Wang ZY, Chen MS, Lin TJ *et al.* Parf: An adaptive abstraction-strategy tuner for static analysis. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY. DOI: 10.1007/s11390-025-5140-6, CSTR: 32374.14.s11390-025-5140-6

Parf: An Adaptive Abstraction-Strategy Tuner for Static Analysis

Zhong-Yi Wang[1] (王钟逸), Ming-Shuai Chen[1,*] (陈明帅), *Senior Member, CCF*, Teng-Jie Lin[1] (林滕劼)
Lin-Yu Yang[1] (杨麟禹), Jun-Hao Zhuo[1] (卓俊豪), Qiu-Ye Wang[2] (王秋野), Sheng-Chao Qin[3] (秦胜潮)
Xiao Yi[2] (伊　晓), and Jian-Wei Yin[1] (尹建伟), *Senior Member, CCF*

[1] *College of Computer Science and Technology, Zhejiang University, Hangzhou 310012, China*
[2] *Fermat Labs, Huawei Inc., Dongguan 523000, China*
[3] *Guangzhou Institute of Technology, Xidian University, Guangzhou 510000, China*

E-mail: wzygomboc@zju.edu.cn; m.chen@zju.edu.cn; tengjie.lin@zju.edu.cn; linyu.yang@zju.edu.cn; jhzhuo@zju.edu.cn
wangqiuye2@huawei.com; shengchao.qin@gmail.com; yi.xiao1@huawei.com; zjuyjw@zju.edu.cn

Received December 31, 2024; accepted June 5, 2025.

Abstract We launch Parf — a toolkit for adaptively tuning abstraction strategies of static program analyzers in a fully automated manner. Parf models various types of external parameters (encoding abstraction strategies) 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 abstraction strategies. Parf is implemented on top of Frama-C/Eva — an off-the-shelf open-source static analyzer for C programs. Parf provides a web-based user interface facilitating the intuitive configuration of static analyzers and visualization of dynamic distribution refinement of the abstraction strategies. It further supports the identification of dominant parameters in Frama-C/Eva analysis. Benchmark experiments and a case study demonstrate the competitive performance of Parf for analyzing complex, large-scale real-world programs.

Keywords automatic parameter tuning, Frama-C/Eva, program verification, static analysis, abstraction strategy

1 Introduction

Static analysis is the process of analyzing a program without ever 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. Identifying an appropriate abstraction strategy — for soundly approximating the concrete semantics — is a crucial task to obtain a delicate trade-off between the accuracy and efficiency of static analysis: a finer abstraction strategy may yield fewer false alarms (i.e., approximation-caused alarms that do not induce RTEs) yet typically incurs less efficient analysis. State-of-the-art sound static analyzers, such as Frama-C/Eva[1], Astrée[2], Goblint[3], and Mopsa[4], integrate abstraction strategies encoded by various external parameters, thereby enabling analysts to balance accuracy and efficiency by tuning these parameters.

Albeit with the extensive theoretical study of sound static analysis[5, 6], the picture is much less clear on its parameterization front[7]: 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. 1) 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 Table 1

Regular Paper

Special Section of ChinaSoft2024—Prototype

This work was supported by the Zhejiang Provincial Natural Science Foundation Major Program under Grant No. LD24F020013, the CCF-Huawei Populus Grove Fund under Grant No. CCF-HuaweiFM202301, the Fundamental Research Funds for the Central Universities of China under Grant No. 226-2024-00140, and the Zhejiang University Education Foundation's Qizhen Talent Program.

*Corresponding Author

Table 1. Parameter Settings in FRAMA-C/EVA

Parameter	Type	Value Space
`min-loop-unroll`	Integer	$\mathbb{N}$
`auto-loop-unroll`	Integer	$\mathbb{N}$
`widening-delay`	Integer	$\mathbb{N}$
`partition-history`	Integer	$\mathbb{N}$
`slevel`	Integer	$\mathbb{N}$
`ilevel`	Integer	$\mathbb{N}$
`plevel`	Integer	$\mathbb{N}$
`subdivide-non-linear`	Integer	$\mathbb{N}$
`split-return`	String	{"", "auto"}
`remove-redundant-alarms`	Boolean	$\{\text{false}, \text{true}\}$
`octagon-through-calls`	Boolean	$\{\text{false}, \text{true}\}$
`equality-through-calls`	String	{"none", "formals"}
`domains`	Set-of-strings	$\{\text{false}, \text{true}\}^5$

Note: For the set-of-strings parameter `domains` with $|\texttt{domains}| = 5$, its value space is the Cartesian product $\{\text{false}, \text{true}\}^5$.

consists of 13 external parameters that are highly relevant to the accuracy and efficiency of FRAMA-C/EVA, among which eight integer parameters have infinite value spaces. 2) 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, few static analyzers provide a fully automated approach to guiding the refinement of abstraction strategies based on such information. Therefore, the use of sound static analysis tools still relies heavily on expert knowledge and experience.

Some advanced static analyzers attempt to address the above challenges using various methods. FRAMA-C/EVA[1] provides the meta option `-eva-precision`, which packs a predefined group of valuations to the parameters listed in Table 1, thus enabling a quick setup of the analysis. Kästner *et al.*[8] summarized the four most important abstraction strategies in Astrée and recommended prioritizing the accuracy of related abstract domains, which amounts to narrowing down the parameter space. However, both FRAMA-C/EVA and Astrée currently do not support automatic parameter generation. GOBLINT[3] 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. See Section 6 for detailed related work.

Following this line of research, we have presented PARF[9], an adaptive and fully automated parameter refining framework for sound 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 abstraction strategies by incrementally refining the probability distributions based on accumulated intermediate results generated via repeatedly sampling and analyzing. Preliminary experiments have demonstrated that PARF outperforms state-of-the-art parameter-tuning mechanisms by discovering abstraction strategies leading to more accurate analysis, particularly for programs of a large scale.

Contributions. This article presents the PARF artifact, whose theoretical underpinnings have been established in [9]. We focus on the design, implementation, and application of the PARF toolkit and make—in position to [9]—the following new contributions.

1) We present design principles underneath the novel abstraction-strategy tuning architecture PARF for establishing provable incrementality (monotonic knowledge retention) and adaptivity (resource-aware exploration) to achieve accuracy-efficiency trade-offs in static analysis parameterization.

2) We develop a web-based user interface (UI) for PARF which facilitates the intuitive configuration of static analysis and visualizes the dynamic distribution refinement of abstraction strategies.

3) We show via a post-hoc analysis that PARF supports the identification of the most influential parameters dominating the accuracy-efficiency trade-off.

4) We demonstrate through a case study how PARF can help eliminate false alarms and, in some cases, certify the absence of RTEs.

2 Problem and Methodology

This section revisits the problem of abstraction-strategy tuning and outlines the general idea behind our PARF framework. More technical details are in [9].

