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

Ugur Koc*  Austin Mordahl†  Shiyi Wei†  Jeffrey S. Foster‡  Adam A. Porter*

*Department of Computer Science, University of Maryland, College Park, MD, USA
{ukoc, aporter}@cs.umd.edu
†Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA
{austin.mordahl, swei}@utdallas.edu
‡Department of Computer Science, Tufts University, Medford, MA, USA
jeffrey.foster@tufts.edu

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 step towards automating the process of finding them.

Index Terms—Empirical software engineering; software analysis; testing, verification, and validation.

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 tradeoffs, program verification 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 depend 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]–[15] and have selectively applied settings of the analysis configuration options [5], [8], [9], [16]. Other work aims to develop an understanding of certain kinds of configuration options in static analysis frameworks or tools [2], [3], [17]. To our knowledge, the prior work has focused on relatively small and specific configuration or tool 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 large configuration spaces of software verification tools.

Our empirical study examines four tools that participate in the annual software verification competition (SV-COMP) [18]: CBMC [19] and Symbiotic [20], [21] verify C/C++ programs, and JBMC [22] and JayHorn [23] verify Java programs. We created a ground-truth 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

1In 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.
significant effect on the number of correct or incorrect results produced by a tool. In other words, 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, 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 \( \frac{\#corrects}{\#corrects + \#incorrects} \)) 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 2–4× 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 have made our dataset, results, and the implementation of SATUNE publicly available at [https://zenodo.org/record/5218510](https://zenodo.org/record/5218510).

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

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 configuration options that affect analysis performance, soundness, and/or precision, instead of those that format output or toggle specific checkers.

Column 3 in Table I shows the number of options we use in each verification tool. For each option, we identified its domain (i.e., the settings it can take on). In terms of domain, there were three different types of options in these verification tools. First, some options may be passed to the tool alone as an argument; i.e., as boolean flags. Inherently, a boolean flag option has two possible settings: \{set, unset\}. For instance, CBMC has 11 boolean flag options, such as --pm, which allows CBMC to model paths that only partially execute loops, rather than fully unwinding them [24]. Second, options may have categorical settings. CBMC has 6 such options, such as --mm, which specifies the memory model for concurrent applications [19]. This option can take on any of the following settings: \{sc, tso, pso\}. Finally, some options take on numerical values. Due to the large number of settings, for numerical options we use a set of representative values from their domains. For instance, the --unwind option of CBMC (which specifies the depth to which loops should be unwound) accepts a positive integer; we considered the following values for this option: \{1, 5, 10, 20, 100\}. We included all settings of boolean and categorical options. Each specific combination of option settings is a configuration. Column 4 shows the number of possible configurations that can be created with the options and settings that we use.

1) Configuration Sampling: The large number of options for our subject tools makes 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 effective way to explore the behavior of configurable software [25], [26]. Furthermore, past research also indicates that changes in the behavior of software tend to be caused by interactions of only a few options [27]. We therefore create a 3-way covering array, which is a list of configurations that include all 3-way combinations of configuration option settings [25], for each tool, using an existing covering array generator [28]. Column 5 in Table I shows the number of sample configurations, i.e., the sizes of the covering arrays.

2) Target Programs: All of our target programs are taken from the SV-COMP competition [18]. To our knowledge, the SV-COMP program set is the largest collection of verification benchmarks for which the ground truths (i.e., whether the benchmark is safe or unsafe according to some property)
are known. Furthermore, this program set aggregates multiple benchmarks from across the literature, increasing the generality of the conclusions we make. It consists of over 10,000 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 9,523 programs in total.\(^3\) We randomly selected a subset of 1,000 programs that are subject to only one verification check. Out of the 1,000 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 1,000 programs, 517 are known to be unsafe.

3) Dataset Collection: We executed each sampled configuration of the subject tools once on each benchmark task to create the dataset for our study.\(^4\) In each execution, we 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.\(^2\) 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 517,748 verification runs. The sizes of the datasets for each tool range from 54,940 to 295,000 runs (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.

4) 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 option settings 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 option settings. 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\(^5\). 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-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., precision value < 0). 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 could correctly verify all tasks other configurations did. 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 incorrect results.

