a C program using the abstract interpretation-based static analyzer Frama-C/Eva. Given a source C program to be analyzed as in Fig. 2a, Frama-C/Eva emits, under a low-level precision, three alarms signifying potential runtime errors including signed overflow and out-of-bound array index (see Fig. 2b). These alarms are expressed as *assertions* written in the ANSI/ISO C specification language (ACSL) [9]. Nonetheless, by increasing the analysis precision to level 3, Frama-C/Eva suffices to rule out the two false-positive alarms concerning signed overflow (see Fig. 2c). This is because with `-eva-precision 3`, Frama-C/Eva unrolls the while loop and calculate the value of the variable `sum` under the assertion `index < 5`, ensuring that no numerical overflow occurs.

To facilitate the effective use of the tool, Frama-C/Eva provides various built-in precision levels, ranging from `-eva-precision 0` to `-eva-precision 11`, each of which packs a group of *fixed* parameter valuations; see Fig. 1 for an example. To balance accuracy and efficiency, the commonly adopted *expert refinement strategy* is to tune the parameters by applying Frama-C/Eva with increasingly higher precision levels. Such strategy is simple and effective in many cases, yet often yields suboptimal results due to the lack of flexibility in tuning individual parameters; see Section 5.

2.2 Parameterization of Static Analysis

Parameterization is a typical design approach to enhancing the flexibility and applicability of static analysis tools. For instance, the parameterization of Frama-C/Eva involves *correctness parameters* and *performance tuning* parameters. Misusing the former may lead

```c
#include <stdio.h>
int main()
{
  int array[5] = {1, 2, 3, 4, 5};
  int index = 0, sum = 0;

  while (index <= 10) {
    sum += array[index];
    sum *= 2;
    index ++;
  }

  printf("Sum of array: %d", sum);
  return 0;
}
```

(a) source C program to be analyzed

```c
#include <stdio.h>
int main(void)
{
  int array[5] = {1, 2, 3, 4, 5}, index = 0, sum = 0;
  while (index <= 10) {
    //@ assert Eva: index_bound: index < 5;
    /*@ assert Eva: signed_overflow:
                    sum + array[index] <= 2147483647; */
    sum += array[index];
    //@ assert Eva: signed_overflow: sum * 2 <= 2147483647;
    sum *= 2; index ++;
  }
  printf("Sum of array: %d", sum);
  return 0;
}
```

(b) analysis result with `-eva-precision 1`

```c
#include <stdio.h>
int main(void)
{
  int array[5] = {1, 2, 3, 4, 5};
  int index = 0, sum = 0;

  while (index <= 10) {
    //@ assert Eva: index_bound: index < 5;
    sum += array[index];
    sum *= 2;
    index ++;
  }
  printf("Sum of array: %d", sum);
  return 0;
}
```

(c) analysis result with `-eva-precision 3`

Figure 2: Identifying potential runtime errors in a C program via the abstract interpretation-based static analyzer Frama-C/Eva.

to unsound analysis results and therefore end-users can primarily configure the latter to balance the accuracy and efficiency of abstract interpretation-based static analysis. Throughout the rest of this paper, we consider only performance tuning parameters.

Fig. 1 lists a typical parameter setting under `-eva-precision 3` in Frama-C/Eva. This setting consists of parameter valuations that are highly relevant to the accuracy and efficiency of abstract interpretation. For instance, users can set the upper bound on the number of times that Frama-C/Eva automatically unrolls a loop by configuring the value of `-eva-auto-loop-unroll`, thereby preventing performance degradation caused by excessive loop unrolling; By configuring the value of `-eva-domains`, users can specify a dedicated set of abstract domains used in the analysis. More concretely, the Frama-C/Eva command

```
frama-c *.c -eva -eva-auto-loop-unroll 4
            -eva-domains cvalues,octagon,gauges
```

performs abstract interpretation-based static analysis over all C program files in the current directory with loop unrolling limit 4 and abstract domains `cvalues`, `octagon`, and `gauges`. The latter two abstract domains are commonly used to infer different forms of relations between program variables, which both rely on the `cvalues` domain. More detailed parameterization of Frama-C/Eva can be found in the manual [9].

2.3 The Complete Lattice Structure

Complete lattices are an important mathematical tool used in formalizing the theory of abstract interpretation, as well as in structuring the parameter spaces in our approach. A *complete lattice* $(L, \sqsubseteq)$ consists of a (possibly infinite) carrier set L and a partial order $\sqsubseteq$ over L, where every subset $S \subseteq L$ has both a *greatest lower bound* $\bigsqcap S \in L$ (also known as the *meet* of S) and a *least upper bound* $\bigsqcup S \in L$ (also known as the *join* of S). For just two elements $\{x, y\} \subseteq L$, we denote their meet by $x \sqcap y$ and their join by $x \sqcup y$. Moreover, we denote by $\top \triangleq \bigsqcup L$ as the *greatest* element of the lattice, and by $\bot \triangleq \bigsqcap \emptyset$ as the *least* element.

Given two complete lattices $(L, \sqsubseteq)$ and $(L', \sqsubseteq')$, a function $f : L \to L'$ is *monotonic* if and only if it respects the partial orders, i.e., for any $x, y \in L$, $x \sqsubseteq y$ implies $f(x) \sqsubseteq' f(y)$ (monotonically increasing) or $f(y) \sqsubseteq' f(x)$ (monotonically decreasing).

3 PROBLEM FORMULATION

This section formalizes our parameter refining problem. To this end, we first model parameter spaces as complete lattices and then encode static analyzers as monotonic functions over these lattices.

3.1 Parameter Spaces

Given a finite sequence of parameters $P = (P^1, P^2, \ldots, P^n)$, we associate each parameter P^i with its corresponding *parameter space* PS^i, which is the (possibly infinite) set of all possible values of parameter P^i. As shown in Fig. 1, commonly used parameters can be classified into four types: (non-negative) integer, Boolean, string, and set-of-strings. For instance, the parameter spaces of the four parameters `-eva-slevel` (integer), `-eva-octagon-through-calls` (Boolean), `-eva-equality-through-calls` (string), and `-eva-domains` (set-of-strings) in Frama-C/Eva are as follows:

$$\begin{aligned} PS^{\text{slevel}} &= \mathbb{Z}^{\infty}_{\geq 0}\,, \\ PS^{\text{octagon-through-calls}} &= \{\mathit{false}, \mathit{true}\}\,, \\ PS^{\text{equality-through-calls}} &= \{\text{`none'}, \text{`formals'}\}\,, \\ PS^{\text{domains}} &= \mathcal{P}\{d_1, d_2, d_3, d_4, d_5\}\,. \end{aligned}$$

Here, $\mathbb{Z}^{\infty}_{\geq 0} \triangleq \mathbb{N} \cup \{\infty\}$ is the set of non-negative integers extended with ∞. Note that ∞ is not a valid parameter value for the underlying static analyzer; rather, it is a symbolic element used to enforce a complete lattice structure of the parameter space (see below). PS^{domains} is the power set of the set of five commonly used abstract domains, namely,

$$\{\texttt{cvalues}, \texttt{octagon}, \texttt{equality}, \texttt{gauges}, \texttt{symbolic-locations}\}\,.$$

Latticed Parameter Spaces. We observe that every parameter space forms a complete lattice $(PS^i, \sqsubseteq)$:

- For an integer parameter, the partial order $\sqsubseteq$ coincides with $\leq$ over $\mathbb{Z}^{\infty}_{\geq 0}$ (where $k \leq \infty$ for any $k \in \mathbb{Z}^{\infty}_{\geq 0}$); The operators $\sqcap$ and $\sqcup$ are then equivalent to min and max, respectively.
- For a Boolean parameter, the partial order $\sqsubseteq$ coincides with implication $\Rightarrow$ over the Boolean domain, e.g., $\mathit{false} \Rightarrow \mathit{true}$; The operators $\sqcap$ and $\sqcup$ are then equivalent to logical connectives $\wedge$ and $\vee$, respectively.