In abstraction-strategy tuning, a static analyzer is modeled as a function $\mathit{Analyze}\colon (\mathit{prog}, p) \mapsto A_p$, which receives a target program prog and a parameter setting p (encoding an abstraction strategy of the analyzer) and returns a set A_p of RTE alarms emitted under p[9]. We assume, as is the case in most state-of-the-art static analyzers[10], that the analyzer exhibits monotonicity over parameters, i.e., an abstraction strategy of higher precision (in an ordered joint parameter space) induces fewer alarms and thereby more accurate analysis.

The abstraction-strategy tuning problem can be formally defined as: given a target program $prog$, a time budget $T \in \mathbb{R}_{>0}$, a static analyzer $Analyze$, and the joint space of parameter settings S of $Analyze$, find a parameter setting $p \in S$ such that $Analyze(prog, p)$ returns as few alarms as possible within T[9].

Our Parf framework[9] addresses the problem as follows. It models external parameters of the static analyzer as random variables subject to probability distributions over parameter spaces equipped with complete lattice structures. 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. More concretely, Parf adopts a multi-round iterative mechanism. In each iteration, Parf 1) repeatedly samples parameter settings based on the initial or refined probability distribution of parameters, and then 2) uses these parameter settings as inputs to the static analyzer to analyze the program, and finally 3) 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.

The core technical challenge lies in designing the representation of probability distributions over latticed parameter spaces and the iterative refinement mechanism that jointly enforce 1) incrementality: rewardless analyses (i.e., no new false alarms are eliminated) with low-precision parameters do not occur; and 2) adaptivity: analysis failures can be avoided while enabling the effective search of high-precision parameters. Specifically, we model each parameter as a combination of dual random variables (P_{base} and P_{delta}) with type-specific initialization. Then we design P_{base} and P_{delta} stratified refinement strategies, respectively, which guarantees: 1) incremental P_{base} expectation to preserve the accumulated knowledge during the iterative procedure, and 2) adaptive P_{delta} expectation scaling to balance trade-offs of exploring the uncharted parameter space and high-precision analysis resource costs. Related details are illustrated in Section 3.

Regarding implementation, we are primarily concerned with the open-source static analyzer Frama-C/Eva[1] for C programs. However, since Parf treats the underlying analyzer as a black-box function, it can be integrated with any static analyzer exhibiting monotonicity (e.g., Mopsa[4] as shown in [9]).

3 Parf Architecture

This section elaborates on Parf artifact that im-

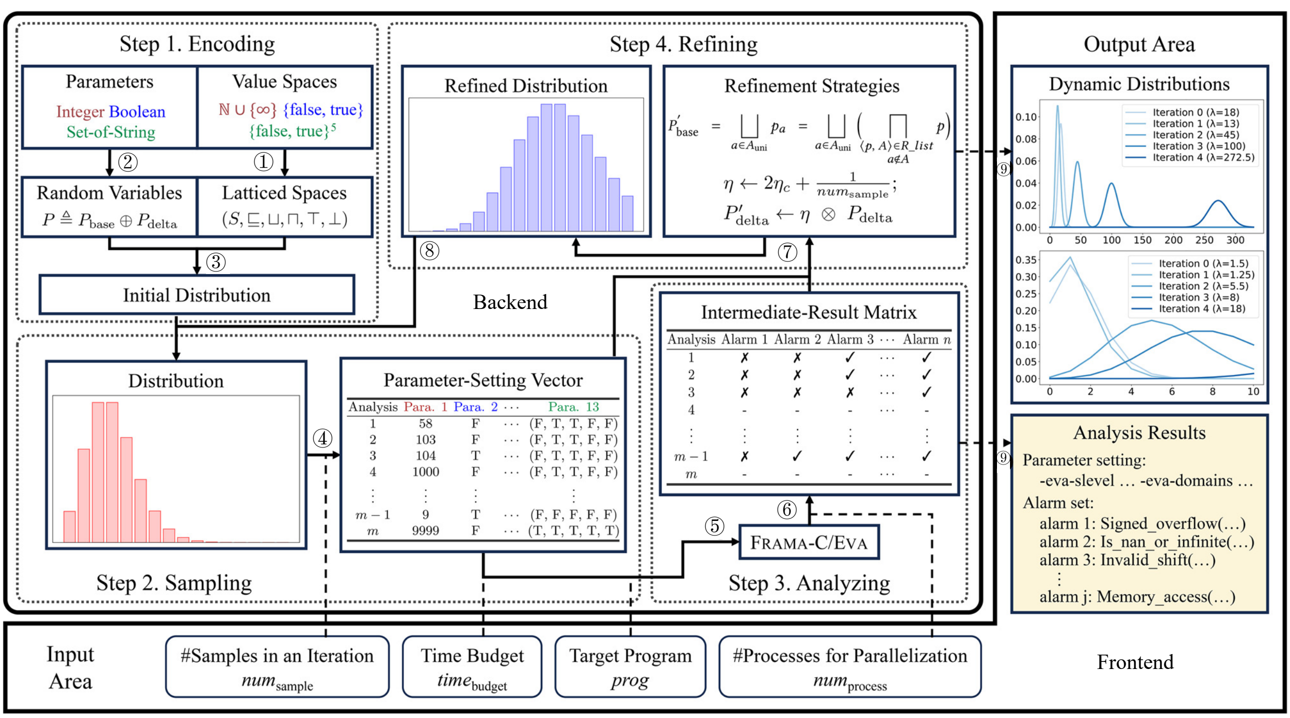


Fig.1. Architecture of the Parf artifact.

plements the aforementioned techniques. As depicted in Fig.1, the artifact is composed of two components: the backend tuning algorithm and the frontend web UI. The former comprises about 1 500 lines of OCaml code and the latter is built using Next.js and TypeScript.

The workflow of the backend tuning algorithm comprises four main steps.

1) *Value-Space Encoding* (Subsection 3.1). Parf encodes the value spaces of parameters as sample spaces with complete lattice structures (①. Meanwhile, it models external parameters of Frama-C/Eva as random variables subject to probability distributions over those latticed spaces (②). This step aims to initialize the parameter distribution (③), which serves as the basis for subsequent sample-analyze-refine iterations.

2) *Parameter Sampling* (Subsection 3.2). Parf repeatedly samples (④) parameter settings as per either the initial distribution (from step 1) or the refined distribution (from step 4). The number of samples is determined by a user-defined hyper-parameter num_{sample}.

3) *Program Analyzing* (Subsection 3.3). Using the parameter settings generated in step 2, Parf performs static analysis (⑤) on the target program *prog* via Frama-C/Eva within the given time budget $time_{\text{budget}}$. The artifact supports parallelization and thus allows multiple analyses to be conducted simultaneously (⑥). Once the time budget for this step is exhausted, Parf collects the intermediate results (e.g., the termination conditions and reported alarms) from each analysis and proceeds to the next step.

4) *Distribution Refining* (Subsection 3.4). Parf utilizes the intermediate results to refine the probability distribution (⑦). It then returns to step 2, using the updated distribution as input (⑧).