### Table I: Subject verification tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Target lang.</th>
<th># of options</th>
<th>Config space size</th>
<th>Sample size</th>
<th>Dataset size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBMC 5.11</td>
<td>C</td>
<td>21</td>
<td>2.9 × 10⁹</td>
<td>295</td>
<td>295,000</td>
</tr>
<tr>
<td>Symbiotic 6.1.0</td>
<td>C</td>
<td>16</td>
<td>9.8 × 10⁵</td>
<td>82</td>
<td>54,940</td>
</tr>
<tr>
<td>JayHorn 0.6-a</td>
<td>Java</td>
<td>12</td>
<td>7.5 × 10⁹</td>
<td>256</td>
<td>94,208</td>
</tr>
<tr>
<td>JBMC 5.10</td>
<td>Java</td>
<td>27</td>
<td>7.2 × 10¹⁰</td>
<td>200</td>
<td>73,600</td>
</tr>
</tbody>
</table>

### Table II: Results of the sample configurations.

<table>
<thead>
<tr>
<th>Tool</th>
<th># of verification tasks (i.e., programs)</th>
<th>Correct / Incorrect / Inconclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>Never solved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>worst-precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>most-correct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>best-precision</td>
</tr>
<tr>
<td>CBMC</td>
<td>1000</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>524 / 456 / 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>635 / 262 / 103</td>
</tr>
<tr>
<td></td>
<td></td>
<td>426 / 1 / 573</td>
</tr>
<tr>
<td>Symbiotic</td>
<td>665</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td></td>
<td>150 / 92 / 423</td>
</tr>
<tr>
<td></td>
<td></td>
<td>261 / 1 / 403</td>
</tr>
<tr>
<td></td>
<td></td>
<td>261 / 1 / 403</td>
</tr>
<tr>
<td>JayHorn</td>
<td>368</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>121 / 98 / 149</td>
</tr>
<tr>
<td></td>
<td></td>
<td>227 / 37 / 104</td>
</tr>
<tr>
<td></td>
<td></td>
<td>184 / 2 / 182</td>
</tr>
<tr>
<td>JBMC</td>
<td>368</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>159 / 204 / 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>331 / 0 / 37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>331 / 0 / 37</td>
</tr>
</tbody>
</table>
TABLE III: Configuration option settings 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). Options without a setting (e.g., --partial-loops of CBMC) are boolean flag options.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Configuration option settings</th>
<th>Significant effects on response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Correct</td>
</tr>
<tr>
<td>CBMC (9)</td>
<td>--partial-loops</td>
<td>-82.64</td>
</tr>
<tr>
<td></td>
<td>--nondet-static</td>
<td>36.91</td>
</tr>
<tr>
<td></td>
<td>--paths def</td>
<td>49.22</td>
</tr>
<tr>
<td></td>
<td>--full-slice</td>
<td>32.70</td>
</tr>
<tr>
<td>Symbiotic (5)</td>
<td>--refine-strings</td>
<td>31.77</td>
</tr>
<tr>
<td></td>
<td>--no-assumptions</td>
<td>18.88</td>
</tr>
<tr>
<td></td>
<td>--solver z3</td>
<td>43.90</td>
</tr>
<tr>
<td></td>
<td>--paths fifo</td>
<td>-24.13</td>
</tr>
<tr>
<td></td>
<td>--solver boolvector</td>
<td>-36.57</td>
</tr>
<tr>
<td></td>
<td>--depth 100</td>
<td>45.06</td>
</tr>
<tr>
<td></td>
<td>--depth 1000</td>
<td>7.29</td>
</tr>
<tr>
<td></td>
<td>--solver yices</td>
<td>-31.58</td>
</tr>
<tr>
<td>JayHorn (6)</td>
<td>--overflow-with-clang</td>
<td>-29.69</td>
</tr>
<tr>
<td></td>
<td>--explicit-symbol</td>
<td>9.44</td>
</tr>
<tr>
<td></td>
<td>--no-slice</td>
<td>13.93</td>
</tr>
<tr>
<td></td>
<td>--undefined-retrav-nosym</td>
<td>6.10</td>
</tr>
<tr>
<td></td>
<td>--repeat-slicing 2</td>
<td>-7.12</td>
</tr>
<tr>
<td></td>
<td>--initial-heap-size 100</td>
<td>-47.65</td>
</tr>
<tr>
<td></td>
<td>--heap-mode bounded</td>
<td>-25.53</td>
</tr>
<tr>
<td></td>
<td>--heap-mode auto</td>
<td>23.74</td>
</tr>
<tr>
<td></td>
<td>--heap-limit 1</td>
<td>-44.90</td>
</tr>
<tr>
<td></td>
<td>--solver elderatica</td>
<td>-39.09</td>
</tr>
<tr>
<td></td>
<td>--heap-limit 10</td>
<td>-10.14</td>
</tr>
<tr>
<td></td>
<td>--initial-heap-size 10</td>
<td>24.37</td>
</tr>
<tr>
<td></td>
<td>--inline-size 100</td>
<td>-19.40</td>
</tr>
<tr>
<td></td>
<td>--bounded-heap-size 10</td>
<td>16.75</td>
</tr>
<tr>
<td></td>
<td>--step-heap-size 10</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>--path def</td>
<td>80.56</td>
</tr>
<tr>
<td></td>
<td>--localize-faults</td>
<td>-40.00</td>
</tr>
<tr>
<td></td>
<td>--paths fifo</td>
<td>-42.13</td>
</tr>
<tr>
<td></td>
<td>--java-threading</td>
<td>-15.07</td>
</tr>
<tr>
<td></td>
<td>--full-slice</td>
<td>-16.09</td>
</tr>
<tr>
<td></td>
<td>--slice-formula</td>
<td>12.17</td>
</tr>
<tr>
<td></td>
<td>--depth 100</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>--depth 1000</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>--synex-driven-lazy-load</td>
<td>28.93</td>
</tr>
</tbody>
</table>

