SATUNE: A Study-Driven Auto-Tuning Approach for Configurable Software Verification Tools

Anonymous Authors

***Abstract*—Many program verification tools can be customized via run-time configuration options that trade off performance, precision, and soundness. However, in practice, users often run tools under their default configurations, because understanding these tradeoffs requires significant expertise. In this paper, we ask how well a single, default configuration can work in general, and we propose SATUNE, a novel tool for automatically configuring program verification tools for given target programs. To answer our question, we gathered a dataset that runs four well-known program verification tools against a range of C and Java benchmarks, with results labeled as correct, incorrect, or inconclusive (e.g., timeout). Examining the dataset, we find there is generally no one-size-fits-all best configuration. Moreover, a statistical analysis shows that many individual configuration options do not have simple tradeoffs: they can be better or worse depending on the program.**

Motivated by these results, we developed SATUNE, which constructs configurations using a meta-heuristic search. The search is guided by a surrogate fitness function trained on our dataset. We compare the performance of SATUNE to three baselines: a single configuration with the most correct results in our dataset; the most precise configuration followed by the most correct configuration (if needed); and the most precise configuration followed by random search (also if needed). We find that SATUNE outperforms these approaches by completing more correct tasks with high precision. In summary, our work shows that good configurations for verification tools are not simple to find, and SATUNE takes an important first step towards automating the process of finding them.

I. INTRODUCTION

Static program verification tools are a promising approach for reasoning about the correctness of software. Because the algorithms that back such tools present various precision, soundness, and performance[1] tradeoffs [1]–[3], such tools often include a host of options for tuning the analysis. However, deciding how to set these options can be quite challenging as it may require deep knowledge of the analysis algorithms [2], [4]. Thus in practice, many users rely on a default configuration recommended by developers for typical scenarios. Unfortunately, prior work suggests that the tradeoffs among options depends on the features of the program being analyzed [5]–[9].

In response to these challenges, several researchers have explored ways to choose a program verifier or configuration of a verifier to best fit a target program [6], [10]–[13] and have selected a single setting of an analysis configuration option [5], [8], [9], [14]. Other work aims to develop an understanding of certain kinds of configuration options in static analysis frameworks or tools [2], [3], [15]. To our knowledge, the prior work has focused on relatively small and specific configuration spaces. As a result, the effects of configurations on program verification tools is still poorly understood, and it remains difficult to tune such tools.

In this paper, we aim to address this gap in two steps. First, to better understand the configurability of program verification tools, we perform an empirical study in which we construct and analyze a dataset of runs of four popular tools on a range of benchmarks. Second, driven by the results of the empirical study, we propose SATUNE, a novel technique for automatically tuning the the large configuration spaces of software verification tools.

Our empirical study examines four tools that participate in the annual software verification competition (SV-COMP) [16]. *CBMC* [17] and *Symbiotic* [18], [19] verify C/C++ programs, and *JBMC* [20] and *JayHorn* [21] verify Java programs. We created a dataset by running sampled configurations of the four tools on a subset of SV-COMP benchmarks. Each of the 517748 tool–configuration–benchmark triples in our dataset is labeled as either producing a correct, incorrect, or inconclusive (e.g., timeout) result. We then analyzed the data to answer two research questions. First, we ask whether, for each tool, there exists a *one-size-fits-all* configuration that produces a superset of complete, correct results (**RQ1**). We found that even the *most-correct-config*—the configuration that produces the most true positive and true negative results for a tool—is unable to complete many verification tasks that other configurations could. Second, we use statistical analysis to investigate the impact of individual configuration options on the tools' performance and precision (**RQ2**). We found that for each tool, at most half of the option settings have a statistically significant effect on the number of correct or incorrect results produced by a tool. Thus, many option settings do not change the results much. We also found that the option settings that do have significant effects increase the number of correct results in some programs and decrease it in others, suggesting their effects can vary greatly from program to program. (See Section II for details of our empirical study.)

Overall, our study suggests that no single configuration is sufficient for general use, and configurations may need to be tuned to the target program. Thus, we propose SATUNE, a novel technique that aims to find a good tool configuration for a given target program. Because the tools' configuration spaces are very large and complex, SATUNE finds configurations using a meta-heuristic search driven by a fitness function,

[1]In this work, when we speak of "performance," we are referring to the quality of producing correct results; in other words, maximizing the number of true positive and true negative results produced.

where the fitness of a configuration is its predicted likelihood to terminate with a correct result on the target program. We implement the fitness function as a machine learning model trained on the dataset from the empirical study. One key feature of SATUNE is that it is both tool- and language-agnostic, as the approach only requires a labeled training dataset, the ability to run a tool from the command line, and the ability to process its output. (Section III describes SATUNE.)

We evaluate SATUNE by comparing it against three baselines that simulate ways a user might tune a verification tool: first, using the configuration *most-correct-config* that produces the most correct results from our dataset; second, using the configuration *best-precision-config* that is the most precise (i.e., maximizes $\frac{\#\text{corrects}}{\#\text{corrects}+\#\text{incorrects}}$) and, if it does not complete, trying *most-correct-config*; and third, using *best-precision-config* and, if it does not complete, using random search. We found that, compared to these baselines, SATUNE provides the best balance between precision and number of complete results (**RQ3**). For example, in the best case, SATUNE was able to analyze 169 (27%) more programs with higher precision than any baseline. We also evaluated SATUNE in terms of run time, and found that it was 2–4x faster than random search and was comparable to the second baseline (**RQ4**). (Section IV presents our evaluation of SATUNE.)

In summary, our results suggest there is no one-size-fits-all best configuration for the studied program verification tools, and that effects of individual configuration options can vary greatly from program to program. Thus, we believe that SATUNE takes an important first step toward automating the process of finding good program verification tool configurations. We plan to make our dataset, results, and implementation publicly available upon paper acceptance.

II. Empirical Study of Configurable Verification Tools

To better understand the configurability of program verification tools, we studied four popular tools to examine the effects of both configurations as a whole and of individual configuration options.

A. Study Setup

1) Subject Tools: Table I lists the verification tools we used in our study. We chose these tools because (1) they are among the best-performing tools in SV-COMP; (2) they come with configuration options that impact the tools' performance and precision; (3) they provide sufficient configuration option documentation so we can understand what individual option settings do; and (4) they target programs written in two widely-used programming languages. *CBMC* and *Symbiotic* verify C programs, and *JBMC* and *JayHorn* verify Java programs.

In our study, we focus on the options that affect analysis performance, soundness, and/or precision, instead of those that format output or toggle specific checkers. For options that take numerical values, we used a set of representative values from their domains. Columns 3 and 4 in Table I show the number of options we use and the number of possible configurations that can be created with them, respectively.

TABLE I: Subject verification tools.