The frontend web-based UI enables intuitive and flexible interaction between users and the backend for, e.g., uploading target programs and configuring hyper-parameters (num_{sample}, $time_{\text{budget}}$, and num_{process}). Moreover, the UI visualizes the dynamic evolution of parameter distributions during the analysis and displays the final analysis results along with the corresponding abstraction strategy (⑨). Videos on accessing and using the UI are available online[①②]. Below, we explain each function module of the artifact in detail.

3.1 Value-Space Encoding

Table 1 lists 13 parameters encoding Frama-C/Eva's abstraction and analysis strategies, categorized into four types with value spaces defined by their type: 1) integer parameters range over $\mathbb{N}$; 2) Boolean parameters have a value space of {false, true}; 3) string parameters have a value space defined as a set of strings; and 4) `domains`, a unique set-of-strings parameter takes values from a power set of five abstract domains, namely, {"cvalues", "octagon", "equality", "gauges", "symbolic-locations"}. A value of `domains` can be represented as a quintuple consisting of true or false, indicating whether the corresponding domain is enabled. For example, the quintuple (false, false, true, true, false) corresponds to {"equality", "gauges"}. Thus, the value space of `domains` is $\{\text{false}, \text{true}\}^5$.

Parf encodes the value spaces of parameters into latticed sample spaces, represented as $(S, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, where S denotes the value space of a parameter, $\sqsubseteq$ is the partial order over S, $\sqcup$ denotes the *join* (aka the least upper bound) operator, $\sqcap$ denotes the *meet* (aka the greatest lower bound) operator, $\top$ and $\bot$ stand for the greatest and least element in S, respectively. Table 2 instantiates these symbols for each parameter type. Note that the two string-typed parameters of Frama-C/Eva (cf. Table 1) have only two possible values corresponding to two abstraction strategies with different precision levels, thus allowing us to treat them as Boolean-typed parameters.

Table 2. Latticed Sample Spaces of Different Parameter Types

Type	$a \sqsubseteq b$	$a \sqcup b$	$a \sqcap b$	$\top$	$\bot$
Integer	$a \leqslant b$	$\max(a, b)$	$\min(a, b)$	∞	0
Boolean	$a \Rightarrow b$	$a \vee b$	$a \wedge b$	true	false
Set-of-strings	$a \subseteq b$	$a \cup b$	$a \cap b$	U	$\emptyset$

Note: the elements a and b in each row are of their respective type, e.g., $a = 2, b = 5$ for integer parameters, $a = \text{false}, b = \text{true}$ for Boolean parameters, and $a = \{\text{"equality"}\}, \text{b} = \{\text{"equality"}, \text{"gauges"}\}$ for set-of-strings parameters.

[①] https://doi.org/10.528 1/zenodo.13934703, Jun. 2025.
[②] https://jcst.ict.ac.cn/news/366, Jul. 2025.

The lattice structure of the sample spaces serves as the basis of the distribution refinement mechanism described in Subsection 3.4.

PARF models each parameter as a composite random variable P in the novel form of:

$$P \triangleq P_{\text{base}} \oplus P_{\text{delta}}, \tag{1}$$

where P_{base} is a base random variable for retaining the accumulated knowledge during the iterative analysis whilst P_{delta} is a delta random variable for exploring the parameter space; they share the same sample space and range with P. P_{base} follows Dirac distributions, i.e., $\Pr[P_{\text{base}} = p] = 1$ for some sample $p \in S$; P_{delta} adopts different types of distributions as per the parameter type: we use Bernoulli distributions for Boolean-typed parameters and Poisson distributions for integer-typed parameters (since the latter naturally encodes infinite-support discrete distributions over $\mathbb{N}$). The constructions of P by combining P_{base} and P_{delta} via the operator $\oplus$ is given in Table 3.

Remark. Determining appropriate distributions for P_{delta} presents a technical challenge. Boolean parameters naturally match Bernoulli distributions due to their binary support set. For integer parameters, we utilize Poisson distributions for two key reasons: 1) the Poisson distributions offers an infinite support set that aligns well with the nature of integers, and 2) the Poisson distributions is characterized by a unique parameter λ.

3.2 Parameter Sampling

PARF repeatedly samples values for each parameter represented as a random variable P following the composite distribution as in (1). For instance, the initial distributions employed by the artifact are collected in online appendix[3].

To generate a sample point p for parameter P, PARF first draws samples p_{base} and p_{delta} independently from the distributions of P_{base} and P_{delta}, respectively, and then applies the binary operation $\oplus$ to construct the sampled value for P, i.e., $p = p_{\text{base}} \oplus p_{\text{delta}}$ (see Table 3). Subsequently, PARF aggregates the sampled values of all parameters into a complete analysis configuration. The total number of generated configurations in a sample-analyze-refine iteration is controlled by the user-defined hyper-parameter num_{sample}. All the configurations are maintained in an internal list structure.

3.3 Program Analyzing

In this step, PARF performs static analysis on the target program *prog* leveraging FRAMA-C/EVA. The analyses pertaining to the num_{sample} parameter settings obtained in the previous step are mutually independent and thus can be parallelized. However, FRAMA-C/EVA per se does not support the execution of parallel tasks. Hence, we implement this functionality using the OCaml module `Parmap`[4]. The degree of parallelization, i.e., the number of processes, is determined by the user-defined hyper-parameter num_{process}.

Some analyses may fail to terminate within the given time limit, which is constrained by the total time budget (controlled by a hyper-parameter $time_{\text{budget}}$) for all the sample-analyze-refine rounds. For each analysis, PARF records whether it terminates and, if yes, the so-reported alarms. These intermediate results are then utilized to refine the distribution of P.

3.4 Distribution Refining

PARF refines the distribution of P_{base} based on its latticed sample spaces $(S, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, leveraging all the collected intermediate results. Table 4 shows an example of such refinement for `slevel`, which is a crucial parameter for controlling the capac-

Table 3. Distributions of P, P_{base}, and P_{delta}

Type	S	Distribution of P_{base}	Distribution of P_{delta}	$P = P_{\text{base}} \oplus P_{\text{delta}}$
Integer	$\mathbb{N} \cup \{\infty\}$	$\Pr[P_{\text{base}} = a] = 1$	$\text{Poisson}(\lambda)$	$a + P_{\text{delta}}$
Boolean	$\{\text{false}, \text{true}\}$	$\Pr[P_{\text{base}} = b] = 1$	$\text{Bernoulli}(q)$	$b \vee P_{\text{delta}}$
Set-of-strings	$\{\text{false}, \text{true}\}^c$	$\Pr[P_{\text{base}} = (b_1, \ldots, b_c)] = 1$	$\text{Bernoulli}(q_1) \times \ldots \times \text{Bernoulli}(q_c)$	$(b_1 \vee P^1_{\text{delta}}) \times \ldots \times (b_c \vee P^c_{\text{delta}})$

Note: P_{base} follows a Dirac distribution where a is an integer sample and $b, b_1, \ldots, b_c$ are Boolean values. P_{delta} adopts one of the following distributions, depending on the parameter type: Poisson distribution, Bernoulli distribution, or c-dimensional independent joint Bernoulli distribution (in this case, P_{delta} can be expressed as $(P^1_{\text{delta}}, \ldots, P^c_{\text{delta}})$). The binary operator $\oplus$ also varies based on the parameter type: it corresponds to addition $(+)$ and logical disjunction $(\vee)$ for an integer or Boolean parameter, respectively; for a set-of-strings parameter with cardinality c, $\oplus$ is defined as the point-wise lifting of $\vee$ to a c-dimensional random vector.