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

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 option settings 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 statistically 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% 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 [24]. 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, $P_1$ (extracted from the program Float_div_true-unreach-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 15 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 32 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.,
ye 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’ esti-
mated effects (e.g., disabling --overflow-with-clang). However,
recall our answer to RQ1; doing so still cannot produce a one-
size-fits-all configuration that completes all the verification
tasks 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
static verification tools so they can successfully verify more
programs. As shown in Section II, the four tools under eval-
uation have very precise 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 Simulated Annealing 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 con-
figuration 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,
$f$. 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
current best configuration. The only way to know a configura-
tion’s true fitness would be to run the verification tool, observe
whether it completes, and validate its result. However, the vali-
dation 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 effectively approximates the fitness of a
tool configuration (such approximations are commonly known
as a surrogate fitness function in the literature [32]). Our model
needs to be trained once for each tool, but then incurs little
overhead when queried during search. The model steers the
search toward configurations that are likely to complete a
verification task with a correct result.

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 II-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

$^4$In our evaluation, we use the best-precision-config as the input configu-
ration (Section IV).
begins. We now discuss two key components of SATUNE: the meta-heuristic configuration search and the surrogate fitness function.

A. Meta-heuristic Configuration Search

There exist various meta-heuristic search algorithms in the literature, such as tabu search [33], hill climbing [34], genetic algorithms [35], and simulated annealing [36]. Among them, simulated annealing [36] has been shown to be more effective in finding fitter objects (like covering arrays [28], [37]–[39] and orthogonal arrays [40]) in large and complex combinatorial spaces quickly [25], [41], [42]. Considering the similarity in the search spaces of the mentioned successful applications, we decided to derive our meta-heuristic configuration search algorithm from simulated annealing [36], [43], [44].

At a high level, simulated annealing is a stochastic search algorithm [45] 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 ?? depicts our meta-heuristic configuration search algorithm that adapts simulated annealing. The inputs to this algorithm are: (1) the tool’s configuration space $CS = (O, D)$ 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 ??-?? perform initialization. First, we initialize the control parameters [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 ??, 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 ??, the program representation vector $V$ is initialized with the features extracted from $P$ (see Section ??B). At line ??, $f$ 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 ?? to ?? 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 ??, $T_s<T$ checks that the temperature $T$ has not decreased below the stopping temperature $T_s$ [36], 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 [46]), the algorithm first generates a new neighboring configuration $c'$ (line ??). 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 [28], [37], [42], [46]). The algorithm then computes the cost of $c'$ using $f$ (line ??), and reduces the running temperature by the cooling rate (line ??).

This random configuration generation repeats until one of the acceptance conditions on line [6] is met: either $\Delta E<0$, meaning $c'$ is better than $c$ according to $f$, or the algorithm decides to accept the inferior $c'$ with probability $e^{-\frac{\Delta E}{R}}$ [47]. This probability reduces with $T$. Once a state is accepted, $c$ and $E_c$ are updated (line ??). If $c$ is the best so far (i.e., $E_c<E_{c^*}$), then $c^*$ and $E_{c^*}$ are also updated accordingly (lines [7]??). 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 ??). Otherwise, the search continues to the next iteration.

Note that running the verification tool is the most expensive step of the Algorithm ???. In comparison, learning $f$, computing

5We 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.
program representations, generating random configurations, and computing their cost take negligible time.

B. Learning the Surrogate Fitness Function

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

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

Recall that Algorithm 2 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 [48]–[51]. In this work, we use a simple Bag of Words model [52] for its simplicity and relative efficacy in representing programs for classifying static analysis results [50]. We represent a program as a frequency vector by counting the frequencies of program instructions like load, store, allocate, and call; and certain constructs like branches, loops, functions, and primitive/array/pointer/compound types. This simplicity makes our approach somewhat 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 [53]–[55]. 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 46 LLVM IR instructions [56]. Java programs (targeted by JayHorn and JBMC) are represented by frequency vectors of 9 program features and 23 WALA IR instructions [57]. 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 for selecting a verification tool configuration, in terms of both correctness of the results and running time.