Tool	Target lang.	# of options	Config space size	Sample size	Dataset size
CBMC 5.11	C	21	2.9×10^9	295	295,000
Symbiotic 6.1.0	C	16	9.8×10^5	82	54,940
JayHorn 0.6-a	Java	12	7.5×10^6	256	94,208
JBMC 5.10	Java	27	7.2×10^{10}	200	73,600

2) Configuration Sampling: The large number of options for our subject tools make it infeasible to study the tools' behaviors under all configurations (column 4 in Table I). Research on combinatorial interaction testing has shown that sampling configuration spaces using covering arrays is an effecive way to explore the behavior of configurable software [22], [23]. Furthermore, past research also indicates that changes in the behavior of software tend to be caused by interactions of only a few options [24]. We therefore create a 3-way covering array, which is a list of configurations that include all 3-way combinations of configuration options [22], for each tool, using an existing covering array generator [25]. Column 5 in Table I shows the number of sample configurations, i.e., the sizes of the covering arrays.

3) Target Programs: All of our target programs are taken from the SV-COMP competition [16]. To our knowledge, the SV-COMP program set is the largest collection of verification benchmarks for which the ground truth (i.e., whether the benchmark is safe or unsafe according to some property) is known. Furthermore, this program set aggregates multiple benchmarks from across the literature, increasing the generality of the conclusions we make. It consists of over 10000 benchmark programs in C and Java. In our study, we selected a subset on which we ran the verification tools with each sampled configuration.

For the two Java tools, we used all 368 benchmark programs from SV-COMP 2019. These Java programs are written with assertions, and the verification tools check if these assertions always hold. Among the 368 programs, 204 (55.4%) are known to be unsafe. For C tools, the SV-COMP 2018 benchmark has 9523 programs in total.[2] We randomly selected a subset of 1000 programs that are subject to only one verification check. Out of the 1000 programs we selected, there are 335 programs that are subject to concurrency safety verification, 41 to memory safety verification, 65 to integer overflow verification, 485 to reachability verification, and 74 to verification of termination. Among the 1000 programs, 517 are known to be unsafe.

4) Dataset Collection: We executed each sampled configuration of the subject tools once on each benchmark task to create the dataset for our study.[3] In each execution, we

[2]SV-COMP 2019 data were not available when we started this research. The benchmark set for C is mostly the same between 2018 and 2019.

[3]*Symbiotic* does not check concurrency safety. Thus, we did not run *Symbiotic* for the 335 programs that are subject to concurrency safety verification.

TABLE II: Results of the sample configurations.

Tool	# of verification tasks (i.e., programs)				
	All	Never solved	Correct / Incorrect / Inconclusive		
			worst-precision	most-correct	best-precision
CBMC	1000	42	524 / 456 / 20	635 / 262 / 103	426 / 1 / 573
Symbiotic	665	338	150 / 92 / 423	261 / 1 / 403	261 / 1 / 403
JayHorn	368	62	121 / 98 / 149	227 / 37 / 104	184 / 2 / 182
JBMC	368	2	159 / 204 / 5	331 / 0 / 37	331 / 0 / 37

used a 1-minute timeout. For the purpose of studying the configuration spaces, this timeout is sufficient because based on SV-COMP 2018 and 2019 results, 95%, 94%, 100%, and 85% of the conclusive runs took less than 1 minute for *CBMC*, *Symbiotic*, *JayHorn*, and *JBMC*, respectively [26], [27]. We say a verification run is *conclusive* if it outputs a judgement that the target program is safe (verified) or unsafe (rejected by the verifier). In total, we performed 517748 verification runs. The sizes of the datasets for each tool range from 54940 to 295000 runs (the last columns of Table I).

All experiments were conducted on an Ubuntu 16.04LTS machine with 24 Intel Xeon Silver 4116 CPUs @ 2.10GHz and 144GB RAM.

5) Research Questions: Our study answers two research questions:

- **RQ1:** Do the subject tools have any one-size-fits-all configurations?
- **RQ2:** What is the impact of individual configuration options on the tools' results?

RQ1 deals with the behavior of configurations as a whole. We used the dataset to determine whether any subject tool has a one-size-fits-all configuration, i.e., a configuration that can complete all verification tasks that were completed by at least one sampled configuration.

RQ2 focuses on the effects of individual configuration options. To address this research question, we aggregated the number of correct and incorrect verification results for each tool. We then performed a main effects screening analysis using ANOVA [28]. In this analysis, we treat the configuration options as cardinal or ordinal factors (i.e., independent variables) and the number of correct and incorrect results as the responses (i.e., dependent variables). We create two models for each tool, one for each response, using the least square method. Each model shows the effect that each factor has on the response along with standard error and the p-value. In our analysis, we consider the factors with statistically significant effects as those with $p\text{-}value<0.05$.

B. Study Results

1) RQ1: Do the subject tools have any one-size-fits-all configurations?: Table II presents results for the subject tools. Columns 2 and 3 show the total number of tasks to verify and the number of tasks the subject tool could not complete within the 1-minute timeout under *any* configuration. For each tool, we identified the *worst-precision-config*, the *most-correct-config*, and the *best-precision-config*. The worst-precision-config is the configuration which had the lowest precision (i.e., $\frac{\#\text{corrects}}{\#\text{corrects}+\#\text{incorrects}}$). The most-correct-config is the one which classified the most programs correctly, and the best-precision-config is the configuration that had the highest precision. Columns 4, 5, and 6 in Table II show the number of correct / incorrect / inconclusive results for the worst-precision-config, most-correct-config, and best-precision-config, respectively.

We find that *no tool has a single configuration that led to a superset of the completed tasks. Even the most-correct-config could not correctly verify 10% to 32% of the tasks that other configurations did.*

For *CBMC*, there is a large variance in the behavior of different configurations. The most-correct-config only completed 635 tasks (63.5%) correctly. However, in aggregate, 96% of the verification tasks could be completed correctly by some configuration of CBMC. We observe similar results for *JayHorn*. Its most-correct-config verified 227 (62%) tasks correctly, yet 83% of the tasks were correctly verified by some configuration. Furthermore, both configurations produce many more incorrect results than the best-precision-config For example, *CBMC*'s most-correct-config had 262 results compared to only 1 incorrect result using its best-precision-config. Depending on the user's requirements, this may be an unacceptable tradeoff.

The most-correct-configs for *Symbiotic* and *JBMC* are more promising. *Symbiotic*'s most-correct-config (which was also its best-precision-config) correctly verified 39% of tasks, and 49% of tasks that could be correctly completed by any configuration. *JBMC* also had the same best-precision-config and most-correct-config, which happened to be its default used in SV-COMP. This configuration completed 331 (90%) tasks correctly, with 0 incorrect results. *JBMC* could complete all but 2 (i.e., 99.5%) tasks correctly with some configuration.

The above findings suggest that *even for the tools with better single configurations, there is still significant room for improvement if the right configuration can be identified for a given verification task.*

2) RQ2: What is the impact of individual configuration options on the tools' results?: The results of the main effects screening analysis are summarized in Table III. Column 2 lists the option settings with statistically significant effects on one or both responses. The number next to each tool's name in Column 1 is the number of options with statistical significant settings. Overall, *for all verification tools, at most half of configuration options had at least one setting with a statistically significant effect on the verification results. Furthermore, a majority of such option settings presented tradeoffs, in that, they either increased or decreased both responses together* (highlighted as underlined blue in Table III).