[3] Table A1 of Appendix A1, https://jcst.ict.ac.cn/article/doi/10.1007/s11390-025-5140-6#Supplements-list, Jun. 2025.
[4] https://opam.ocaml.org/packages/parmap/, Jun. 2025.

Table 4. Example of Refining P_{base} for `slevel`

Analysis	Value	Alarm 1	Alarm 2	Alarm 3	Alarm 4
1	58	✘	✘	✔	✔
2	103	✘	✘	✔	✔
3	104	✘	✘	✘	✔
4	1 000	✘	✘	✘	✔
5	9	✘	✔	✔	✔
6	9 999	–	–	–	–

Note: The second column lists the values of parameter `slevel`. ✔ and ✘ in the $(j+2)$-th column indicate whether the analysis produces Alarm j (✔) or not (✘), and – marks a failed analysis.

ity of separate (unmerged) states during the static analysis. The individual analyses as exemplified in Table 4 are produced in parallel within a single iteration. Our artifact then constructs a matrix $\boldsymbol{R} \in \{✔, ✘\}^{m \times n}$, to represent the intermediate results (excluding failed analyses), where m is the number of successfully completed analyses and n is the cardinality of the universal set of reported alarms. We also use an m-dimensional vector $\boldsymbol{V}$ to denote the parameter values used in each analysis (V_i signifies the parameter value for the i-th analysis). Next, Parf performs Algorithm 1 to refine the distribution of P_{base}.

Algorithm 1. Refining the Distribution of P_{base}

Input: $\boldsymbol{R}$: $m \times n$ intermediate-result matrix; $\boldsymbol{V}$: m-dimensional parameter value vector; P_{base}: original distribution.

Output: P'_{base}: refined distribution.

$P'_{\text{base}} \leftarrow P_{\text{base}}$;

for $j \leftarrow 1$ **to** n **do** ▷ `iterate over columns of` $\boldsymbol{R}$ `(alarms)`

  $tmp \leftarrow \top$;

  for $i \leftarrow 1$ **to** m **do** ▷ `scan rows of` $\boldsymbol{R}$ `(analyses)`

    if $R_{ij} =$ ✗ **then** $tmp \leftarrow tmp \sqcap V_i$

  if $tmp \neq \top$ **then** $P'_{\text{base}} \leftarrow P'_{\text{base}} \sqcup tmp$;

return P'_{base};

Algorithm 1 employs two nested loops. For the j-th column of $\boldsymbol{R}$ (w.r.t. Alarm j), the inner loop computes the greatest lower bound (for the lowest precision) of all sampled parameters which can eliminate (false) Alarm j. The outer loop casts the least upper bound for eliminating all such false alarms with the lowest precision. Considering the example in Table 4, P_{base} is refined as:

$$\begin{aligned} P'_{\text{base}} = P_{\text{base}} &\sqcup (\top \sqcap 58 \sqcap 103 \sqcap 104 \sqcap 1\,000 \sqcap 9) \\ &\sqcup (\top \sqcap 58 \sqcap 103 \sqcap 104 \sqcap 1\,000) \\ &\sqcup (\top \sqcap 104 \sqcap 1\,000) \\ = P_{\text{base}} &\sqcup 9 \sqcup 58 \sqcup 104. \end{aligned}$$

It follows that P'_{base} is the least precise parameter setting (w.r.t. `slevel`) that can eliminate all newly discovered false alarms in the current iteration.

For refining the distribution of P_{delta}, Parf uses the so-called completion rate η_c, i.e., the ratio of successfully completed analyses to all the num_{sample} analyses. P_{delta} is then refined via the scaling factor $\eta = 2\eta_c + (1/num_{\text{sample}})$ as per Table 5 ($\eta > 1$ for $\eta_c \geqslant 0.5$). A larger value of η indicates that more analyses have been completed within the allocated time budget, suggesting that a more extensive exploration of the parameter space (by scaling up P_{delta}) is possible, and vice versa.

Table 5. Refining the Distribution of P_{delta}

Type	Original P_{delta}	Refined P_{delta}
Integer	Poisson(λ)	Poisson($\lambda \times \eta$)
Boolean	Bernoulli(q)	Bernoulli($1-(1-q)^{\eta}$)
Set-of-strings	B(q_1) $\times \ldots \times$ B(q_c)	B($1-(1-q_1)^{\eta}$) $\times \ldots \times$ B($1-(1-q_c)^{\eta}$)

Note: B(q) is shorthand for Bernoulli(q).

4 Empirical Evaluation

In this section, we evaluate the Parf artifact⑤ to answer the following research questions.

*RQ*1 (*Consistency*). Can the artifact reproduce experimental results as reported in [9] (given the inherent randomness of Parf due to the sampling module)?

*RQ*2 (*Verification Capability*). Can Parf improve Frama-C in verification competitions?

*RQ*3 (*Dominancy*). Which are the dominant (i.e., most influential) parameters in Frama-C/Eva?

*RQ*4 (*Interpretability*). How does Parf help eliminate false alarms or even certify the absence of RTEs?

4.1 Experimental Setup

Benchmarks. We evaluate Parf over two benchmark suites.

1) The first suite is Frama-C Open Source Case Study (OSCS) Benchmarks⑥ (as per [9]), comprising 37 real-world C code bases, such as the "X509" parser project (a Frama-C-verified parser)[11] and "chrony" (a versatile implementation of the Network Time Protocol). The benchmark details are provided in Table 6.

2) The second suite is collected from the verifica-

⑤https://hub.docker.com/repository/docker/parfdocker/parf-jcst/general, Jun. 2025.
⑥https://git.frama-c.com/pub/open-source-case-studies, Jun. 2025.