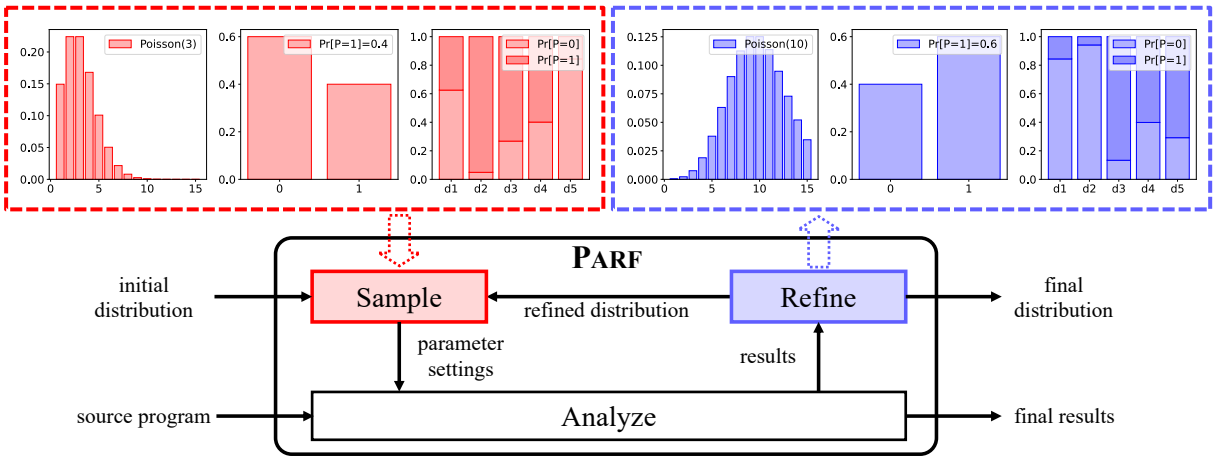

Figure 3: Architecture of the PARF framework. PARF adopts a multi-round iterative mechanism: In each iteration, PARF (i) repeatedly *samples* parameter settings based on the initial or refined probability distribution of parameters, then (ii) uses these parameter settings as inputs to the static analyzer to *analyze* the program, and finally (iii) utilizes the analysis results to *refine* the probability distribution of parameters. PARF continues this process until the prescribed time budget is exhausted, upon which it returns the analysis results of the final round together with the final probability distribution of parameters.

- Since all string parameters in our setting have only two possible values, we map string parameters to the Boolean domain and treat them as Boolean parameters.[1]
- For a set-of-strings parameter, $\sqsubseteq$ coincides with $\subseteq$; The operators $\sqcap$ and $\sqcup$ are then equivalent to set operations $\cap$ and $\cup$, respectively.

Next, we define the joint *space of parameter settings* as

$$PS \triangleq PS^1 \times PS^2 \times \cdots \times PS^n \ .$$

Notice that PS also forms a complete lattice $(PS, \preceq)$, where $\preceq$ is the point-wise lifting of $\sqsubseteq$ over individual parameters. Moreover, the meet $\wedge$ and join $\vee$ operators on $(PS, \preceq)$ are

$$p \wedge q \triangleq (p^1 \sqcap q^1, p^2 \sqcap q^2, \ldots, p^n \sqcap q^n) \ ,$$
$$p \vee q \triangleq (p^1 \sqcup q^1, p^2 \sqcup q^2, \ldots, p^n \sqcup q^n)$$

for any parameter settings $p, q \in PS$ with $p = (p^1, p^2, \ldots, p^n)$ and $q = (q^1, q^2, \ldots, q^n)$. For ease of presentation, we abuse the notations $\sqsubseteq$, and $\sqcap, \sqcup$ to denote the partial order $\preceq$ and operators $\wedge, \vee$, respectively, in the joint space as well.

3.2 The Static Analyzer

Next, we show how a static analyzer can be formulated as a monotonic function over the latticed joint parameter space PS. To this end, we abstract the procedure of executing static analysis on some program *prog* with some parameter setting $p \in PS$ as a *function* receiving these two parameters and returning a set of (RTE) alarms:

$$Analyze\colon \quad Prog \times PS \ \to \ \mathcal{P}(A_{\text{uni}}) \ , \quad (prog, p) \mapsto A_p$$

where $Prog$ denotes the set of all valid source programs, A_{uni} denotes the universe of all possible alarms that can be emitted by the analyzer (which is determined by running the analyzer with the *least* precise parameter setting), and $A_p \subseteq A_{\text{uni}}$ denotes the set of all alarms emitted under parameter setting p.

Static Analyzer as Monotonic Function. Note that the codomain of the function *Analyze*, i.e., $\mathcal{P}(A_{\text{uni}})$, is naturally equipped with a complete lattice structure $(\mathcal{P}(A_{\text{uni}}), \subseteq)$. Our incremental refining framework requires that the underlying static analyzer exhibits *monotonicity* over the parameters, that is, for all source programs $prog \in Prog$ and pairs of parameter settings $p_1, p_2 \in PS$,

$$p_1 \sqsubseteq p_2 \quad \text{implies} \quad Analyze(prog, p_2) \subseteq Analyze(prog, p_1) \quad (1)$$

namely, a *greater* parameter setting (in the latticed joint parameter space) induces *fewer* alarms and thereby *more accurate* analysis. Note that monotonicity is a reasonable and commonly adopted assumption in most state-of-the-art static analyzers [35];[2] it is also the rationale behind the aforementioned expert refinement strategy where increasing the precision level yields more accurate analysis.

The complete lattice structure of the joint parameter space and the monotonicity of static analyzers allow us to *compare* different parameter settings in terms of accuracy (within a given time budget), which forms the basis of our *parameter refining problem*:

> **Problem Statement.** Given a source program $prog \in Prog$, a time budget $T \in \mathbb{R}_{>0}$, an abstraction interpretation-based static analyzer *Analyze*, and the joint space of parameter settings PS of *Analyze*, find a parameter setting $p \in PS$ such that $Analyze(prog, p)$ returns as few alarms as possible within T.

We remark that finding the optimum parameter setting – which amounts to a brute-force search over a possibly infinite joint parameter space – is often intractable in practice. We thus aim to develop a fully automated framework to derive parameter settings that

[1] For string-typed parameters with $k > 2$ possible values: If the parameter exhibits a total order in terms of precision as per Eq. (1), its space can be mapped to $\{0, 1, \ldots, k-1\}$ and thus be treated as an integer-typed parameter. See example in Section 5.4.

[2] FRAMA-C/EVA exhibits monotonicity on most target programs; a corner case is given in [9, Section 6.7], where adding a new domain may unpredictably induce new alarms.

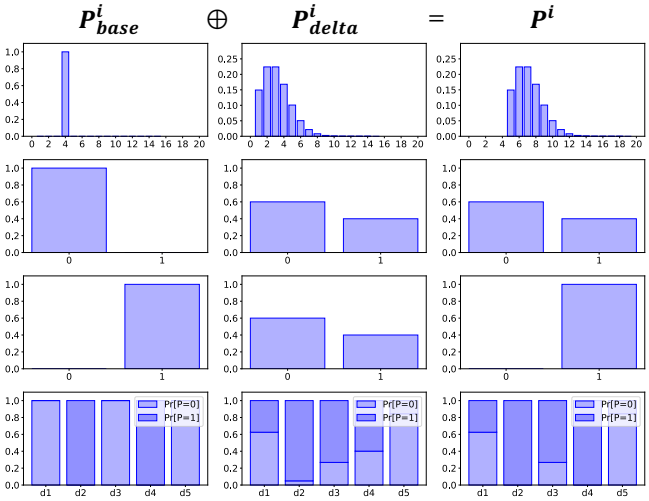

Figure 4: Constructing P^i from P^i_{base} and P^i_{delta} via $\oplus$. The three columns, from left to right, are P^i_{base}, P^i_{delta}, and P^i, respectively; The four rows, from top to bottom, correspond to an integer parameter, a Boolean parameter with $Pr[P^i_{\text{base}} = 0] = 1$, a Boolean parameter with $Pr[P^i_{\text{base}} = 1] = 1$, and a set-of-strings parameter P^{domains} over $\mathcal{P}\{d_1, d_2, d_3, d_4, d_5\}$, respectively.

yield more accurate analysis than those by the expert refinement strategy. Our framework treats the underlying static analyzer as a black-box function, and thus can be integrated with any abstraction interpretation-based static analyzer (see Section 4). For clarity, we take the open-source C analyzer Frama-C/Eva as an example; its extension to other static analyzers is investigated in Section 5.