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 (using the \( f \) internally). We then rotated the partitions and repeated this process 25 times to perform 5-fold cross-validation [58] 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 trials as 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-configuration 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-configuration) 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-configuration. If it produces a conclusive result, we report it. Otherwise, we fall back to run the most-correct-configuration and report the result. We call this baseline precision→correct. Because the best-precision-configuration and most-correct-configuration are the same for Symbiotic and JBMC.
TABLE IV: Results for SATUNE and three baselines. The numbers in normal font are the 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.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Approach</th>
<th>Correct</th>
<th>Incorrect</th>
<th>Inconclusive</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBMC</td>
<td>SATUNE precision→random</td>
<td>804</td>
<td>196</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>precision→correct</td>
<td>738</td>
<td>195</td>
<td>8</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>most-correct-config</td>
<td>704</td>
<td>262</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>635</td>
<td>262</td>
<td>0</td>
<td>103</td>
</tr>
<tr>
<td>Symbiotic</td>
<td>SATUNE precision→random</td>
<td>277</td>
<td>1</td>
<td>0</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td>precision→correct</td>
<td>264</td>
<td>1</td>
<td>0</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>most-correct-config</td>
<td>261</td>
<td>1</td>
<td>0</td>
<td>403</td>
</tr>
<tr>
<td>JayHorn</td>
<td>SATUNE precision→random</td>
<td>240</td>
<td>13</td>
<td>2</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>precision→correct</td>
<td>211</td>
<td>14</td>
<td>2</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>most-correct-config</td>
<td>247</td>
<td>38</td>
<td>0</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>227</td>
<td>37</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>JBMC</td>
<td>SATUNE precision→random</td>
<td>348</td>
<td>7</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>precision→correct</td>
<td>343</td>
<td>6</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>most-correct-config</td>
<td>331</td>
<td>0</td>
<td>0</td>
<td>37</td>
</tr>
</tbody>
</table>

(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 by random alteration of the configuration, and this 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 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 than baselines?
- RQ4: How efficient is SATUNE?

RQ3 compares SATUNE to the three baselines in terms of the 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. This is attributable to SATUNE’s fitness function—specifically, because 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 (with 60 seconds timeout). 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 partially 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 combinations of configuration option settings, and past research in configurable software indicates that the majority of program behaviors are attributable to the interaction of few options [27]. Finally, there could be variance in the performance of SATUNE (and precision→random) caused by non-deterministic operations in machine learning and configuration selection. We partially 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 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 more broadly attempt to learn performance models of tool configuration spaces [59–68]. 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 researchers [11–15] have explored selecting a verification tool or SAT solver from a set 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]. Xu et al. developed SATZILLA2012 [14], which is the most recent version of the SATZILLA tool [15]. SATZILLA2012, given a SAT problem and a set of SAT solvers as input, attempts to select a solver that will perform the best in terms of run time. Our research complements these works by selecting a configuration of a single static analysis tool. The configuration spaces we select from are much larger than the sets of tools these works used, motivating the need for a meta-heuristic search strategy.

Finally, some software product line (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) [60]. Similarly, Ha and Zhang’s DeepPerf models a tool’s configuration space by training deep neural networks [63]. Nair et al.’s WHAT predicts a performance model using a small number of sampled configurations by performing dimensionality reduction on a target program’s configuration space [66]. FLASH, by Nair et al., builds a performance model using sequential model-based optimization, which allows the model to continue learning about the configuration space as it explores it and produces samples [67]. Finally, Nair et al., demonstrated that cheap and inaccurate predictors that rank configurations often perform as well or better than other more expensive approaches [68].

Other researchers have focused not on performance models, but rather on evaluating and contributing sampling approaches for SPLs. Pereira et al. evaluated six different configuration sampling approaches to determine their relative strengths and weaknesses for selecting representative configurations [64]. Oh et al. contributed a sampling approach which models feature spaces as counting binary decision diagrams, and then produces truly random samples of SPL configurations [65].

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, 6, 9, 16]. 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 [69]. 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 [70]. Jin and Sendhoff use ensembles of neural networks to improve the performance of evolutionary algorithms [71]. Singh et al. evaluated both regression models and radial basis functions as surrogate fitness functions in simulated annealing [72]. 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 distinctly 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 [73]. Ansel et al. present an extensible framework, OPENTUNER, that enables writing domain-specific HPC tuners [74]. For a more comprehensive review of the literature on HPC system tuning, we refer readers to a recent survey by Ashouri et al. [75]. 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 configurations 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 5. 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.

ACKNOWLEDGMENTS

This work is partly supported by NSF grants CCF-2007314, CCF-2008905 and CCF-2047682, and the NSF graduate research fellowship program.

REFERENCES