Table 6. Experimental Results in Terms of RQ1 (Consistency)

OSCS Benchmark Details				#Alarms of Baselines				#Alarms of Parf	
Benchmark Name	LOC	#Statements	`-eva-precision`	`-eva-precision` 0	Default	Expert	Official	Parf_Opt	Parf_Avg
2048	440	329	6	13	7	5	7	**4**	4.33
chrony	37 177	41	11	9	9	7	8	7	7.00
debie1	8 972	3 243	2	33	33	3	**1**	2	3.33
genann	1 183	1 042	10	236	236	69	77	69	69.00
gzip124	8 166	4 835	0	885	884	885	866	**807**	836.00
hiredis	7 459	87	11	9	9	0	9	0	0.00
icpc	1 302	424	11	9	9	1	1	1	1.00
jsmn-ex1	1 016	1 219	11	58	58	1	1	1	1.00
jsmn-ex2	1 016	311	11	68	68	1	1	1	1.00
kgflags-ex1	1 455	474	11	11	11	0	11	0	0.00
kgflags-ex2	1 455	736	10	33	33	19	33	19	19.00
khash	1 016	206	11	14	14	2	14	2	2.00
kilo	1 276	1 078	2	523	523	445	688	**419**	421.67
libspng	4 455	2 377	7	186	186	122	122	126	145.33
line-following-robot	6 739	857	10	1	1	1	1	1	1.00
microstrain	51 007	3 216	6	1 177	1 177	616	646	**601**	606.00
mini-gmp	11 706	628	6	83	83	71	83	**65**	68.67
miniz-ex1	10 844	3 659	1	2 291	2 291	1832	2 291	**1**	763.67
miniz-ex2	10 844	5 589	1	2 748	2 742	2 220	2 742	2 219	2 475.33
miniz-ex3	10 844	3 747	1	585	577	552	577	**432**	510.67
miniz-ex4	10 844	1 246	4	264	258	217	258	**188**	206.67
miniz-ex5	10 844	3 430	2	431	425	**371**	425	385	389.00
miniz-ex6	10 844	2 073	2	220	220	190	220	**175**	183.33
monocypher	25 263	4 126	2	606	606	564	568	572	577.67
papabench	12 254	36	11	1	1	1	1	1	1.00
qlz-ex1	1 168	229	11	68	68	11	68	11	21.33
qlz-ex2	1 168	75	11	8	8	8	8	8	8.00
qlz-ex3	1 168	294	8	94	94	82	94	82	82.00
qlz-ex4	1 168	164	11	17	17	13	17	13	13.00
safestringlib	29 271	13 029	7	855	855	**256**	300	263	268.33
semver	1 532	728	9	29	29	22	25	22	23.00
solitaire	338	396	11	216	216	18	213	18	18.00
stmr	781	500	6	63	63	58	59	58	58.00
tsvc	5 610	5 478	4	413	413	355	379	354	356.00
tutorials	325	89	11	5	5	1	5	**0**	0.00
tweetnacl-usable	1 204	659	11	126	126	25	30	25	25.00
x509-parser	9 457	3 112	3	208	208	198	198	**181**	185.33
Overall (tied-best+**exclusively best**)				3/37	3/37	24/37	9/37	33/37 (89.2%)	–
Overall (**exclusively best**)				0/37	0/37	2/37	1/37	11/37 (29.7%)	–

Note: The benchmark details include: 1) name of each benchmark; 2) LOC (lines of code): size of each benchmark's source files; 3) #Statements: the number of statements covered during analysis for each benchmark; 4) `-eva-precision`: the highest precision level identified by Expert under the experimental configuration. #Alarms is the number of alarms generated by different parameter-tuning mechanisms (baselines and Parf).

tion tasks of SV-COMP 2024[12], where Frama-C participated in the NoOverflows category with a specific version called Frama-C-SV[13].

Baselines. We compare Parf against four parameter-tuning mechanisms: `-eva-precision` 0, Default, Expert, and Official. The former two adopt the lowest-precision and default abstraction strategies of Frama-C/Eva, respectively. EXPERT dynamically adjusts the precision of abstraction strategies by sequentially increasing the `-eva-precision` meta-option from 0 to 11 until the given time budget is exhausted or the highest precision level is reached. The OFFICIAL mechanism uses the tailored strategies provided by Frama-C/Eva for the OSCS benchmarks, which can be regarded as "high-quality" configurations.

Configurations. All experiments are performed on a system equipped with two AMD EPYC 7542 32-core processors and 128 GB RAM running Ubuntu

22.04.5 LTS. To attain consistency, we adopt the same hyper-parameters as in [9] (with $num_{\text{sample}} = 4$ and $time_{\text{budget}} = 1$ hour for each benchmark).

4.2 RQ1: Consistency

Table 6 reports the analysis results in terms of the number of emitted alarms. For PARF, due to its inherent randomness, we repeat each experiment three times and report both the best result (PARF_OPT) and the averaged result (PARF_AVG). Since the former is also adopted in [9], we primarily compare PARF_OPT against the four baselines. We mark results with the exclusively fewest alarms (with difference $> 1\%$) as **exclusively best** and results with the same least number of alarms (modulo a difference of $\leqslant 1\%$) as tied-best.

Overall, PARF achieves the least number of alarms on 33/37 (89.2%) benchmarks with exclusively best results on 11/37 (29.7%) cases, significantly outperforming its four competitors. These results are consistent with those obtained in [9] (best: 34/37; exclusively best: 12/37). The minor differences stem primarily from the inherent randomness of PARF and changes in hardware configurations. We observe a special case for "miniz-ex1", where #alarms reduces from 1 828 as in [9] to 1. This correlates with the fact that PARF finds an abstraction strategy that triggers a drastic decrease in FRAMA-C's analysis coverage[1]. Moreover, as is observed in [9], PARF is particularly suitable for analyzing complex, large-scale real-world programs (i.e., benchmarks featuring low levels of `-eva-precision`).

4.3 RQ2: Verification Capability

Static analyzers, such as FRAMA-C, can be applied in verification scenarios[12]. Table 7 shows that PARF can improve the performance of FRAMA-C in SV-COMP. The detailed scoring schema is presented in online appendix⑦, as per [12]. Since the analysis resource for each verification task is limited to 15 minutes of CPU time, the FRAMA-C-SV$_{\text{precision11}}$ strategy uses a fixed highest `-eva-precision11` parameter for analysis (as per [13]). We set the hyper-parameters $time_{\text{budget}}$, num_{process}, and num_{sample} of FRAMA-C-SV$_{\text{PARF}}$ to 7.5 minutes, 2, and 4, respectively.

The experimental results demonstrates PARF's methodological robustness in enhancing verification capacity of FRAMA-C. Specifically, PARF eliminates all 104 analysis failures (due to the timeout) and verifies 39 more tasks, thereby improving the total score from 1 006 to 1 084. Fig.2 illustrates that PARF adaptively identifies 42 high-accuracy analysis results among the 104 failure cases, thus successfully verifying them. Furthermore, among all the 1 057 true correct cases verified by FRAMA-C-SV$_{\text{precision11}}$, PARF misses only 3.