4 THE PARAMETER REFINING FRAMEWORK

This section presents our Parf framework for Adaptive Parameter ReFining. Fig. 3 illustrates the architecture of Parf: It models external parameters of a static analyzer as *random variables* subject to probability distributions over latticed parameter spaces. It *incrementally* refines the probability distributions based on accumulated intermediate results generated by repeatedly sampling and analyzing, thereby ultimately yielding a set of highly accurate parameter settings within a given time budget.

The key components in Parf are the *representation of probability distributions* and the *strategy to refine the distributions*; they two cooperate to guarantee that the refined joint distribution produces a parameter setting under which Parf either (i) yields more accurate analysis in expectation (i.e., *incrementality*), or (ii) in case of analysis failure, terminates with a higher probability (i.e., *adaptivity*).

4.1 Probability Distributions of Parameters

Consider the joint parameter space PS spanned by a fixed set of parameters $\{P^1, P^2, \ldots, P^n\}$. We represent every parameter P^i as a *random variable* with sample space PS^i. More concretely, P^i is a (measurable) function of the form

- $P^i : PS^i \to \mathbb{Z}^\infty_{\geq 0}$ for an integer parameter,
- $P^i : PS^i \to \{0, 1\}$ for a Boolean parameter, and
- $P^i : PS^i \to \{0, 1\}^c$, i.e., a c-dimensional random vector, for a set-of-strings parameter with cardinality c. Here, c is the number of available strings in the set.

For Frama-C/Eva, the only considered set-of-strings parameter is `-eva-domains`, which represents the employed abstract domains.

The (ordered) sequence of parameters is then an n-dimensional random vector $P = (P^1, P^2, ..., P^n)$ with sample space PS:

$$P\colon \quad PS \to \underbrace{\mathbb{Z}^\infty_{\geq 0} \times \cdots \times \mathbb{Z}^\infty_{\geq 0}}_{\text{integer parameters}} \times \underbrace{\{0,1\} \times \cdots \times \{0,1\}}_{\text{Boolean parameters}} \times \{0,1\}^c$$

where c is the cardinality of the unique set-of-strings parameter `-eva-domains`. For cases with $k > 1$ set-of-strings parameters, the codomain of P is naturally extended by $\times\{0,1\}^{c_1} \times \cdots \times \{0,1\}^{c_{k-1}}$.

Refinable Probability Distribution. As shown in Fig. 3, Parf adopts an iterative `Sample-Analyze-Refine` mechanism to achieve a high-precision (joint) probability distribution of parameter settings within a given time budget. Therefore, the underlying probability distribution needs to feature two abilities: (i) it can effectively *retain* the accumulated knowledge during the iterative procedure, and (ii) it can efficiently *explore* the uncharted parameter space.

To this end, we represent every random variable P^i of the i-th parameter as a combination of a *base* random variable and a *delta* random variable:

$$P^i \quad = \quad \underbrace{P^i_{\text{base}}}_{\text{for retaining}} \quad \oplus \quad \underbrace{P^i_{\text{delta}}}_{\text{for exploring}} \tag{2}$$

where P^i_{base} is be dedicated to *retaining* the accumulated knowledge whilst P^i_{delta} is used to *explore* the parameter space; they share the same sample space and range with P^i. P^i_{base} follows a one-point distribution, i.e., $Pr[P^i_{\text{base}} = p^i] = 1$ for some sample $p^i \in PS^i$; The distribution of P^i_{delta} depends on the parameter type:

- For an integer parameter, $P^i_{\text{delta}} \sim Poisson(\lambda)$ with $\lambda \in \mathbb{R}_{>0}$.
- For a Boolean parameter, $P^i_{\text{delta}} \sim Bernoulli(q)$; $q \in [0, 1]$.
- For a set-of-strings parameter with cardinality c, P^i_{delta} follows a c-dimensional independent joint Bernoulli distribution: $P^i_{\text{delta}} \sim Bernoulli(q_1) \times \cdots \times Bernoulli(q_c)$.

We now illustrate by Fig. 4 how to construct P^i from P^i_{base} and P^i_{delta} via the binary operator $\oplus$ as in Eq. (2):

- For an integer parameter, $\oplus$ is equivalent to $+$: Suppose $Pr[P^i_{\text{base}} = p^i] = 1$ for some non-negative integer $p^i \in \mathbb{Z}^\infty_{\geq 0}$, $P^i = P^i_{\text{base}} \oplus P^i_{\text{delta}}$ is then a rightward shift of P^i_{delta} by p^i.
- For a Boolean parameter, $\oplus$ simulates logical disjunction $\vee$: If $Pr[P^i_{\text{base}} = 0] = 1$, then $P^i = P^i_{\text{delta}}$; otherwise if $Pr[P^i_{\text{base}} = 1] = 1$, then $P^i = P^i_{\text{base}}$.
- For a set-of-strings parameter with cardinality c, $\oplus$ is the point-wise lifting of $\vee$ to c-dimensional random vectors; see Fig. 4 for an example of P^{domains}.

We can now lift the binary operator $\oplus$ to random vectors as

$$P \;=\; P_{\text{base}} \oplus P_{\text{delta}} \;\triangleq\; \left(P^1_{\text{base}} \oplus P^1_{\text{delta}}, \ldots, P^n_{\text{base}} \oplus P^n_{\text{delta}}\right).$$

Algorithm 1 PARF: Adaptive Parameter Refining

Input: The source program *prog*, initial probability distribution of parameters P_{init}, and time budget T.
Output: Final probability distribution of parameters P_{final}.

1: $P_{\text{base}},\ P_{\text{delta}} \leftarrow \texttt{Extract}(P_{\text{init}})$;
2: $time,\ A_{\text{uni}} \leftarrow \texttt{Analyze}(prog, P_{\text{base}})$;
3: $T_r \leftarrow \text{Max}(time \times \alpha, T \times \beta)$;
4: $count \leftarrow 0$;
5: **repeat**
6: $\quad p\_list \leftarrow \texttt{Sample}(P_{\text{base}} \oplus P_{\text{delta}},\ num_{\text{sample}})$;
7: $\quad R\_list \leftarrow \texttt{MapAnalyze}(prog,\ p\_list,\ T_r)$;
8: $\quad P_{\text{base}}, P_{\text{delta}} \leftarrow \texttt{Refine}(p\_list, R\_list, A_{\text{uni}}, P_{\text{base}}, P_{\text{delta}})$;
9: $\quad T, T_r, count \leftarrow T - T_r, T_r \times 2, count + 1$;
10: **until** $T \leq 0$ **or** $count = num_{\text{refine}}$;
11: $P_{\text{final}} \leftarrow P_{\text{base}}$;
12: **return** P_{final} ;

Throughout the rest of this section, we will show how to refine P_{base} and P_{delta} in a way such that the composed random variable vector P ensures incrementality and adaptivity.

4.2 The PARF Algorithm

Algorithm 1 outlines the workflow of PARF: Lines 1-4 correspond to the initialization of variables and the basic analysis of the source program; Lines 5-10 describe the iterative `Sample-Analyze-Refine` procedure; Lines 11-12 acquire and return the final distribution.

In Line 1, PARF extracts P_{base} and P_{delta} from the initial joint probability distribution P_{init} specifying the initial parameter settings. Here, P_{base} follows a one-point distribution, corresponding to a unique set of parameters, which is actually the default parameter setting of FRAMA-C/EVA. In Line 2, we analyze the program using the default parameter setting and record the analysis time *time* and the universe set of alarms A_{uni}. Note that, due to the monotonicity assumption (see Section 3.2), alarms obtained in subsequent analyses using refined parameter settings will be subsets of A_{uni}. In Line 3, we set the initial time budget T_r for every round of the `Sample-Analyze-Refine` process based on two hyper-parameters α and β. In Line 4, we initialize a counter *count*.