Specifically, there were 12 (18%), 5 (20%), 10 (21%), and 9 (14%) option settings with statistically significant effects for *CBMC*, *Symbiotic*, *JayHorn*, and *JBMC*, respectively. More notably, for *CBMC*, *JayHorn*, and *JBMC*, 67%, 40%, and 56%

TABLE III: Configuration options with statistically significant effects on the verification results, ranked by the size of the estimated effect on the *Correct* response. Options in *underlined blue* present tradeoffs (i.e., increase or decrease both corrects and incorrects together).

Tool	Configuration option settings	Significant effects on response	
		Correct	Incorrect
CBMC (9)	*--partial-loops*	-82.64	-76.63
	--nondet-static	36.91	-19.16
	--paths def	49.22	19.24
	--full-slice	32.70	34.30
	--refine-strings	31.77	16.69
	--no-assumptions	18.88	20.82
	--solver z3	43.90	N/A
	--paths:fifo	-24.13	-14.32
	--solver boolvector	-36.57	N/A
	--depth 100	45.06	90.23
	--depth 1000	7.29	22.82
	--solver yices	-31.58	N/A
Symbiotic (5)	*--overflow-with-clang*	-29.69	14.42
	--explicit-symbolic	9.44	N/A
	--no-slice	13.93	-26.01
	--undefined-retval-nosym	6.10	N/A
	--repeat-slicing 2	-7.12	8.22
JayHorn (6)	*-initial-heap-size 100*	-47.65	-8.45
	-heap-mode bounded	-25.53	2.77
	-heap-mode auto	23.74	N/A
	-heap-limit 1	-44.90	24.05
	-solver eldarica	-39.09	-22.67
	-heap-limit 10	-10.14	2.75
	-initial-heap-size 10	24.37	N/A
	-inline-size 100	-19.40	-5.36
	-bounded-heap-size 10	16.75	N/A
	-step-heap-size 10	N/A	-6.43
JBMC (8)	*--path def*	80.56	30.76
	--localize-faults	-40.00	-37.54
	--paths fifo	-42.13	-13.65
	–java-threading	-15.07	N/A
	--full-slice	-16.09	9.19
	--slice-formula	12.17	7.45
	--depth 100	N/A	74.68
	--depth 1000	N/A	34.63
	--symex-driven-lazy-load	28.93	N/A

of such significant option settings presented tradeoffs. None of *Symbiotic*'s settings presented tradeoffs in our models.

As an example of the tradeoffs a settings can present, *CBMC*'s --partial-loops has an estimated effect of decreasing the number of correct and incorrect results by 83 and 77, respectively. This option allows partial execution of loops, which can make finding counterexamples at small unwinding bounds easier [29]. The drawback is that it may model spurious paths that do not exist in the original program, which could cause incorrect results.

We use two code examples in Figure 1 to illustrate the tradeoff --partial-loops presents. Both examples were extracted from the SV-COMP 2018 program set. In Figure 1a, P_1

```
1  int main() {
2    float x = 1.0f;
3    float x1 = x/2.5f;
4
5    while(x1 ≠x) {
6      x = x1;
7      x1 = x/2.5f;
8    }
9    assert(x == 0);
10   return 0;
11 }
```

(a) P_1 (safe)

```
1  #define N 1000000
2  int main() {
3    int i, a[N];
4    for (i=0; i<N; i++)
5      a[i] = 1;
6    for (i=0; i<N; i++)
7      a[i++] = 2;
8    for (i=0; i<N; i++)
9      a[i++] = 3;
10   for (i=0; i<N; i++)
11     assert(a[i] == 2);
12   return 0;
13 }
```

(b) P_2 (unsafe)

Fig. 1: Simplified code examples from the SV-COMP 2018.

(extracted from the program $Float\_div\_true\text{-}unreach\text{-}call.c$) is a safe program in that the assertion at line 9 always holds. This is because the loop at lines 5-8 keeps dividing x by 2.5 until it reaches a very small number that is below the sensitivity of the float type in C. The loop eventually ends as the pre- and post- division values become 0. In our dataset, 116 configurations incorrectly judged P_1 as unsafe, and they all set --partial-loops. This is because the analysis insufficiently unwinds the while loop, and the --partial-loops option allows the analysis to accept the partial loop execution as a valid path. To successfully verify P_1, a configuration needs to include a sufficient level of loop unwinding and disable --partial-loops.

On the other hand, P_2 (extracted from the program $standard\_init5\_false{-}unreach{-}call\_ground.c$) is an unsafe program in that the assertion at line 11 in Figure 1b never holds. In our dataset, P_2 was correctly judged as unsafe *only* by the 63 configurations that set --partial-loops. For P_2, analyzing all loop iterations is not necessary to determine that the assertion will fail as long as the first iteration of the loop at lines 8-9 is analyzed. Therefore, partially accepting loops is a safe assumption for P_2. The configurations that did not use this option (including the best-precision-config) spent too much time in loop unwinding and eventually timed out. The above examples illustrate that the effectiveness of a tool's configuration options may depend on the target program.

However, not all options present tradeoffs. In Table III, *some option settings (19 out of 36) have uniform effects*. These option settings can have uniformly positive effects (i.e., they increase the number of corrects and/or decrease the number of incorrects), or uniformly negative effects (i.e., they decrease the number of corrects and/or decrease the number of incorrects). For example, setting --explicit-symbolic in *Symbiotic* is estimated to produce 9 more correct results without a significant effect on the number incorrect results. This option makes *Symbiotic* initialize parts of memory with non-deterministic values. Without this option, evaluation is done with symbolic values, a costly step that requires tracking many more execution paths, causing *Symbiotic* to timeout.

Interestingly, all of *Symbiotic*'s options had uniform effects. One can use such uniform options in the configuration if the goal is to increase the likelihood of completing a verification

task. Indeed, we confirmed that the most-correct-config set 4 out of 5 of these options consistently with the models' estimated effects (e.g., disabling --overflow-with-clang). However, recall our answer to RQ1; doing so still cannot make a single configuration to complete the verification tasks all other configurations did.

In summary, not all configuration option settings have significant effects on the verification results. Those that do often present tradeoffs that depend on the target program, further supporting our argument that these tools likely do not have any one-size-fits-all configuration that would apply to all target programs.

III. The SATune Approach

The results of our empirical study motivated us to design an automated approach to tune the configuration spaces of the static verification tools that can successfully verify more programs. As shown in Section II, the four tools under evaluation have very precise configurations in their best-precision-configs. However, these configurations can complete fewer tasks relative to the total number of tasks all configurations can complete. Therefore, we designed SATune (for **S**imulated **A**nnealing Tune) with the goal of outperforming the tools' most-correct-config – in other words, to maximize the number of correct results.

At a high level, given a tool and target program, SATune searches through the tools' configuration space to find a configuration that is likely to complete with a correct verification result on the target program. We made three key design choices, based on the results of the study in Section II, that differentiate SATune from other approaches.

The first design choice is the adaptation of a *meta-heuristic search algorithm*. The findings in Section II-B demonstrate that there is no one-size-fits-all configuration for any tool. Thus, for each target program, it is necessary to explore the configuration space for a suitable configuration. However, it is infeasible to explore every configuration of a verification tool with even a modest number of configuration options. A meta-heuristic search algorithm probabilistically explores such search spaces to quickly locate a suitable configuration with which to run the tool for a given verification task. This choice is critical for both the efficacy and efficiency of SATune.