4.4 RQ3: Dominancy

We show via a post-hoc analysis that PARF supports the identification of the most influential parameters dominating the performance of FRAMA-C/EVA. To this end, we conduct 13 pairs (each for a single parameter) of controlled experiments for each OSCS benchmark⑧. For instance, Fig.3 depicts the results of 13 pairs of analyses for the "2048" benchmark. The analyses using the PARF_OPT configuration (reporting four alarms) and `-eva-precision 0` configuration (reporting 13 alarms) signify a high-precision upper bound and a low-precision lower bound, respectively. For each parameter, we devise two types of controlled experiments: 1) SELECTED. The parameter is selected and retained from the PARF_OPT configuration, while the other 12 parameters are taken from the `-eva-precision 0` configuration. This controlled experiment assesses the impact of the parameter w.r.t. the lower-bound baseline. 2) EXCLUDED. The parameter is excluded from the PARF_OPT configuration and replaced with its counterpart from `-eva-precision 0`, while the remaining 12 parameters are retained from the PARF_OPT configuration. This controlled experiment evaluates the parameter's influ-

Table 7. SV-COMP Verification Results in Terms of RQ2 (Verification Capability)

Setting	Verification Result							Score
	Correct		Incorrect		Invalid			
	True (+2)	False (+1)	True (−32)	False (−16)	Unknown (0)	Failure (0)	Error (0)	
FRAMA-C-SV$_{\text{precision11}}$	1 057	12	35	0	564	104	48	1 006
FRAMA-C-SV$_{\text{PARF}}$	**1 096**	12	35	0	629	**0**	48	**1 084**

⑦Appendix A2, https://jcst.ict.ac.cn/article/doi/10.1007/s11390-025-5140-6#Supplements-list, Jun. 2025.

⑧Trivial benchmarks where all parameter-tuning mechanisms yield identical performance (e.g., "papabench") are excluded.

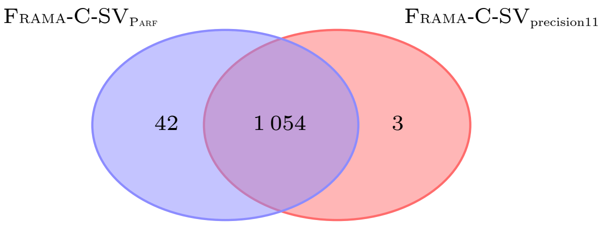


Fig.2. Venn-diagram depicting the sets of true correct verification tasks by Frama-C-SV$_{\text{Parf}}$ and Frama-C-SV$_{\text{precision11}}$, respectively.

ence w.r.t. the upper-bound baseline.

We then devise for each parameter a scoring function s to quantitatively characterize its influence:

$$s \triangleq \frac{0.5 \times a + 0.5 \times b}{d},$$

where a, b, and d capture the difference in #alarms respectively between three cases: 1) the `-eva-precision` 0 baseline and the Selected experiment, 2) the Excluded experiment and the Parf_Opt baseline, and 3) the `-eva-precision` 0 baseline and the Parf_Opt baseline. For instance, for Para.13 in Fig.3, we have $a = 13 - 8 = 5$, $b = 5 - 4 = 1$, and $d = 13 - 4 = 9$.

The influence score for all parameters across the OSCS benchmark are collected in online appendix[9]. For each benchmark, we mark the parameter with the highest score as the dominant parameter. It follows that, overall, `slevel` (Para.5) is the most influential parameter in Frama-C/Eva (with an averaged score of 0.490) and `domains` (Para.13) is the second most influential parameter (with an averaged score of 0.258). This observation conforms to the crucial roles of `slevel` and `domains` in static analysis: the former restricts the number of abstract states at each control point, and the latter determines the types of abstract representations.

Nonetheless, the dominant parameter can vary for different target programs, e.g., the dominant parameters for "debie1" and "miniz-ex6" are `auto-loop-unroll` (Para.2) and `min-loop-unroll` (Para.1), respectively. This suggests that many false alarms emitted for these benchmarks can be eliminated through loop unrolling. Notably, "miniz-ex6" contains multiple nested loops that require extensive iterations to be fully unwound.

An unexpected observation is that certain parameters exhibit negative influence scores on a few benchmarks, such as Para.8 on "jsmn-ex2" and most parameters on "qlz-ex3". These negative scores arise when the Selected experiments produce more alarms than the `-eva-precision` 0 baseline, or when the Excluded experiments result in fewer alarms than the Parf_Opt baseline[10]. This phenomenon suggests that

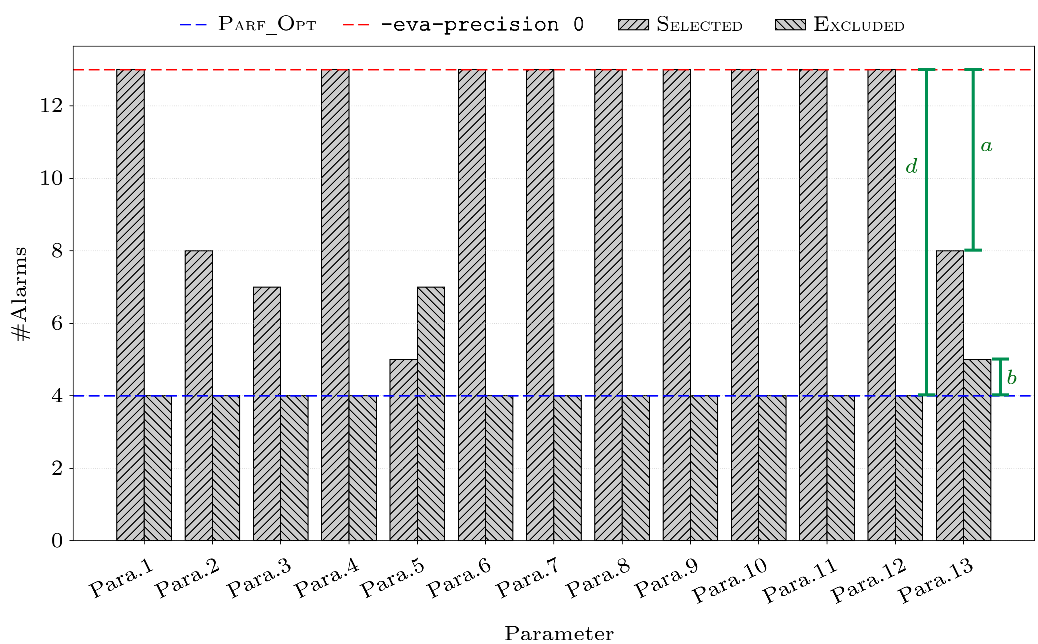


Fig.3. Number of alarms (#Alarms) of analyses on benchmark "2048" upon tuning individual parameters based on the abstraction strategies produced by `-eva-precision` 0 and Parf_OPT. Para.n refers to the n-th parameter listed in Table 1.

[9]Table A2 of Appendix A3, https://jcst.ict.ac.cn/article/doi/10.1007/s11390-025-5140-6#Supplements-list, Jun. 2025.
[10]Fig.A1 of Appendix A3, https://jcst.ict.ac.cn/article/doi/10.1007/s11390-025-5140-6#Supplements-list, Jun. 2025.

Frama-C/Eva is not strictly monotonic in certain cases. Nevertheless, our refinement mechanism equips Parf with the potential to handle these corner cases effectively. One such case is examined by the Frama-C community[11] as well as discussed in Subsection 4.5.