Lines 5-10 represent the iterative `Sample-Analyze-Refine` process in PARF. In Line 6, PARF samples num_{sample} (a hyper-parameter) times based on the distribution determined by $P_{\text{base}} \oplus P_{\text{delta}}$ and returns a list *p_list* storing all the parameter settings. In Line 7, PARF separately analyzes the program with each parameter setting in *p_list* and obtains a list of results *R_list*. Each element of *R_list* is a pair $\langle p, A\rangle$ storing the parameter setting together with its corresponding analysis alarms. The time of `MapAnalyze` is limited to T_r. In Line 8, PARF utilizes the information from *p_list*, *R_list*, and A_{uni} to refine the random vectors P_{base} and P_{delta} for the next round of analysis. In Line 9, the left time budget and counter are updated, and the time budget of the next round is doubled due to the increase in parameter precision. The loop terminates when either the time budget T is exhausted or the number of refinement iterations reaches the hyper-parameter num_{refine}. Then the one-point distributed P_{base} is returned as the final distribution, which corresponds to a unique set of parameters.

Algorithm 2 `Refine`: Incremental Refining

Input: List of parameter settings *p_list*, list of results *R_list*, universe alarms A_{uni}, and $P_{\text{base}}, P_{\text{delta}}$.
Output: Refined distributions P'_{base} and P'_{delta}.

1: /* Step 1: Refine P_{base} */
2: $P'_{\text{base}} \leftarrow P_{\text{base}}$;
3: **for** all $a \in A_{\text{uni}}$ **do**
4: $\quad P_a \leftarrow \top$;
5: $\quad$ **for** all $\langle p, A\rangle \in R\_list$ **and** $a \notin A$ **do**
6: $\quad\quad p_a \leftarrow p_a \sqcap p$;
7: $\quad$ **end for**
8: $\quad$ **if** $p_a \neq \top$ **then**
9: $\quad\quad P'_{\text{base}} \leftarrow P'_{\text{base}} \sqcup p_a$;
10: $\quad$ **end if**
11: **end for**
12:
13: /* Step 2: Refine P_{delta} */
14: $\eta_{\text{scale}} \leftarrow \frac{2\times|R\_list|+1}{|p\_list|}$;
15: $P'_{\text{delta}} \leftarrow \eta_{\text{scale}} \otimes P_{\text{delta}}$;
16: **return** $P_{\text{base}}, P_{\text{delta}}$;

Remark. Although FRAMA-C/EVA per se does not support parallel analysis, our PARF algorithm allows for *parallelization*. Specifically, in Line 6 of Algorithm 1, PARF generates a list *p_list* containing parameter settings obtained through the `Sample` function in one go; Then, in Line 7, PARF uses each parameter setting in *p_list* for static analysis. As these analyses share no data/control-flow dependencies, they can be threaded in parallel across multiple processes. ◁

4.3 The Refinement Strategy

Algorithm 2 describes our incremental strategy to refine the probability distribution of parameter setting, i.e., $P_{\text{base}} \oplus P_{\text{delta}}$, based on the list of parameter settings *p_list*, list of results *R_list*, and universe set of alarms A_{uni}. The refinement strategies for P_{base} (Step 1, Lines 2-11) and P_{delta} (Step 2, Lines 14-15) are detailed below.

Refine P_{base}. The refining method for P_{base} involves two nested loops: In the inner loop (Lines 5-7), PARF calculates the "parameter setting with *lowest* precision" p_a that can eliminate the given false alarm a; In the outer loop (Lines 3-11), PARF calculates the "parameter setting with lowest precision" P'_{base} that can eliminate *all* newly discovered false alarms in this `Sample-Analyze-Refine` round. The core idea of the refining method for P_{base} can be formalized as

$$P'_{\text{base}} \;=\; \bigsqcup_{a\in A_{\text{uni}}} p_a \;=\; \bigsqcup_{a\in A_{\text{uni}}} \Big(\bigsqcap_{\substack{\langle p,A\rangle \in R\_list \\ a\notin A}} p \Big). \tag{3}$$

Here, $\bigsqcap_{\langle p,A\rangle \in R\_list,\, a\notin A} p$ represents the *greatest lower bound* (for the *lowest* precision) of all sampled parameter settings p which can eliminate false alarm a (signified by $a \notin A$); $\bigsqcup_{a\in A_{\text{uni}}} p_a$ represents the *least upper bound* (for eliminating *all* false alarms) of all such p_a.

Remark. Our assumption on the monotonicity of the underlying analyzer is crucial to guarantee the improvement of precision using our refinement strategy: Without monotonicity, the (outer) least

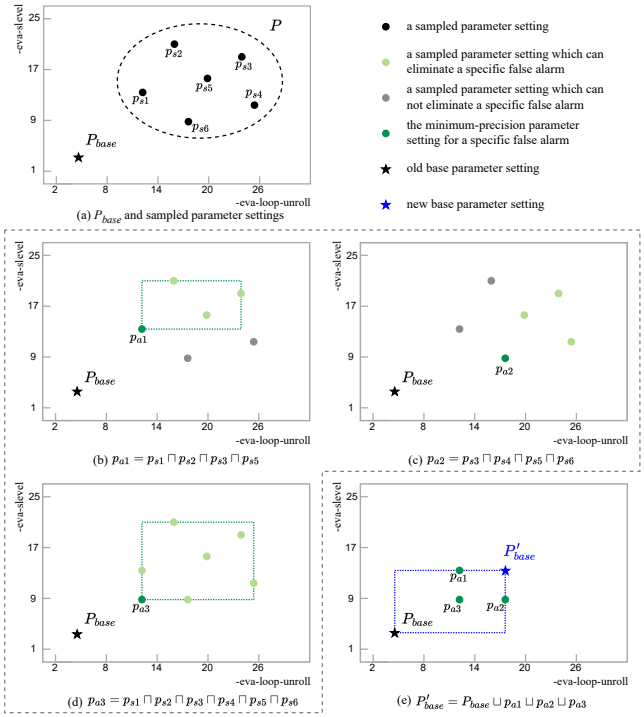

Figure 5: Incremental refinement of P_{base}.

upper bound in Eq. (3) may introduce again a false alarm ruled out by the (inner) greatest lower bound. ◁

We demonstrate the refinement of P_{base} by means of the following example and Fig. 5.

Example 4.1 (Refinement of P_{base}). Consider two integer parameters `-eva-slevel` and `-eva-loop-unroll` encoded by random vector P in a two-dimensional parameter space $PS = PS^{\text{slevel}} \times PS^{\text{loop-unroll}}$ as shown in Fig. 5(a)-(e). Suppose we have $P_{\text{base}} = (4, 4)$, the universe set of alarms $A_{\text{uni}} = \{a_1, a_2, a_3\}$, the list of sampled parameter settings $p\_list = (p_{s1}, p_{s2}, p_{s3}, p_{s4}, p_{s5}, p_{s6})$ (see Fig. 5(a)), and the list of analysis results:

$$\begin{aligned} R\_list &= (\langle p_{s1}, A_1\rangle, \langle p_{s2}, A_2\rangle, \langle p_{s3}, A_3\rangle, \langle p_{s4}, A_4\rangle, \langle p_{s5}, A_5\rangle, \langle p_{s6}, A_6\rangle) \\ &= (\langle (12, 14), \{a_2\}\rangle, \langle (16, 21), \{a_2\}\rangle, \langle (24, 19), \emptyset\rangle, \\ &\quad \langle (26, 12), \{a_1\}\rangle, \langle (20, 16), \emptyset\rangle, \langle (18, 9), \{a_1\}\rangle) \end{aligned}$$

i.e., false alarm a_1 can be eliminated by analyses of $\{p_{s1}, p_{s2}, p_{s3}, p_{s5}\}$, a_2 by $\{p_{s3}, p_{s4}, p_{s5}, p_{s6}\}$, and a_3 by $\{p_{s1}, p_{s2}, p_{s3}, p_{s4}, p_{s5}, p_{s6}\}$. Then Fig. 5(b)-(d) visualizes the computation of p_{a1}, p_{a2}, and p_{a3} (Lines 5-7 of Algorithm 2):

$$\begin{aligned} p_{a1} &= p_{s1} \sqcap p_{s2} \sqcap p_{s3} \sqcap p_{s5} = (12, 14)\,, \\ p_{a2} &= p_{s3} \sqcap p_{s4} \sqcap p_{s5} \sqcap p_{s6} = (18, 9)\,, \\ p_{a3} &= p_{s1} \sqcap p_{s2} \sqcap p_{s3} \sqcap p_{s4} \sqcap p_{s5} \sqcap p_{s6} = (12, 9)\,. \end{aligned}$$

Fig. 5(e) depicts the computation of P'_{base} (Lines 3-11 of Algorithm 2):

$$P'_{\text{base}} = P_{\text{base}} \sqcup p_{a1} \sqcup p_{a2} \sqcup p_{a3} = (18, 14)$$

P'_{base} thus is the *least precise* parameter setting that can eliminate *all* newly discovered false alarms in the current iteration. ◁

Note that, according to the definition of $\sqcup$, $P_{\text{base}} \sqsubseteq P'_{\text{base}}$ always holds. This means that P_{base} is incrementally refined.