The second design choice is that of the fitness function, *f*. In order to perform a meta-heuristic search, we need a method to determine the fitness of a configuration – that way, the search knows whether a new candidate configuration is better than the configuration it currently has stored as the best one. The only way to know a configuration's true fitness would be to run the verification tool, observe whether it completes, and validate its result. However, the validation step may not even be possible to perform automatically, and even if it were, it would be prohibitively expensive to do repeatedly throughout the search. Instead, we use *f*, which is a learned model that provides an effective approximation of the fitness of a tool configuration (such approximations are commonly known as a surrogate fitness function in the literature [30]). Our model needs to be trained once for each tool, but then incurs little overhead when we query it during the search. The model steers the search toward configurations that are likely to complete a verification task with a correct result.

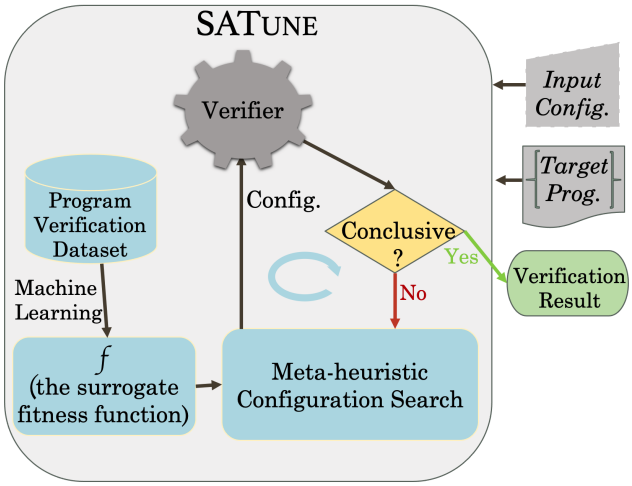

Fig. 2: Workflow of the SATune approach.

The third design choice is in the data we use to train *f*. As demonstrated in Section II-B, there are many configuration options that present tradeoffs in terms of producing correct and incorrect results, based on the specific target programs. These options should be evaluated for each verification task individually and set in a way that will increase the chance of getting a correct result. Thus, we train *f* not only on the results of configurations in the dataset, but also on the features of the target program, so that it can learn the ways that options interact with program features (see Section III-B).

Figure 2 shows the workflow of SATune. To use SATune, a user provides a target program and a verification tool with an input configuration.[4] SATune then runs the tool on the target program using the provided input configuration. If the run produces a conclusive result, it is reported to the user; in this case, SATune incurs no overhead. If the run does not lead to a conclusive result, the meta-heuristic configuration search begins. We now discuss two key components of SATune: the meta-heuristic configuration search and the the surrogate fitness function.

A. Meta-heuristic Configuration Search

There exist various different meta-heuristic search algorithms in the literature, such as tabu search [31], hill climbing [32], genetic algorithms [33], and simulated annealing [34]. Among them, simulated annealing [34] has been shown to be more effective in finding fitter objects (like covering arrays [25], [35]–[37] and orthogonal arrays [38]) in large and complex combinatorial spaces more quickly [22], [39], [40]. Considering the similarity in the search spaces of the mentioned successful applications, we decided to derive our

[4]In our evaluation, we use the best-precision-config as the input configuration (Section IV).

meta-heuristic configuration search algorithm from simulated annealing [34], [41], [42].

At a high level, simulated annealing is a stochastic search algorithm [43] that iteratively searches for a good solution by altering the current state to generate a new candidate state called a *neighboring state*. If the neighboring state is judged to be better than the current state according to some heuristic, it is *accepted* as the current state for the next search iteration. If the neighboring state is judged to be worse than the current state, it may still be *accepted probabilistically* to help the model avoid becoming stuck in local optima. The probability of selecting a worse configuration decreases over the search. This is done through the three control parameters: the *initial temperature* T_0, the *cooling rate* R by which the temperature (T) is reduced every iteration, and the *stopping temperature* T_s. Higher temperatures lead to higher probabilities of accepting inferior candidates (i.e., inferior candidates are more likely to be accepted in early iterations than in later iterations), which allows simulated annealing to be more flexible and exploratory early in the search process. The search ends either when an acceptable solution is found according to some criteria or the temperature falls below T_s.

Algorithm 1 depicts our meta-heuristic configuration search algorithm that adapts simulated annealing. The inputs to this algorithm are: (1) the tool's configuration space $CS = \langle O, D\rangle$ where O is the set of configuration options and D is their domains, such that $d_i \in D$ is the set of possible values that option $o_i \in O$ could take on (sampled for integer domains and exhaustive otherwise); (2) the target program P; and (3) the surrogate fitness function *f*.

Lines 2-7 perform initialization. First, we initialize the control parameters[5] as T_0=1, T_s=10^{-5}, R=10^{-4}. Then, we set the running temperature T to the starting temperature T_0, and the `isConclusive` flag is initialized as $\perp$ indicating an inconclusive result. At line 5, the current configuration c and the best configuration c^* are both initialized with the default configuration (or a randomly generated one if the default is not available). At line 6, the program representation vector V is initialized with the features extracted from P (see Section III-B). At line 7, *f* is used to compute the cost of c, E_c, using the concatenation of V and c as denoted by $\langle V{+\!\!+}c\rangle$. E_c is the probability of getting either an inconclusive or incorrect result if c were used to run the verification tool on the target program represented with features V.

Lines 8 to 18 implement an iterative search process that aims to select a configuration that is likely to complete the verification task with a correct result. On line 8, T_s<T checks that the temperature T has not decreased below the stopping temperature T_s [34], which is the standard stopping condition for simulated annealing. In addition, the search stops if the verification run is conclusive, because our goal is to find a configuration which will produce a conclusive verification result rather than an optimal one. During each iteration of the inner loop (lines 9-14), the algorithm first generates a new neighboring configuration c' (line 10). To generate a new neighboring configuration, we change the value of a single option o_i in the current configuration to another random value from its domain d_i. This simple approach has been shown to be effective for exploring large search spaces in a cost-effective manner in similar search problems (e.g., combinatorial testing [25], [35], [40], [44]). The algorithm then computes the cost of c' using *f* (line 11), and reduces the running temperature by the cooling rate (line 13).

This random configuration generation repeats until one of the acceptance conditions on line 14 is met: either ΔE<0, meaning c' is better than c according to *f*, or the algorithm decides to accept the inferior c' with probability $e^{-k\Delta E/T}$ [45]. This probability reduces with T (i.e., the model is more likely to accept inferior states early in the search). Once a state is accepted, c and E_c are updated (line 15). If c is the best so far (i.e., E_c<E_{c^*}), then c^* and E_{c^*} are also updated accordingly (lines 16-17). When a new c^* is found, we run the verification tool using c^* for the task P. If the verifier produces a conclusive result, the search ends and we return the result (line 19). Otherwise, the search continues to the next iteration.

Note that running the verification tool is the most expensive step of the Algorithm 1. In comparison, learning f, computing program representations, generating random configurations, and computing their cost take negligible time.