4.5 RQ4: Interpretability

We show how Parf helps eliminate false alarms or even certify the absence of RTEs through a case study. Fig.4 gives a simplified version of the "tutorials" benchmark — a toy program used to calculate differences between the ID of each parent process and its children.

```c
1  #define MAX_CHILD_LEN 10
2  #define MAX_BUF_SIZE 100
3  #define MAX_PROCESS_NUM 50
4
5  struct process {
6    uint8_t pid;
7    uint8_t child_len;
8    uint8_t child[MAX_CHILD_LEN];
9  };
10 struct process p[MAX_PROCESS_NUM];
11
12 // init returns -1 if initializing p[p_id] fails and 0 otherwise
13 int init(uint8_t *buf, uint16_t *offset, uint8_t p_id){
14   ... // the concrete implementation body is abstracted away
15 };
16
17 int main(){
18   uint8_t buf[MAX_BUF_SIZE], p_nb;
19   uint16_t offset = 0;
20   random_init((char*)buf, MAX_BUF_SIZE);
21
22   // initialize the global array p of process structures
23   for(p_nb = 0; p_nb < MAX_PROCESS_NUM; p_nb++){
24     int r = init(buf, &offset, p_nb);
25     if(r) break;
26   }
27
28   // print pid diff. btw. each valid process and its children
29   for(uint8_t p_id = 0; p_id < p_nb; p_id++){
30     for(uint8_t i = 0, c_id; i < p[p_id].child_len; i++)
31       c_id = p[p_id].child[i];
32       //@ assert c_id < MAX_PROCESS_NUM;
33       printf("␣%i",p[p_id].pid - p[c_id].pid);
34   }
35   return 0;
36 };
```

Fig.4. Simplified version of the benchmark "tutorials".

Table 6 shows that Parf suffices to eliminate all alarms, yet Expert reports one false alarm. This alarm corresponds to the assertion `c_id` < `MAX_PROCESS_NUM` in line 32, signifying a potential out-of-bound RTE. A typical way to eliminate this false alarm is by maintaining a sufficiently large number of abstract states at this control point (loop condition in line 30) by setting a high `slevel` to prevent an over-approximation of the value of `c_id`. This trick, unfortunately, does not work for this specific program (Expert sets `slevel` to 5 000, as is similar to Parf). The reason why Parf can eliminate the false alarm lies in its configuration of `partition-history`: Expert sets it to 2, yet Parf sets it to 0. When `partition`-history is set to $n \geqslant 1$, it delays the application of join operation on abstract domains, leading to an exponential increase (in n) of the number of abstract states required to avoid over-approximations at control points. Consequently, Expert using both high-precision `slevel` and `partition-history` fails to eliminate the false alarm in question, whilst Parf succeeds by pairing a high-precision `slevel` with a low-precision `partition-history`.

The effectiveness of Parf roots in its ability to maintain low-precision distributions for disturbing parameters (those with negative contributions to eliminating false alarms for specific programs, e.g., `partition-history` for "tutorials") while achieving high-precision distributions for dominant parameters (e.g., `slevel` for "tutorials") during the refinement procedure. This ingenuity can be attributed to two key factors. 1) Unlike Expert, which groups and binds all parameters into several fixed configuration packs, Parf models each parameter as an independent random variable. 2) The "meet-and-join" refinement strategy (described in Algorithm 1) restrains the growth of P_{base} for disturbing parameters while increasing P_{base} for dominant parameters. In a nutshell, despite its assumption on monotonic analyzers (cf. Section 2), Parf exhibits strong potential to improve the performance of static analyzers that lack strict monotonicity.

5 Limitations and Future Work

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

First, Parf models different parameters of a static analyzer as independent random variables. However, the interactions between parameters can potentially lead to complex parameter dependencies. For instance, 1) larger `partition-history` requires (exponentially) larger `slevel` to delay approximations for all conditional structures[1], and 2) the modification of `domains` can unpredictably interact with `slevel`[1]. 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 repre-

[11] https://stackoverflow.com/q/79 497 136/15 322 410, Jun. 2025.

sentation of stochastic dependencies, such as conditional random variable models.

Second, parameter initialization in Parf relies on fixed heuristically-defined distributions, without leveraging historical experience (e.g., expert knowledge on configuring typical programs) or program-specific features (e.g., the syntactic or semantic characteristics of the source program) to optimize initial configurations. While neural networks or fine-tuned large language models could automate this process, their deployment requires balanced training data pairing programs with optimal parameters — a dataset traditionally requiring expert curation. Notably, Parf's automated configuration generation capability paves the way for constructing such datasets at scale, enabling data-driven initialization as promising future work.

Third, while Parf enhances static analyzers' capacity, it cannot fully eliminate false positives due to the fundamental precision-soundness trade-offs of static analysis. As shown in Table 6, residual alarms require manual inspection. A promising direction involves integrating Parf with formal verification tools (e.g., proof assistants or SMT solvers) to classify alarm validity.

6 Related Work

Abstraction Strategy Refinement. Beyer *et al.*[14] proposed CPA+, a framework that augments the program verifier CPA[15] with deterministic abstraction-strategy tuning schemes based on intermediate analysis information (e.g., predicates and abstract states). CPA+ aims to enhance the scalability and efficiency of verification, such as predicate abstraction-based model checking, while Parf focuses on improving the accuracy of static analysis leveraging the alarm information. Zhang *et al.*[10] introduced BinGraph, a framework for learning abstraction selection in Bayesian program analysis. Yan *et al.*[16] proposed a framework that utilizes graph neural networks to refine abstraction strategies for Datalog-based program analysis. These data-driven methods[10, 16] require datasets for training a Bayesian/neural network, while Parf requires no pre-training effort. The theoretical underpinnings of Parf were established in [9]. In this paper, we focus on the design, implementation, and application of Parf and make new contributions detailed in Section 1.

Improving Static Analyzers. Modern static analyzers employ diverse parameterization strategies to balance precision and performance. Kästner *et al.*[8] summarized the four most important abstraction mechanisms in Astrée and recommended prioritizing the accuracy of related abstract domains, which amounts to narrowing down the parameter space. However, these mechanisms need hand-written directives and thus are not fully automated. Mopsa[4] adopts a fixed sequence of increasingly precise configurations akin to Frama-C/Eva's EXPERT mechanism when participating SV-COMP 2024. Parf can be generalized to Mopsa by modeling its specific parameters, thus helping to decide the best configuration to analyze a given program. Saan *et al.*[3] implemented in Goblint a simple, heuristic autotuning method based on syntactical criteria, which can 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.

7 Conclusions

We presented the Parf toolkit for adaptively tuning abstraction strategies of static program analyzers. It is — to the best of our knowledge — the first fully automated approach that supports incremental refinement of such strategies. The effectiveness of Parf is demonstrated through a case study where it certified the absence of RTEs in a special case, alongside benchmark evaluations showing that it enhanced Frama-C's performance in SV-COMP 2024 by verifying 39 additional tasks. Interesting future directions include extending Parf to cope with dependencies between parameters, neural network-based parameter initialization, and combining formal verification tools.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Bühler D, Cuoq P, Yakobowski B. The Eva plug-in for Frama-C 27.1 (Cobalt). 2023. https://www.frama-c.com/download/frama-c-eva-manual.pdf, Jul. 2025.