Refine P_{delta}. We use the number of (successfully) completed analyses, e.g., $|R\_list|$, and the number of generated parameter settings, e.g., $|p\_list| = num_{\text{sample}}$, to refine P_{delta} in each round, yielding a scaling factor η_{scale} as in Line 14 of Algorithm 2. A *larger* value of η_{scale} indicates that more analyses have been completed within time T_r, suggesting that a *more extensive exploration* of the parameter space (by scaling up P_{delta}) is possible, and vice versa.

More concretely, we define the scaling operator $\otimes$ as

$$\eta_{\text{scale}} \otimes P_{\text{delta}} = \left(\eta_{\text{scale}} \otimes P^1_{\text{delta}}, \cdots, \eta_{\text{scale}} \otimes P^n_{\text{delta}}\right)\,, \quad (4)$$

where each component is subject to the distribution type of P^i_{delta}:

- If $P^i_{\text{delta}} \sim Poisson(\lambda)$, then
$$\eta_{\text{scale}} \otimes P^i_{\text{delta}} \sim Poisson\left(\eta_{\text{scale}} \times \lambda\right)\,.$$
- If $P^i_{\text{delta}} \sim Bernoulli(q)$, then
$$\eta_{\text{scale}} \otimes P^i_{\text{delta}} \sim Bernoulli\left(1 - (1-q)^{\eta_{\text{scale}}}\right)\,.$$
- For a set-of-strings parameter with cardinality c, $\otimes$ is, again, the point-wise lifting of $\otimes$ to c-dimensional random vectors.

Observe that the refinement of P^i_{delta} as per Eq. (4) features the following *quantitative* properties: (i) When $\eta_{\text{scale}} < 1$, i.e., more than a half of analyses fail, we have $E[\eta_{\text{scale}} \otimes P^i_{\text{delta}}] < E[P^i_{\text{delta}}]$, meaning that the scope of exploration is "shrunk" in expectation in the next iteration; (ii) Otherwise if $\eta_{\text{scale}} > 1$, we have $E[\eta_{\text{scale}} \otimes P^i_{\text{delta}}] > E[P^i_{\text{delta}}]$, meaning that Parf will "extend" the scope of exploration in expectation.

Incrementality and Adaptivity. By means of separately representing and refining the distributions P_{base} and P_{delta}, Parf features incrementality and adaptivity. The former guarantees that "rewardless analyses with low-precision parameters do not occur", since we have $P_{\text{base}} \sqsubseteq P'_{\text{base}}$. The latter allows for an adaptive and quantitative control of the exploration scope, thus avoiding analysis failures while enabling effective search of high-precision parameters.

Remark. Parf is not confined to Frama-C/Eva; rather, it can be readily integrated with any static analyzer exhibiting monotonicity over the parameters. This is because (i) Parf treats the static analyzer as a black box, and (ii) Parf covers a wide range of parameter types commonly used in static analyzers. Moreover, we can extend Parf without substantial changes to incorporate extra types of parameters, e.g., real-valued parameters, by formulating a latticed parameter space with Gaussian distributions. As an example, we interface Parf with the static analyzer Mopsa [16] in Section 5.

Moreover, due to its black-box nature, Parf exhibits no strong correlation with abstract interpretation and thus can be paired with other static analysis techniques. We opt for abstract interpretation as it is a typical analysis technique featuring monotonicity. ◁

5 IMPLEMENTATION AND EXPERIMENTS

The experiments aim to answer the following research questions:

RQ1: How does Parf compare against other parameter-selecting strategies?
RQ2: How does Parf perform on different hyper-parameters?
RQ3: Can Parf be generalized to other static analyzers?
RQ4: Can Parf improve Frama-C in verification competitions?

To answer **RQ1**, **RQ2**, and **RQ4**, we implement Parf as a plugin of Frama-C, which is an open-source collaborative platform dedicated to scalable source-code analysis of C programs [22]. For **RQ3**, we further integrate Parf with the static analyzer Mopsa [16]. Parf supports parallelization across multiple processes: We adopt a four-process parallelization mechanism for **RQ1**, **RQ2**, and **RQ3**, and a single process for **RQ4**.[3] The experiments for **RQ1**, **RQ2**, and **RQ4** are conducted on an 8-core Apple M2 processor with 16GB RAM running 64-bit macOS Sonoma 14; **RQ3** is evaluated on a 16-core Intel i7 processor with 16GB RAM running Arch Linux (as Mopsa [16] runs only on Linux). Experimental results reported in this paper can be found in the Docker image at https://hub.docker.com/r/parfdocker/parf.

5.1 Experimental Settings

Settings of Parf. Parf provides an optional interface for setting the initial parameter distribution, which allows users to provide a prior probability distribution P_{init} based on their own experience. In absence of user-specified P_{init}, Parf sets P_{base} and P_{delta} to default values (see [32, Table 1]). In our experiments, we use such default initial distribution and set the hyper-parameters α, β, num_{sample}, and num_{refine} in Algorithm 1 to 0.1, 2, 4, and 7, respectively. We examine the effect of these hyper-parameters in **RQ2**.

Benchmarks. The first benchmark suite we use is the Frama-C official Open Source Case Study (OSCS) benchmarks[4] [3]. It includes many real-world C projects, such as the X509 parser project (a Frama-C-verified static analyzer) [13, 33]. Table 1 lists the detailed information about the benchmarks as described below:

- Benchmark name: Name of each benchmark in C.
- LOC (lines of code): Size of each benchmark's source files.
- #statements: The number of statements analyzed by Frama-C in each benchmark. A statement is the smallest syntactic functional element of the target program [8].
- `-eva-precision`: This index reflects how fast each benchmark can be analyzed. A larger number indicates that the analysis can terminate in less time under the same parameter settings; see details in Section 5.2.

The second suite is collected from the verification tasks of SV-COMP 2022 [2, 6, 7], where Frama-C participated in the NoOverflows [1] category with a specific version called Frama-C-SV.

Baselines and Time Budget. We compare our approach Parf against three baselines: Default, Official, and Expert, which correspond to three existing parameter-selecting strategies. The Default strategy uses default parameter settings of Frama-C/Eva or Mopsa to perform static analysis; The Official strategy uses official parameter settings provided by Frama-C together with the OSCS benchmarks, which can be considered as "high quality" parameter settings; Expert is a dynamic parameter-tuning strategy for Frama-C/Eva, which sequentially increases the parameters from `-eva-precision 0` to `-eva-precision 11` for analysis until the given time budget is exhausted or the highest precision level is reached. We set the total time budget for each benchmark to 1 hour.

[3]For the latter, parallelization does not make a difference for SV-COMP as it counts the total CPU time for all processes.
[4]We filtered out several projects which contain, e.g., configuration issues or test suites.

5.2 RQ1: Parf vs. Other Strategies

Table 1 shows our experimental results in response of **RQ1**. Let us first illustrate the evaluation method for analysis results of the four parameter-selection strategies: Default, Expert, Official, and Parf. We mark the result with the least number of alarms and not tied with the other three as the **exclusively best**. For instance, the analysis result of the Parf strategy for the first benchmark 2048 has **4** alarms. We also mark the results with the same least number of alarms as tied-best. Given that some analysis results contain thousands of alarms, it is unfair and unreasonable to distinguish two results based on minor differences in the number of alarms. Therefore, we mark multiple sets of results with $\leq 1\%$ difference in the number of alarms from the least one as tied-best. For instance, the Expert and Parf analysis results for benchmark miniz-ex1 contain 1832 and 1828 alarms as tied-best respectively ($1832 - 1828 \leq 0.01 \times 1832$). We adopt the same convention for **RQ3**.