Algorithm 1 Meta-heuristic configuration search.

```
1: function ConfigSearch(CS= ⟨O, D⟩, P, f)
2:     T_0 ← 1; T_s ← 10^-5; R ← 10^-4        ▷ control parameters
3:     T ← T_0
4:     isConclusive ← ⊥
5:     c* ← c ← getRandomOrDefault(CS)
6:     V ← getProgramRepresentation(P)
7:     E_c* ← E_c ← f(⟨V ++ c⟩)                ▷ cost for ⟨V++c⟩
8:     while T_s < T ∧ ¬isConclusive do
9:         repeat
10:            c' ← getNeighboringConfig(CS, c)
11:            E_c' ← f(⟨V++c'⟩)
12:            ΔE ← E_c' − E_c
13:            T ← T − (T × R)
14:        until ΔE < 0 ∨ rand(0,1) < e^(−kΔE/T)
15:        c, E_c ← c', E_c'                   ▷ accept
16:        if E_c < E_c* then
17:            c*, E_c* ← c, E_c               ▷ best config so far
18:            isConclusive ← runVerifier(P, c)
19: return ⟨isConclusive, c*⟩
```

B. Learning the Surrogate Fitness Function

We learn the surrogate fitness function *f* using the dataset we created for the empirical study (Section II). In this dataset, each data point is of the form $\langle V{+\!\!+}c\rangle$=$X$ where $X \in$ {*correct*, *incorrect*, *inconclusive*} is the verification result. By including both the program features and configuration, our

[5]We empirically determined these values with preliminary experiments that showed that T_0 and T_s did not impact the performance significantly, while R did. Larger R values (0.01, 0.001) caused the search end too quickly without sufficient exploration while smaller values (10^{-5}) caused longer search times.

models can learn from the interactions between them. f is trained to differentiate between data points with either an *incorrect* or *inconclusive* verification result and data points with a *correct* verification result. Effectively, f returns the probability of producing an incorrect or inconclusive for a given $\langle V + c \rangle$ combination as its cost. Formally,

$$E_c = f(\langle V + c \rangle) = P[incorrect \mid \langle V + c \rangle] + P[inconclusive \mid \langle V + c \rangle]$$

Recall that Algorithm 1 aims at minimizing E_c, which translates to locating configurations that are more likely to produce conclusive and correct results.

Past research has applied a variety of models and features to learn from program code [46]–[49]. In this work, we use a simple Bag of Words model [50] for its simplicity and relative efficacy in representing programs for classifying static analysis results [48]. We represent a program as a frequency vector by counting the frequencies of program instructions and certain constructs such as branches and loops. This simplicity makes our approach to be language-agnostic (i.e., extendable to any programming language by identifying the relevant program instructions and constructs).

C. Implementation

We instantiated SATUNE to tune the configuration spaces of *CBMC*, *Symbiotic*, *JayHorn*, and *JBMC*. We learned the fitness functions with a random forest algorithm from Weka [51]–[53]. To construct the Bag of Words model, we counted the occurrence of intermediate representation (IR) instruction types and different program features (e.g., loops, branches, and function calls) in our benchmarks. The program features were collected via simple static analyses for C and Java. C programs (targeted by *CBMC* and *Symbiotic*) are represented by frequency vectors for 11 program features and 47 LLVM IR instructions [54]. Java programs (targeted by *JayHorn* and *JBMC*) are represented by frequency vectors of 9 program features and 23 WALA IR instructions [55]. LLVM and WALA are popular frameworks for the analysis of C and Java programs, respectively.

We implemented SATUNE in 600 lines of Java code with a command-line interface (`cli`). The SATUNE `cli` takes a verification tool, a target program to be verified, and the initial configuration as input.

IV. Evaluation

In this section, we discuss the experimental setup for evaluating SATUNE and answer two research questions on how well it performs.

A. Experimental Setup

We trained SATUNE's fitness function on the dataset we generated in Section II. We then evaluated it against other potential strategies a user may use to select a configuration to run the verification tools with, in terms of both correctness of the results and the time it takes to run.

1) Training of SATUNE*'s fitness function:* We split the dataset of each verification tool into five equal partitions that are disjoint by benchmark programs. Four of them are used for training the fitness function, *f*, while one partition is held out to evaluate SATUNE (which used the *f* internally). We then rotated the partitions and repeated this process 25 times to perform 5-fold cross-validation [56] with five different random seeds. Since these repetitions allow all of the data to be used for both training and evaluation (at different iterations), we were able to evaluate SATUNE on the entire dataset.

In total, we trained 100 surrogate fitness functions (5 random seeds $\times$ 5-fold cross-validation $\times$ 4 tools). We report the total number of conclusive results across all cross-validation sets as the SATUNE's results.

2) Comparison Baselines: To the best of our knowledge, SATUNE is the first tool- and language-agnostic approach that automatically configures program verification tools. We designed three baselines that simulate ways a user might use and tune a verification tool.

The first baseline simulates a user who would only try a single configuration of a tool and accept whatever outcome that configuration gives. We assume the user does not have extensive domain expertise. Rather than manually tuning the tool for a target program, they simply try a configuration that does well overall. In our evaluation, we use the *most-correct-config* of each tool as this baseline.

The second baseline simulates a user who has more time to try to get a correct result. In this case, the user first tries a highly precise configuration (e.g., best-precision-config) and if the result is inconclusive, tries again on a less precise but performant configuration. In our evaluation, we first run each tool's best-precision-config. If it produces a conclusive result, we report it. Otherwise, we fall back to run the most-correct-config and report the result. We call this baseline *precision→correct*. Because the best-precision-config and most-correct-config are the same for *Symbiotic* and *JBMC* (Section II), the results of *precision→correct* are the same as *most-correct-config* for these tools.

Finally, the third baseline simulates a user whose target program may be difficult for a verification tool to complete. The user also has a large amount of time to experiment with different configurations to find one that may work. In this baseline, we start by using the best-precision-config. If it fails to complete within the timeout, a random search begins based off of the best-precision-config. The neighbor generation strategy is the same as SATUNE, in that for each iteration we randomly alter a single setting of a configuration option. The random search continues until it finds a configuration that finishes within the time limit, or it reaches 60 attempts. We call this baseline *precision→random*.

3) Metrics: We use three metrics in our evaluation: (1) the number of correct verification results, (2) precision (i.e., the percentage of conclusive results that are correct), and (3) the total run time to complete each verification task in minutes. Each experiment was repeated five times, and we report the

TABLE IV: Results for SATUNE and three baselines. The numbers in normal font are median of 5 runs (with different random seeds for SATUNE and *precision→random*), and the numbers in the smaller font are the semi-interquartile range (SIQR). For *Symbiotic* and *JBMC*, the results of *precision→correct* are not shown because they are the same as the *most-correct-config*.