[2] Kästner D, Wilhelm R, Ferdinand C. Abstract interpretation in industry-experience and lessons learned. In *Lecture Notes in Computer Science 14284*, Hermenegildo M V, Morales J F (eds.), Springer, 2023, pp.10–27. DOI: 10.1007/978-3-031-44245-2_2.

[3] Saan S, Schwarz M, Erhard J, Pietsch M, Seidl H, Tilsch-

er S, Vojdani V. GOBLINT: Autotuning thread-modular abstract interpretation. In *Lecture Notes in Computer Science 13994*, Sankaranarayanan S, Sharygina N (eds.), Springer, 2023, pp.547–552. DOI: 10.1007/978-3-031-30820-8_34.

[4] Monat R, Milanese M, Parolini F, Boillot J, Ouadjaout A, Miné A. Mopsa-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In *Lecture Notes in Computer Science 14572*, Finkbeiner B, Kovács L (eds.), Springer, 2024, pp.387–392. DOI: 10.1007/978-3-031-57256-2_26.

[5] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1977, pp.238–252. DOI: 10.1145/512950.512973.

[6] Venet A J. The gauge domain: Scalable analysis of linear inequality invariants. In *Proc. the 24th International Conference on Computer Aided Verification*, Jul. 2012, pp.139–154. DOI: 10.1007/978-3-642-31424-7_15.

[7] Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. A static analyzer for large safety-critical software. In *Proc. the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2003, pp.196–207.

[8] Kaestner D, Wilhelm S, Mallon C, Schank S, Ferdinand C, Mauborgne L. Automatic sound static analysis for integration verification of AUTOSAR software. SAE Technical Paper 2023-01-0591. SAE, 2023, pp.65–68. https://saemobilus.sae.org/papers/automatic-sound-static-analysis-integration-verification-autosar-software-2023-01-0591, Jun. 2025.

[9] Wang Z, Yang L, Chen M, Bu Y, Li Z, Wang Q, Qin S, Yi X, Yin J. Parf: Adaptive parameter refining for abstract interpretation. In *Proc. the 39th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2024, pp.1082–1093.

[10] Zhang Y, Shi Y, Zhang X. Learning abstraction selection for Bayesian program analysis. *Proceedings of the ACM on Programming Languages*, 2024, 8(OOPSLA1): 954–982. DOI: 10.1145/3649845.

[11] Ebalard A, Mouy P, Benadjila R. Journey to a RTE-free X. 509 parser. *SSTIC* 2019. https://www.sstic.org/media/SSTIC2019/SSTIC-actes/journey-to-a-rte-free-x509-parser/SSTIC2019-Article-journey-to-a-rte-free-x509-parser-ebalard_mouy_benadjila_3cUxSCv.pdf, Jul. 2025.

[12] Beyer D. State of the art in software verification and witness validation: SV-COMP 2024. In *Proc. the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2024, pp.299–329. DOI: 10.1007/978-3-031-57256-2_15.

[13] Beyer D, Spiessl M. The static analyzer Frama-C in SV-COMP (competition contribution). In *Proc. the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2022, pp.429–434. DOI: 10.1007/978-3-030-99527-0_26.

[14] Beyer D, Henzinger T A, Théoduloz G. Program analysis with dynamic precision adjustment. In *Proc. the 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2008, pp.29–38. DOI: 10.1109/ASE.2008.13.

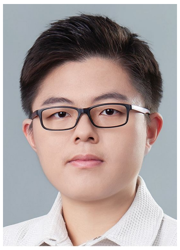
[15] Beyer D, Henzinger T A, Théoduloz G. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. the 19th International Conference on Computer Aided Verification*, Jul. 2007, pp.504–518. DOI: 10.1007/978-3-540-73368-3_51.

[16] Yan Z, Zhang X, Di P. Scaling abstraction refinement for program analyses in datalog using graph neural networks. *Proceedings of the ACM on Programming Languages*, 2024, 8(OOPSLA2): 1532–1560. DOI: 10.1145/3689765.

Zhong-Yi Wang received his B.S. degree in bioinformatics from Shanghai Jiao Tong University, Shanghai, in 2023. He is currently pursuing his Ph.D. degree in software engineering at Zhejiang University, Hangzhou. His research interests include static program analysis and automated verification.

Ming-Shuai Chen received his Ph.D. degree in computer science from Institute of Software, Chinese Academy of Sciences, Beijing, in 2019. He then worked as a postdoctoral researcher at RWTH Aachen University, Aachen. Since 2023, he joined the College of Computer Science and Technology at Zhejiang University, Hangzhou, as an assistant professor (ZJU100 Young Professor). His research interests include formal verification, programming theory, and logical aspects of computer science.

Teng-Jie Lin received his B.E. degree in computer science and technology from Xidian University, Xi'an, in 2025. He is currently a master student in software engineering at Zhejiang University, Hangzhou. His research interests include static analysis, automated verification, and LLM+SE+FM.



Lin-Yu Yang received his B.E. degree in information engineering from Nanjing University of Information Science and Technology (NUIST), Nanjing, in 2024. He is currently a Ph.D. candidate at Zhejiang University (ZJU), Hangzhou. He is a member of FICTION, the Formal Verification Group in the College of Computer Science and Technology at ZJU. His research interests include programming language theory, static analysis, and theorem proving.

Jun-Hao Zhuo is currently an undergraduate student majoring in computer science and technology at Zhejiang University, Hangzhou. His research interests include deep learning and the deployment of large-scale pretrained models in production-level systems and real-life scenarios.

Qiu-Ye Wang received his Ph.D. degree from the Institute of Software, Chinese Academy of Sciences (ISCAS), Beijing, in 2022, working on formal method research. He joined Huawei after that to seek the application of formal methods in the industrial world. His major focus is on formal-method-related software engineering.

Sheng-Chao Qin received his B.Sc. degree and Ph.D. degree from Peking University, Beijing. He is currently a Chair Professor at Xidian University, Guangzhou, and his research interests include formal methods and their applications, software engineering, programming languages, as well as AI+FM.

Xiao Yi received his B.E. degree in software engineering from Xi'an Jiaotong University, Xi'an, his M.S. degree in computer science from Boston University, Boston, and his Ph.D. degree in information engineering from the Chinese University of Hong Kong, Hong Kong. His research interests during Ph.D. study focus on vulnerability analysis and detection in blockchain systems. He is currently a researcher in program verification at Fermat Labs, Huawei Hong Kong Research Center.

Jian-Wei Yin received his Ph.D. degree in computer science from Zhejiang University, Hangzhou, in 2001. He was a visiting scholar with the Georgia Institute of Technology, Atlanta. He is currently a full professor at the College of Computer Science and Technology at Zhejiang University, Hangzhou. His research interests include software theory and engineering for emerging computing paradigms. He is an associate editor of the IEEE Transactions on Services Computing.