Table 1 shows that Parf significantly outperforms other strategies for Frama-C in terms of accuracy on the OSCS dataset: Overall, Parf achieves the best results (i.e., exclusively best or tied-best) on 34/37 (91.9%) benchmarks, while the Default, Expert, and Official strategies achieve the best results on 3/37, 23/37, and 8/37 benchmarks, respectively. Furthermore, over 12/37 (32.4%) benchmarks, Parf achieves exclusively best results, while the competitors achieve exclusively best results only on 0 or 1 specific benchmark.

The above results exhibit the capacity of Parf to achieve high-accuracy analysis results in various real-world scenarios, rather than specific scenarios. This is because the OSCS benchmarks possess diversity and representativeness, with project sizes ranging from 338 to 51,007 LOC and numbers of analyzed statements ranging from 36 to 13,029. Moreover, the values in the `-eva-precision` column intuitively reflect the analysis complexity for a given benchmark, as they are the highest `-eva-precision` parameter values identified by Expert strategy within 1 hour. Therefore, benchmarks with lower complexity require less analysis time under the same parameters, resulting in higher values in the `-eva-precision` column within the given time. Meanwhile, the range of `-eva-precision` also indicates the diversity of the OSCS benchmarks.

Interestingly, among around half (18/37) of the OSCS benchmarks with high `-eva-precision` values (≥ 9), Parf achieves tied-best results in almost all cases, with only one exclusively best (benchmark tutorials). Considering that Parf achieves the exclusively best results on 12/37 (32.4%) benchmarks, Parf performs exclusively best on 57.9% of the remaining 19 benchmarks with low-to-medium `-eva-precision` values (≤ 8). The reason behind this observation is as follows. On one hand, benchmarks with high `-eva-precision` values have low analysis complexity, allowing various strategies to readily find high-accuracy or even the most accurate analysis results

Table 1: Experimental results in terms of RQ1 and RQ3.

OSCS Benchmark Details				#Alarms of Frama-C (RQ1)				#Alarms of Mopsa (RQ3)	
Benchmark name	LOC	#statements	-eva-precision	Default	Expert	Official	Parf	Default	Parf
2048	440	329	6	7	5	7	**4**	141	**67**
chrony	37177	41	11	9	7	8	7	–	–
debie1	8972	3243	2	33	3	**1**	19	8245	**5656**
genann	1183	1042	9	236	69	77	69	1308	1308
gzip124	8166	4835	1	802	885	866	810	–	–
hiredis	7459	87	11	9	0	9	0	43	43
icpc	1302	424	11	9	1	1	1	11	**10**
jsmn-ex1	1016	1219	11	58	1	1	1	1762	**1253**
jsmn-ex2	1016	311	11	68	1	1	1	87	**86**
kgflags-ex1	1455	474	11	11	0	11	0	280	280
kgflags-ex2	1455	736	10	33	19	33	19	386	386
khash	1016	206	11	14	2	14	2	19	19
kilo	1276	1078	2	523	445	688	**429**	5299	5290
libspng	4455	2377	6	186	122	122	**113**	–	–
line-following-robot	6739	857	11	1	1	1	1	–	–
microstrain	51007	3216	6	1177	616	646	**598**	6237	6196
mini-gmp	11706	628	6	83	71	83	71	513	**491**
miniz-ex1	10844	3659	1	2291	1832	2291	1828	3020	3004
miniz-ex2	10844	5589	1	2742	2220	2742	**2172**	3916	3899
miniz-ex3	10844	3747	1	577	552	577	**442**	2808	2792
miniz-ex4	10844	1246	4	258	217	258	**189**	162	162
miniz-ex5	10844	3430	1	425	402	425	**377**	1575	**1474**
miniz-ex6	10844	2073	1	220	198	220	**173**	1197	**1075**
monocypher	25263	4126	1	606	570	568	606	TO	TO
papabench	12254	36	11	1	1	1	1	–	–
qlz-ex1	1168	229	11	68	11	68	11	82	82
qlz-ex2	1168	75	11	8	8	8	8	50	50
qlz-ex3	1168	294	8	94	82	94	**75**	–	–
qlz-ex4	1168	164	11	17	13	17	13	–	–
safestringlib	29271	13029	6	855	**256**	300	356	–	–
semver	1532	728	9	29	22	25	22	3556	**2850**
solitaire	338	396	11	216	18	213	18	700	**663**
stmr	781	500	6	63	58	59	58	1391	1391
tsvc	5610	5478	4	413	355	379	356	–	–
tutorials	325	89	11	5	1	5	**0**	–	–
tweetnacl-usable	1204	659	11	126	25	30	25	667	**657**
x509-parser	9457	3112	3	208	198	198	**187**	364	**339**
Overall (tied-best+**exclusively best**)				3/37	23/37	8/37	34/37 (91.9%)	14/27 (51.9%)	26/27 (96.3%)
Overall (**exclusively best**)				0/37	1/37	1/37	12/37 (32.4%)	0/27 (0.0%)	12/27 (44.4%)

within the given time. Therefore, Parf can only achieve tied-best results with them. On the other hand, benchmarks with low-to-medium `-eva-precision` values have high analysis complexity, making it difficult for Default, Official, and Expert strategies to avoid analysis failure and find the most accurate parameters. However, Parf's adaptivity enables it to handle such challenging tasks. In summary, this demonstrates that *Parf is particularly suitable for analyzing complex, large-scale real-world programs.*

Note that, as the computation in Parf is randomized, we repeatedly conduct two experiments with Parf, limiting the analysis time budget of each benchmark to 30 minutes in each experiment, and select the better analysis result (the same applies to **RQ3**). This is to simulate the real situation of experts using static analyzers: trying to find the best analysis result within a given total time budget. However, this cannot completely eliminate randomness; Parf still has the possibility to fail to find the best analysis result within the time limit (e.g., benchmarks debie1, monocypher, and safestringlib).

Running Time under Different Strategies. Table 2 shows the timings (averaged over the OSCS benchmarks) of (i) identifying the final parameter setting p and (ii) analyzing the source program under p. As formulated in Section 3, the application scenario of our approach is to automatically find a highly accurate parameter setting within a given time budget. In this context, we find it less interesting to compare the *analysis time under the specific parameter setting* produced by different strategies (bottom row of Table 2), since parameters of higher precision typically require more time to perform static analysis. In contrast, we are concerned with the *time for identifying high-precision parameter settings* (first row of Table 2), where (i) The Default strategy provides low-precision parameters in no time but suffers from significantly low accuracy; (ii) The Official strategy yields high-precision parameters prepared by Frama-C officially for the OSCS benchmarks, but it does not specify the human effort and time invested in identifying these parameters. This implies that manually selecting parameters is still necessary for programs beyond OSCS; (iii) The Expert strategy represents a simple yet effective automated dynamic parameter-tuning strategy employed by experts, which achieves results with significantly higher accuracy than the Default and Official strategies within a reasonable time budget (1 hour in our case); (iv) Parf identifies the most accurate parameter settings in 91.9% benchmarks only with around half of the time used by Expert (the only automated, dynamic competitor).

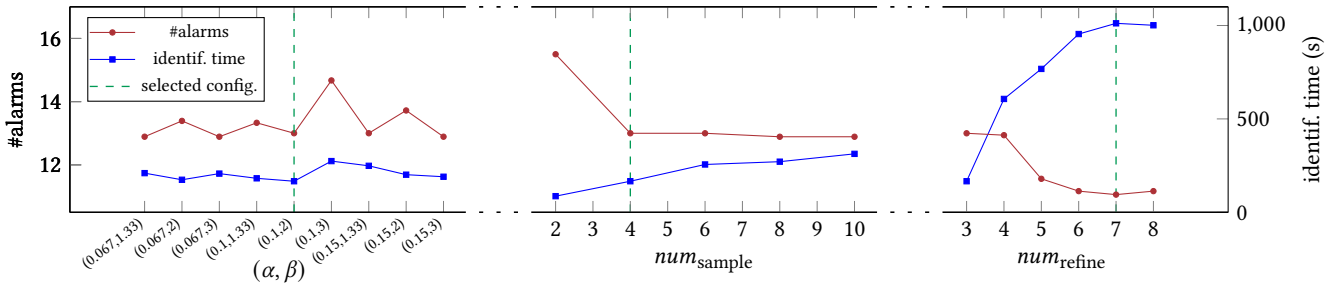

Figure 6: Effect of hyper-parameters on the performance of PARF in terms of accuracy (#alarms) and efficiency (time).