Tool	Approach	Correct		Incorrect		Inconclusive		Precision
CBMC	SATUNE	804	12	196	13	0	3	80.40%
	precision→random	738	13	195	8	67	13	79.10%
	precision→correct	704	0	262	0	34	34	72.88%
	most-correct-config	635	0	262	0	103	0	70.79%
Symbiotic	SATUNE	277	2	1	0	387	2	99.64%
	precision→random	264	1	1	0	400	1	99.62%
	most-correct-config	261	0	1	0	403	0	99.62%
JayHorn	SATUNE	240	0	13	2	115	2	94.86%
	precision→random	211	4	14	2	143	2	93.79%
	precision→correct	247	0	38	0	83	0	86.67%
	most-correct-config	227	0	37	0	104	0	85.98%
JBMC	SATUNE	348	2	7	3	13	1	98.08%
	precision→random	343	2	6	1	19	1	98.28%
	most-correct-config	331	0	0	0	37	0	100%

median and semi-interquartile range (SIQR) values for the first two metrics.

4) Research Questions: Our evaluation aims to answer two research questions:

- **RQ3:** Can SATUNE correctly verify more programs?
- **RQ4:** How efficient is SATUNE?

RQ3 compares SATUNE to the three baselines in terms of number of correct results and precision. Due to the randomness in both approaches, we also performed statistical analysis to determine whether the results produced by *precision→random* are significantly different from those produced by SATUNE. **RQ4** aims to determine how efficient SATUNE's meta-heuristic configuration search is. We explore the distribution of execution times to determine, for each tool, how SATUNE compares to the three baselines.

B. Experimental Results

1) RQ3: Can SATUNE *correctly verify more programs?:* Table IV presents the results of SATUNE and the three baselines for each tool using median and SIQR metrics. Overall, we find that *compared to the baselines,* SATUNE *consistently achieves the best balance between the number of correct results and precision*.

In all tools but *JayHorn*, SATUNE completes more tasks correctly than any other baseline strategy. In *Symbiotic*, SATUNE produced 16 and 13 more correct results than *most-correct-config* and *precision→random*, respectively, without any more incorrect results. In *JayHorn*, *precision→correct* completed 7 more tasks correctly than SATUNE, but at the cost of 25 more incorrect results. Notably, SATUNE allowed *CBMC* to complete every task, at the cost of only a single more incorrect result than the next-best baseline (*precision→random*).

In terms of precision, SATUNE achieved higher precision than all other baselines for every tool but *JBMC*. SATUNE was still highly precise in *JBMC* (98.08%), but recall that *JBMC*'s best-precision-config achieved 100% precision. Still, SATUNE was able to complete 17 more verification tasks correctly than *most-correct-config* in *JBMC*.

Interestingly, we found that SATUNE was able to correctly complete some verification tasks that no single configuration in our study could (i.e., tasks that are in the "never solved" column of Table II). SATUNE correctly completed 1, 8, 1, and 3 such tasks for *CBMC*, *Symbiotic*, *JBMC*, and *JayHorn*, respectively. This suggests that SATUNE's fitness function was able to generalize to configurations it had not previously seen in the training data.

Last, we discuss the variations in the results using the SIQR metric. Overall, SATUNE and *precision→random* had small variations in their results while *most-correct-config* and *precision→correct* had none. It is expected that no variation is present in the *most-correct-config* and *precision→correct* results because both are deterministic. The variations of SATUNE and *precision→random* are due to the randomness in their search process. For SATUNE, its largest variation from the *CBMC* results is still relatively small, accounting for about 1% of the total number of tasks.

2) RQ4: How efficient is SATUNE*?:* Figure 3 illustrates the distribution of the execution times to verify each program (y-axis in logarithmic scale) as box-plots for all of our experiments. Each box-plot represents the conclusive verification runs of a tool using SATUNE or a baseline approach. The width of the box-plots reflects the population size, i.e., the number of correct + incorrect results (shown in Table IV). 50% of the data points fall inside the box. The line inside the box is the median, and the lower and upper ends of the box correspond to the first and third quartiles, respectively. The red dot and number show the longest time it takes for each approach to complete one task. The other dots on the central line of each box show the outliers.

We found that all four approaches were relatively fast for most tasks, with 75% of tasks being completed in under a minute in all cases. Still, we see that *even in the worst case (red dots),* SATUNE *was 2-4x faster than precision→random*.

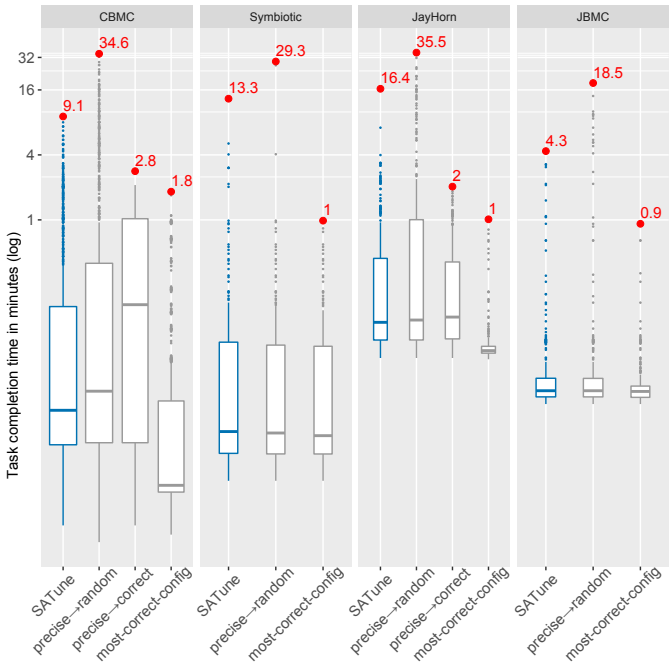

Fig. 3: Execution time of SATUNE and the baselines.

This is attributable to SATUNE's fitness function – specifically, the fact that it screens configurations in advance and only runs them if they are judged to be fit. In *JayHorn*, where SATUNE had the highest median and maximum run time, it generated a median of 143 configurations and only ran 3 of them. In contrast, random search generated and ran a median of 9 configurations.

Figure 3 also shows that *most-correct-config* was the only baseline that consistently did better than SATUNE in terms of median execution time. This is expected since *most-correct-config* only runs a single configuration. More notably is that SATUNE is comparable to (or, in the case of *CBMC*, outperforms) *precision→correct*, which only runs two configurations. These findings demonstrate the efficiency of SATUNE and the significant advantage its fitness function gives it over other search strategies.

V. Threats to Validity

Here we enumerate the potential threats to the validity of our work and the steps we took to mitigate them. First, the benchmark dataset we used from SV-COMP is primarily composed of artificial benchmarks. Thus, the conclusions we made about the relative quality of SATUNE compared to other baselines may not generalize to large, real-world programs. Unfortunately, we are unaware of a large real-world benchmark for which the ground truths are known, but we believe the large number of programs we used and the diversity of the SV-COMP benchmark mitigate this potential threat. Second, the configuration samples in our empirical study may not be representative of the tools' full configuration spaces. Our configuration samples include every three-way interaction of configuration options, and past research in configurable software indicates that the majority of program behaviors are attributable to the interaction of few options [24]. Finally, there could be variance of the performance of SATUNE (and *precision→random*) caused by non-deterministic operations in machine learning and configuration selection. We mitigated this potential threat by running 5 replications of each experiment, and reporting the median and semi-interquartile range values which suggested small variations (RQ3).