Table 2: Average timings of the experiments in terms of RQ1.

Phase	Average Time for FRAMA-C (s)			
	DEFAULT	EXPERT	OFFICIAL	PARF
Identification of p	0.00	2340.59	–	**1315.97**
Analysis under p	9.00	465.05	21.78	78.92

Remark. Our PARF strategy adopts a four-process parallelization mechanism. We remark that a similar mechanism does not apply to the other parameter-selecting strategies: (i) The parameters given by DEFAULT and OFFICIAL are fixed; (ii) For the EXPERT strategy, parallelizing the trials with different precision levels does often not bring observable enhancement in efficiency, since it is still the trial with the highest precision level that consumes significantly more time than those with lower precision levels. ◁

5.3 RQ2: Effect of Hyper-Parameters

Fig. 6 depicts the effect of hyper-parameters on PARF's performance – averaged over 18/37 OSCS benchmarks (with `-eva-precision` ≥ 9)[5] – in terms of accuracy (#alarms) and efficiency (time for identifying the final parameter setting):

- Effect of $\alpha \in (0, 1)$ and $\beta > 1$: These two hyper-parameters control the initial time budget T_r for every round of the `Sample-Analyze-Refine` process in the PARF algorithm. We observe that PARF is not sensitive to the values of α and β.
- Effect of num_{sample}: This parameter upper-bounds the number of samples in one refinement iteration. Observe that both the analysis accuracy and the time remain stable for $num_{\text{sample}} \geq 6$ (due to the time budget T_r per iteration). In our other experiments, we set $num_{\text{sample}} = 4$ due to our 4-process parallelization; $num_{\text{sample}} \geq 6$ yields similar accuracy with slightly more time.
- Effect of num_{refine}: This parameter upper-bounds the number of refinement iterations. We observe similar performance stability for $num_{\text{refine}} \geq 6$. In our other experiments, we set $num_{\text{refine}} = 7$ to a relatively large number to make full use of the time budget for more refinement iterations; $num_{\text{refine}} = 5$ yields higher efficiency with similar accuracy.

Overall, we do not observe significant sensitivity of PARF to these hyper-parameters (except for small values of num_{sample}, num_{refine}).

[5]This subset of benchmarks features a relatively low analysis complexity and thus our evaluation of **RQ2** can be completed in a reasonable amount of time.

Table 3: SV-COMP verification results in terms of RQ4. A more detailed explanation is provided in [32, Section 5.5].

Setting	Verification Result				Score
	correct	wrong	unknown	failure	
FRAMA-C-SV$_{\text{precision11}}$	146	3	272	33	186
FRAMA-C-SV$_{\text{PARF}}$	**151**	3	300	**0**	**196**

The specific configuration of hyper-parameters (marked by dashed lines in Fig. 6) in our other experiments is not finely tuned.

5.4 RQ3: Generality of PARF

In this experiment, we demonstrate the generality of PARF by interfacing it with another static analyzer MOPSA [16]. The parameters considered for MOPSA and their distributions are described in [32, Table 4]. We use the same hyper-parameter values as for FRAMA-C/EVA except that we set $num_{\text{refine}} = 3$ to ensure that most analyses can terminate within the time budget (since MOPSA runs much slower than FRAMA-C/EVA on the OSCS benchmarks).

The last two columns of Table 1 demonstrate how our PARF strategy improves the performance of MOPSA.[6] Observe that MOPSA fails to analyze some OSCS benchmarks which either run out of time (marked by TO) or have language features, data types, and/or symbols not supported by MOPSA (marked by –). For the rest benchmarks, PARF achieves the best results on 26/27 (96.3%) program repositories with exclusively best results on 12/27 (44.4%) cases, which significantly outperforms MOPSA's DEFAULT strategy.

Overall, this experiment demonstrates that PARF can be generalized to improve the performance of other static analyzers.

5.5 RQ4: Improving FRAMA-C in SV-COMP

Table 3 shows that PARF can slightly improve the performance of FRAMA-C in SV-COMP. Since the analysis resource for each verification task is limited to 15 minutes of CPU time, FRAMA-C-SV$_{\text{precision11}}$ strategy uses a fixed highest `-eva-precision 11` parameter for analysis. We set the experimental strategy of FRAMA-C-SV$_{\text{PARF}}$ as: (i) identifying the final parameter setting p via PARF within 7.5 minutes, and (ii) analyzing under p within the left 7.5 minutes.

[6]The OFFICIAL and EXPERT strategies are unavailable in this experiment since MOPSA provides neither official parameter settings for OSCS benchmarks, nor built-in precision levels (like `-eva-precision` in FRAMA-C/EVA) for applying the EXPERT strategy.

The experimental results show that Parf can eliminate all analysis failures and successfully verify 5 more tasks, thus slightly improving the total score from 186 to 196. This is because (i) Most analyses using `-eva-precision 11` parameter of the verification tasks under the NoOverflows category can terminate within 15 minutes, as discussed in Section 5.2, so it is difficult for Parf strategy to obtain more accurate analysis results; (ii) Parf adaptively finds 5 high-accuracy analysis results among 33 Frama-C-SV$_{\text{precision11}}$ analysis failures, thus successfully verifying these 5 tasks.

6 LIMITATIONS AND FUTURE WORK

We pinpoint several scenarios for which the proposed Parf framework is inadequate and provide potential solutions thereof.

First, Parf treats the underlying static analyzer as a black box. Although this feature facilitates the portability and generality of Parf (as discussed in Section 4), it does not exploit analyzer-specific functionalities and/or internal results, e.g., the data/control-flow analysis of Frama-C [20] to further improve the quality of the generated parameters. It is therefore worth investigating to what extent we can enhance parameter refining by leveraging a *white/grey-box model*, as is the use of Sparrow [26] in Bingo [27]. As an example, one may utilize the call graph of a program – as an intermediate result of a static analyzer – to achieve more fine-grained parameter tuning, e.g., setting analysis parameters for individual functions of a program (aka, the problem of *function-level refining*).

Second, Parf does not employ the syntactic or semantic characteristics of the source program, e.g., the maximum number of loop iterations. However, we foresee that machine-learning models and techniques may be developed to *learn a good parameter setting* based on such characteristics, which can then be used as the initial (distribution of) parameter setting of Parf.

Third, Parf models different parameters as independent random variables. Taking into account the dependencies between parameters is expected to reduce the search space and thereby accelerate the parameter refining process. To this end, we need to extend Parf to admit the *representation of stochastic dependencies*.

7 RELATED WORK

Adaptive Parameter Tuning. Automatic parameter-tuning techniques have witnessed numerous applications in many fields, including database [5] and big data [14, 25] systems, machine learning hyper-parameter tuning [23, 24], mathematical software [34], and symbolic execution engines [10]. Our work is dedicated to providing automated parameter-tuning support for static analyzers.

We emphasize that our Parf framework is inspired by SymTuner [10], an adaptive parameter tuning framework for symbolic executors based on a formulated discrete sample space of external parameters. It updates sampling probability according to symbolic executing behaviors. Whereas symbolic execution can stop at any time and yield useful intermediate information, a static analyzer only produces complete outcomes until the whole analysis terminates. Considering the inherent feature of static analyzer, we have designed a novel representation of probability distributions and refinement strategy to ensure the incrementality and adaptivity of the parameter probability distribution. These properties effectively avoid analysis failure and rewardless analysis.

Improving Static Analysis Tools. Many static analyzers integrated parameterization strategies, such as Astrée [18] and Goblint [29]. Kästner et al. [17] summarize the four most important abstraction mechanisms in Astrée and recommend prioritizing the accuracy of related abstract domains, which amounts to narrowing down the parameter space. However, these mechanism are not fully automated since and need directives provided by the user. Saan et al. [28, 30] implement in Goblint a simple, heuristic autotuning method based on syntactical criteria, which can automatically activate or deactivate abstraction techniques before analysis. However, this method only generates an initial analysis configuration once and does not dynamically adapt to refine the parameter configuration. Salvi et al. [31] present an approach to coupling model-based testing with static analysis based on a tool coupling between Astrée and BTC*EmbeddedTester*®; but this method cannot tune the parameters of the static analyzer.