VI. Related Work

To the best of our knowledge, this work is the first to use meta-heuristic search and machine learning to tune software verification tools with large configuration spaces. Our work is related to work that (1) studies the configuration spaces of analysis tools, (2) selects a static analysis tool or a configuration of a static analysis tool that is most suited to a given task, (3) selectively applies algorithms in a static analysis, (4) uses machine learning models as fitness functions in meta-heuristic search, and (5) tunes high performance computing systems for a given system architecture and hardware.

Studies of Tool Configuration Spaces. We believe we are the first work to systematically study the configuration spaces of static program verification tools. Other work has engaged in similar goals with other types of static analyzers, specifically focusing on the tradeoffs presented by different configurations and configuration option settings [1]–[3]. Wei et al. present a study that evaluates the tradeoffs in the 162 different configurations of a numerical static analysis for Java programs [2]. Smaragdakis et al. [1] and Lhoták and Hendren [3] instantiate multiple variants of context-sensitive points-to analysis for Java in order to understand the tradeoffs of different design decisions. The tools we studied present much larger configuration spaces than those in the past studies that required us to apply statistical analysis to understand the impact of configuration options.

Configuration and tool selection. Our work is also highly relevant to those that select strategies (i.e., full configurations) within a static analysis tool [6], [10], predict or rank which static analysis tool is suitable for a given task [11]–[13], and research within software product lines (SPL) that attempts to learn performance models of tool configuration spaces [57]–[61]. Beyer and Dangl present a selection approach that uses four manually defined binary program features to select between three manually defined verification strategies for CPACHECKER [6]. Richter and Wehrheim present PESCO, which uses machine learning to rank five CPACHECKER verifiers [10]. SATUNE differs from these efforts in that it explores large and complex configuration spaces in a tool- and language-agnostic way using meta-heuristics, instead of using predefined configurations, manually defined heuristics, or ranking tools.

Other works [11]–[13] have explored selecting a verification tool out of a list of tools that would be the most appropriate for a given task. For example, Tulsian et al. present MUX, a machine learning-based approach that uses features extracted

from Windows device drivers to select the fastest verification tool to analyze them [11]. Our research complements these works by selecting a configuration of a single static analysis tool.

Finally, some SPL research is closely related to our work, in that these works use machine learning or statistical methods to model the effects of setting configurations. SPLCONQUEROR is a well-known tool that tries to compute the optimal configurations of a tool, given a target metric (e.g., precision or run time) [58]. Similarly, Ha and Zhang's DeepPerf models a tool's configuration space by training deep neural networks [61]. While SATUNE also aims to select configurations, it also takes into account the features of the target program, which allows it to consider the tradeoffs configuration options of the static program verification tools present.

Selective Static Analysis. Other related work selectively applies static analysis algorithms to parts of the target program. Among analysis algorithms, context sensitivity is the most studied [5], [8], [9], [14]. For example, Wei and Ryder present an adaptive context-sensitive analysis that uses eight features extracted from the points-to and call graphs of JavaScript programs [8]. Other algorithms such as flow sensitivity have also been used to develop selective static analysis [62]. Our work similarly studies the relationship between program features and analysis algorithms to achieve a good balance between performance, precision, and soundness, but we consider a wide range of configuration options while being agnostic to the analysis algorithm. In addition, instead of developing new analysis algorithms or tools, our approach automatically configures existing verification tools.

ML Models as Fitness Functions. Several researchers explore the use of machine learning models as "surrogate" fitness functions. Brownlee et al. demonstrated that a Markov network could be an effective surrogate fitness function in genetic algorithms for feature selection in Case-Based Reasoning [63]. Jin and Sendhoff use ensembles of neural networks to improve the performance of evolutionary algorithms [64]. Singh et al. evaluated both regression models and radial basis functions as surrogate fitness functions in simulated annealing [65]. While these papers focus on improving the meta-heuristic algorithms, we are not aware of any other work that automatically learns surrogate fitness functions for tuning verification tools with large configuration spaces.

High Performance Computing. Lastly, a distantly related line of work includes automatically tuning high performance computing (HPC) systems for a given system architecture and hardware. Agakov et al. use machine learning to perform iterative optimization for HPC systems at compile time [66]. Ansel et al. present an extensible framework, OPENTUNER, that enables writing domain-specific HPC tuners [67]. For a more comprehensive review of the literature on HPC system tuning, we refer readers to a recent survey by Ashouri et al [68]. Our work differs from the work above in that we use a meta-heuristic search augmented with a surrogate fitness function for tuning program verification tools–which are not HPC systems–to get a desired verification outcome.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented SATUNE, the first auto-tuning approach for static program verification tools with large configuration spaces. The design of SATUNE is motivated by an empirical study that provided important insights on the characteristics of configuration spaces and the impacts of individual configuration options on the precision and overall correctness of the verification tools' results. First, we demonstrated that there is no one-size-fits-all configuration in any tool. Even the most-correct-config could not complete certain tasks that other configurations did. Second, we found that many configuration options present tradeoffs between precision and performance, and they should be tuned individually for given target programs to get the most out of the tools' capabilities. SATUNE is novel in that it uses a simple meta-heuristic search algorithm with surrogate fitness functions learned from data to explore large configuration spaces and avoid running configurations that are likely to produce incorrect results. It is tool- and language-agnostic. We applied SATUNE to four popular verification tools for both C and Java programs and evaluated its performance using the ground-truth datasets. The evaluation shows that SATUNE achieves the best balance between performance and precision improvements compared to the baselines we used.

In future work, we will integrate other machine learning techniques, such as neural networks, into SATUNE to train the surrogate fitness function. This will elide the need to manually identify and extract program features, and enable SATUNE to take advantage of more complex structural information that neural networks can potentially learn. We will also extend the configuration generation step of SATUNE to incorporate the findings from our empirical study in Section II-B. This will enable more effective scanning of the configuration space and help improve the tools' precision by better avoiding configurations that are likely to lead to incorrect results. We will also extend our dataset and apply SATUNE to additional tools targeting other programming languages.

REFERENCES

[1] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," *SIGPLAN Not.*, vol. 46, no. 1, p. 17–30, Jan. 2011. [Online]. Available: https://doi.org/10.1145/1925844.1926390

[2] S. Wei, P. Mardziel, A. Ruef, J. S. Foster, and M. Hicks, "Evaluating design tradeoffs in numeric static analysis for java," in *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Cham: Springer International Publishing, 2018, pp. 653–682. [Online]. Available: https://doi.org/10.1007/978-3-319-89884-1_23

[3] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, Oct. 2008. [Online]. Available: https://doi.org/10.1145/1391984.1391987

[4] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association

for Computing Machinery, 2015, p. 307–319. [Online]. Available: https://doi.org/10.1145/2786805.2786852

[5] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 100:1–100:28, Oct. 2017.

[6] D. Beyer and M. Dangl, "Strategy Selection for Software Verification Based on Boolean Features," in *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 144–159.

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

[8] S. Wei and B. G. Ryder, "Adaptive context-sensitive analysis for javascript," in *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 712–734. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

[9] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 129–140.

[10] C. Richter and H. Wehrheim, "Pesco: Predicting sequential combinations of verifiers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer. Cham: Springer International Publishing, 2019, pp. 229–233.

[11] V. Tulsian, A. Kanade, R. Kumar, A. Lal, and A. V. Nori, "Mux: algorithm selection for software model checkers," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ACM. New York, NY, USA: ACM, 2014, pp. 132–141.

[12] M. Czech, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim, "Predicting Rankings of Software Verification Tools," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*, ser. SWAN 2017. New York, NY, USA: ACM, 2017, pp. 23–26.

[13] Y. Demyanova, T. Pani, H. Veith, and F. Zuleger, "Empirical software metrics for benchmarking of verification tools," *Formal Methods in System Design*, vol. 50, no. 2, pp. 289–316, Jun. 2017.

[14] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 141:1–141:29, Oct. 2018.

[15] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 153–169.

[16] D. Beyer, "Automatic verification of c and java programs: Sv-comp 2019," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 133–155.

[17] D. Kroening and M. Tautschnig, "Cbmc – c bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391.

[18] J. Slabỳ, J. Strejček, and M. Trtík, "Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution," in *International Workshop on Formal Methods for Industrial Critical Systems*, Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 207–221.

[19] J. Slaby, J. Strejček, and M. Trtík, "Symbiotic: synergy of instrumentation, slicing, and symbolic execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer. Cham: Springer International Publishing, 2013, pp. 630–632.

[20] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "JBMC: A bounded model checking tool for verifying Java bytecode," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 10981. Cham: Springer International Publishing, 2018, pp. 183–190.

[21] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf, "Jayhorn: A framework for verifying java programs," in *International Conference on Computer Aided Verification*, Springer. Cham: Springer International Publishing, 2016, pp. 352–358.

[22] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, vol. 43, pp. 11:1–11:29, February 2011.

[23] C. Yilmaz, S. Fouché, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc, "Moving Forward with Combinatorial Interaction Testing," *Computer*, vol. 47, no. 2, pp. 37–45, Feb. 2014.

[24] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2580950

[25] U. Koc and C. Yilmaz, "Approaches for computing test-case-aware covering arrays," *Software Testing, Verification and Reliability*, vol. 28, no. 7, p. e1689, 2018.

[26] "Results of the competition," 2019, https://sv-comp.sosy-lab.org/2019/results/results-verified/, Accessed: 2021-04-22.

[27] "Results of the competition," 2018, https://sv-comp.sosy-lab.org/2018/results/results-verified/, Accessed: 2021-04-22.

[28] T. Speed, "Introduction to fisher (1926) the arrangement of field experiments," in *Breakthroughs in statistics*. New York, NY: Springer New York, 1992, pp. 71–81.

[29] "Cbmc, understanding loop unwinding," 2014, http://www.cprover.org/cprover-manual/cbmc/unwinding 2021-04-019.

[30] A. E. Brownlee, J. R. Woodward, and J. Swan, "Metaheuristic design pattern: Surrogate fitness functions," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Companion '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1261–1264.

[31] F. Glover and M. Laguna, *Tabu Search*. Boston, MA: Springer US, 1998, pp. 2093–2229.

[32] B. Selman and C. P. Gomes, "Hill-climbing search," *Encyclopedia of cognitive science*, vol. 81, p. 82, 2006.

[33] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *Computer*, vol. 27, no. 6, p. 17–26, Jun. 1994.

[34] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[35] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *Proc. of the 14th Int'l Symposium on Software Reliability Engineering*, ser. ISSRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 394–405.

[36] R. C. Bryce and C. J. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1082–1089.

[37] H. Mercan, C. Yilmaz, and K. Kaya, "Chip: A configurable hybrid parallel covering array constructor," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1270–1291, 2018.

[38] R. Wang and R. Safadi, "Generating mixed multilevel orthogonal arrays by simulated annealing," in *Computing Science and Statistics*. Springer, 1992, pp. 557–560.

[39] J. Stardom, *Metaheuristics and the search for covering and packing arrays*. Simon Fraser University Burnaby, 2001.

[40] J. Torres-Jimenez and E. Rodriguez-Tello, "New bounds for binary covering arrays using simulated annealing," *Information Sciences*, vol. 185, no. 1, pp. 137–152, 2012.

[41] P. J. Van Laarhoven and E. H. Aarts, *Simulated annealing*. Dordrecht: Springer Netherlands, 1987.

[42] V. Granville, M. Krivánek, and J.-P. Rasson, "Simulated annealing: A proof of convergence," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, no. 6, pp. 652–656, 1994.

[43] F. Neumann and C. Witt, *Stochastic Search Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 21–32.

[44] B. S. Gulavani and S. K. Rajamani, "Counterexample Driven Refinement for Abstract Interpretation," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Berlin, Heidelberg, Mar. 2006, pp. 474–488.

[45] A. Bach, "Boltzmann's probability distribution of 1877," *Archive for History of Exact Sciences*, vol. 41, no. 1, pp. 1–40, 1990.

[46] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting Accurate Method and Class Names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49.

[47] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *Proceedings of the 1st ACM SIGPLAN International Workshop*

on Machine Learning and Programming Languages, ser. MAPL 2017. New York, NY, USA: ACM, 2017, pp. 35–42.

[48] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, IEEE. USA: IEEE Computer Society, 2019, pp. 288–299.

[49] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided Selectively Unsound Static Analysis," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 519–529.

[50] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[51] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[52] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[53] E. Frank, M. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg, *Weka-A Machine Learning Workbench for Data Mining*. Boston, MA: Springer US, 2010, pp. 1269–1277.

[54] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, IEEE. USA: IEEE Computer Society, 2004, pp. 75–86.

[55] IBM, "T. J. Watson Libraries for Analysis (WALA)," http://wala.sourceforge.net/, 2006.

[56] R. R. Picard and R. D. Cook, "Cross-validation of regression models," *Journal of the American Statistical Association*, vol. 79, no. 387, pp. 575–583, 1984.

[57] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1826–1867, 2018.

[58] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 284–294.

[59] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[60] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "Spl conqueror: Toward optimization of non-functional properties in software product lines," *Software Quality Journal*, vol. 20, no. 3, pp. 487–517, 2012.

[61] H. Ha and H. Zhang, "Deepperf: Performance prediction for configurable software with deep sparse neural ne twork," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1095–1106.

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

[63] A. E. I. Brownlee, O. Regnier-Coudert, J. A. W. McCall, and S. Massie, "Using a markov network as a surrogate fitness function in a genetic algorithm," in *IEEE Congress on Evolutionary Computation*. Barcelona, Spain: IEEE, 2010, pp. 1–8.

[64] Y. Jin and B. Sendhoff, "Reducing fitness evaluations using clustering techniques and neural network ensembles," in *Genetic and Evolutionary Computation – GECCO 2004*, K. Deb, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 688–699.

[65] H. K. Singh, T. Ray, and W. Smith, "Surrogate assisted simulated annealing (sasa) for constrained multi-objective optimization," in *IEEE Congress on Evolutionary Computation*. Barcelona, Spain: IEEE, 2010, pp. 1–8.

[66] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. USA: IEEE Computer Society, 2006, p. 295–305.

[67] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 303–316.

[68] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, vol. 51, no. 5, Sep. 2018.