Probability-Based Algorithms. In the past several years, researchers have presented numerous approaches to filter alarms of static analysis tools. These works leverage probabilistic methods to obtain prior knowledge of alarms, in order to determine whether a generated alarm is really caused by potential bugs instead of the upper approximation introduced by the analyzer.

Raghothaman et al. [27] proposed Bingo, a system that utilizes Bayesian inference [4, 21] to decide the confidence of each alarm. Their model is refined through ground truth labels provided by users in each iteration. Heo et al. [15] proposed Drake, a probabilistic framework to identify alarms relative to a certain program change, thus helping improve the efficacy of discovering true bugs. Several probabilistic reasoning techniques [11, 19] have recently been proposed to incorporate external feedback on semantic facts, thereby reducing user efforts on alarm inspection.

Different from our task, the above works mainly focus on checking the alarm list and determining true positives. They only run the static analyzer once and conduct no adjustments on input parameters, though both tasks aim to optimize the final analysis results.

8 CONCLUSION

We have presented a novel framework called Parf for adaptively tuning external parameters of abstract interpretation-based static analyzers. Parf is – to the best of our knowledge – the first *fully automated* approach that supports *incremental* refinement of such parameters. The effectiveness of Parf has been demonstrated on a collection of standard benchmarks. Future directions include extending Parf to cope with function-level refining and dependencies between parameters, and encoding the algorithms into a probabilistic programming paradigm using Bayesian inference [4, 21].

ACKNOWLEDGMENTS

This work has been partially funded by the ZJNSF Major Program (No. LD24F020013), by the CCF-Huawei Populus Grove Fund (No. CCF-HuaweiFM202301), by the Fundamental Research Funds for the Central Universities of China (No. 226-2024-00140), and by the ZJU Education Foundation's Qizhen Talent program. The authors would like to thank Shenghua Feng for helpful discussions and the anonymous reviewers for their constructive feedback.

REFERENCES

[1] 2022. Benchmark Verification Tasks of SV-COMP 2022. https://sv-comp.sosy-lab.org/2022/benchmarks.php. Accessed: 2024-06-08.

[2] 2022. Collection of SV-COMP 2022 Verification Tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks. Accessed: 2024-06-08.

[3] 2024. Open source case studies for Frama-C. https://git.frama-c.com/pub/open-source-case-studies. Accessed: 2024-06-08.

[4] Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2019. On the Computability of Conditional Probability. *J. ACM* 66, 3 (2019), 23:1–23:40.

[5] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD Conference*. ACM, 1009–1024.

[6] Dirk Beyer. 2022. Progress on Software Verification: SV-COMP 2022. In *TACAS (2)*. Lecture Notes in Computer Science, Vol. 13244. Springer, 375–402.

[7] Dirk Beyer and Martin Spiessl. 2022. The Static Analyzer Frama-C in SV-COMP (Competition Contribution). In *TACAS (2) (Lecture Notes in Computer Science, Vol. 13244)*. Springer, 429–434.

[8] David Bühler. 2017. *Structuring an Abstract Interpreter through Value and State Abstractions:EVA, an Evolved Value Analysis for Frama-C. (Structurer un interpréteur abstrait au moyen d'abstractions de valeurs et d'états :Eva, une analyse de valeur évoluée pour Frama-C)*. Ph. D. Dissertation. University of Rennes 1, France.

[9] David Bühler, Pascal Cuoq, and Boris Yakobowski. [n. d.]. *Eva - the Evolved Value Analysis Plug-In*.

[10] Sooyoung Cha, Myungho Lee, Seokhyun Lee, and Hakjoo Oh. 2022. SYMTUNER: Maximizing the Power of Symbolic Execution by Adaptively Tuning External Parameters. In *ICSE*. ACM, 2068–2079.

[11] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *ESEC/SIGSOFT FSE*. ACM, 1154–1165.

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[13] Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. 2019. Journey to a RTE-free X. 509 parser. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC 2019)*.

[14] Anastasios Gounaris, Georgia Kougka, Rubén Tous, Carlos Tripiana Montes, and Jordi Torres. 2017. Dynamic Configuration of Partitioning in Spark Applications. *IEEE Trans. Parallel Distributed Syst.* 28, 7 (2017), 1891–1904.

[15] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously reasoning about programs using differential Bayesian inference. In *PLDI*. ACM, 561–575.

[16] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2019. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In *VSTTE (Lecture Notes in Computer Science, Vol. 12031)*. Springer, 1–18.

[17] Daniel Kaestner, Stephan Wilhelm, Christoph Mallon, Stefana Schank, Christian Ferdinand, and Laurent Mauborgne. 2023. *Automatic sound static analysis for integration verification of AUTOSAR software*. Technical Report. SAE Technical Paper.

[18] Daniel Kästner, Reinhard Wilhelm, and Christian Ferdinand. 2023. Abstract Interpretation in Industry - Experience and Lessons Learned. In *SAS (Lecture Notes in Computer Science, Vol. 14284)*. Springer, 10–27.

[19] Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning probabilistic models for static analysis alarms. In *Proceedings of the 44th International Conference on Software Engineering*. 1282–1293.

[20] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects Comput.* 27, 3 (2015), 573–609.

[21] Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024. Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 127 (2024), 31 pages.

[22] Nikolai Kosmatov and Julien Signoles. 2016. Frama-C, A Collaborative Framework for C Code Verification: Tutorial Synopsis. In *RV (Lecture Notes in Computer Science, Vol. 10012)*. Springer, 92–115.

[23] Yang Li, Yu Shen, Huaijun Jiang, Wentao Zhang, Jixiang Li, Ji Liu, Ce Zhang, and Bin Cui. 2022. Hyper-Tune: Towards Efficient Hyper-parameter Tuning at Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1256–1265.

[24] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *J. Mach. Learn. Res.* 23 (2022), 54:1–54:9.

[25] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. 2019. Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems. *Proc. VLDB Endow.* 12, 12 (2019), 1970–1973.

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

[27] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *PLDI*. ACM, 722–735.

[28] Simmo Saan, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. 2024. Goblint: Abstract Interpretation for Memory Safety and Termination - (Competition Contribution). In *TACAS (3) (Lecture Notes in Computer Science, Vol. 14572)*. Springer, 381–386.

[29] Simmo Saan, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. 2021. Goblint: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints - (Competition Contribution). In *TACAS (2) (Lecture Notes in Computer Science, Vol. 12652)*. Springer, 438–442.

[30] Simmo Saan, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. 2023. Goblint: Autotuning Thread-Modular Abstract Interpretation - (Competition Contribution). In *TACAS (2) (Lecture Notes in Computer Science, Vol. 13994)*. Springer, 547–552.

[31] Sayali Salvi, Daniel Kästner, Tom Bienmüller, and Christian Ferdinand. 2014. True Error or False Alarm? Refining Astrée's Abstract Interpretation Results by Embedded Tester's Automatic Model-Based Testing. In *SAFECOMP Workshops (Lecture Notes in Computer Science, Vol. 8696)*. Springer, 84–96.

[32] Zhongyi Wang, Linyu Yang, Mingshuai Chen, Yixuan Bu, Zhiyang Li, Qiuye Wang, Shengchao Qin, Xiao Yi, and Jianwei Yin. 2024. Parf: Adaptive Parameter Refining for Abstract Interpretation. *CoRR* abs/2409.05794 (2024). https://doi.org/10.48550/arXiv.2409.05794

[33] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models using Static Analysis and Program Verification. *CoRR* abs/2404.00762 (2024).

[34] Mengyuan Zhang, Wotao Yin, Mengchang Wang, Yangbin Shen, Peng Xiang, You Wu, Liang Zhao, Junqiu Pan, Hu Jiang, and KuoLing Huang. 2023. MindOpt Tuner: Boost the Performance of Numerical Software by Automatic Parameter Tuning. *CoRR* abs/2307.08085 (2023).

[35] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 954–982.