Automatically Tailoring Abstract Interpretation to Custom Usage Scenarios

Muhammad Numair Mansur ✉[1], Benjamin Mariano[2], Maria Christakis[1], Jorge A. Navas[3], and Valentin Wüstholz[4]

[1] MPI-SWS, Kaiserslautern and Saarbrücken, Germany
{numair,maria}@mpi-sws.org
[2] The University of Texas at Austin, Austin, USA
bmariano@cs.utexas.edu
[3] SRI International, Menlo Park, USA
jorge.navas@sri.com
[4] ConsenSys, Kaiserslautern, Germany
valentin.wustholz@consensys.net

Abstract. In recent years, there has been significant progress in the development and industrial adoption of static analyzers, specifically of abstract interpreters. Such analyzers typically provide a large, if not huge, number of configurable options controlling the analysis precision and performance. A major hurdle in integrating them in the software-development life cycle is tuning their options to custom usage scenarios, such as a particular code base or certain resource constraints.
In this paper, we propose a technique that automatically tailors an abstract interpreter to the code under analysis and any given resource constraints. We implement this technique in a framework, tAIlor, which we use to perform an extensive evaluation on real-world benchmarks. Our experiments show that the configurations generated by tAIlor are vastly better than the default analysis options, vary significantly depending on the code under analysis, and most remain tailored to several subsequent code versions.

1 Introduction

Static analysis inspects code, without running it, in order to prove properties or detect bugs. Typically, static analysis approximates code behavior, for instance, because checking the correctness of most properties is undecidable. *Performance* is another important reason for this approximation. Typically, the closer the approximation is to the actual code behavior, the less efficient and the more *precise* the analysis is, that is, the fewer false positives it reports. For less tight approximations, the analysis tends to become more efficient but less precise.

Recent years have seen tremendous progress in the development and industrial adoption of static analyzers. Notable successes include Facebook's Infer [8,7] and AbsInt's Astrée [5]. Many popular analyzers, such as these, are based on *abstract interpretation* [12], a technique that abstracts the concrete program

semantics and reasons about its abstraction. In particular, program states are abstracted as elements of *abstract domains*. Most abstract interpreters offer a wide range of abstract domains that impact the precision and performance of the analysis. For instance, the Intervals domain [11] is typically faster but less precise than Polyhedra [16], which captures linear inequalities among variables.

In addition to the domains, abstract interpreters usually provide a large number of other options, for instance, whether backward analysis should be enabled or how quickly a fixpoint should be reached. In fact, the sheer number of option combinations (over 6M in our experiments) is bound to overwhelm users, especially non-expert ones. To make matters worse, the best option combinations may vary significantly depending on the code under analysis and the resources, such as time or memory, that users are willing to spend.

In light of this, we suspect that most users resort to using the default options that the analysis designer pre-selected for them. However, these are definitely not suitable for all code. Moreover, they do not adjust to different stages of software development, e.g., running the analysis in the editor should be much faster than running it in a continuous integration (CI) pipeline, which in turn should be much faster than running it prior to a major release. The alternative of enabling the (in theory) most precise analysis can be even worse, since in practice it often runs out of time or memory as we show in our experiments. As a result, the widespread adoption of abstract interpreters is severely hindered, which is unfortunate since they constitute an important class of practical analyzers.

Our approach. To address this issue, we present the first technique that automatically tailors a generic abstract interpreter to a custom usage scenario. With the term *custom usage scenario*, we refer to a particular piece of code and specific resource constraints. The key idea behind our technique is to phrase the problem of customizing the abstract-interpretation configuration to a given usage scenario as an optimization problem. Specifically, different configurations are compared using a cost function that penalizes those that prove fewer properties or require more resources. The cost function can guide the configuration search of a wide range of existing optimization algorithms. This problem of tuning abstract interpreters can be seen as an instance of the more general problem of *algorithm configuration* [31]. In the past, algorithm configuration has been used to tune algorithms for solving various hard problems, such as SAT solving [33,32], and more recently, training of machine-learning models [3,52,18].

We implement our technique in an open-source framework called tAIlor[5], which configures a given abstract interpreter for a given usage scenario using a given optimization algorithm. As a result, tAIlor enables the abstract interpreter to prove as many properties as possible within the resource limit without requiring any domain expertise on behalf of the user.

Using tAIlor, we find that tailored configurations vastly outperform the default options pre-selected by the analysis designers. In fact, we show that this is possible even with very simple optimization algorithms. Our experiments

[5]The tool implementation is found at `https://github.com/Practical-Formal-Methods/tailor` and an installation at `https://doi.org/10.5281/zenodo.4719604`.

also demonstrate that tailored configurations vary significantly depending on the usage scenario—in other words, there cannot be a single configuration that fits all scenarios. Finally, most of the generated configurations remain tailored to several subsequent code versions, suggesting that re-tuning is only necessary after major code changes.

Contributions. We make the following contributions:

1. We present the first technique for automatically tailoring abstract interpreters to custom usage scenarios.
2. We implement our technique in an open-source framework called tAIlor.
3. Using a state-of-the-art abstract interpreter, Crab [25], with millions of configurations, we show the effectiveness of tAIlor on real-world benchmarks.

2 Overview

We now illustrate the workflow and tool architecture of tAIlor and provide examples of its effectiveness.

Terminology. In the following, we refer to an abstract domain with all its options (e.g., enabling backward analysis or more precise treatment of arrays etc.) as an *ingredient*.

As discussed earlier, abstract interpreters typically provide a large number of such ingredients. To make matters worse, it is also possible to combine different ingredients into a sequence (which we call a *recipe*) such that more properties are verified than with individual ingredients. For example, a user could configure the abstract interpreter to first use Intervals to verify as many properties as possible and then use Polyhedra to attempt verification of any remaining properties. Of course, the number of possible configurations grows exponentially in the length of the recipe (over 6M in our experiments for recipes up to length 3).

Workflow. The high-level architecture of tAIlor is shown in Fig. 1. It takes as input the code to be analyzed (i.e., any program, file, function, or fragment), a user-provided resource limit, and optionally an optimization algorithm. We focus on time as the constrained resource in this paper, but our technique could be easily extended to other resources, such as memory.

The optimization engine relies on a recipe generator to generate a fresh recipe. To assess its quality in terms of precision and performance, the recipe evaluator computes a cost for the recipe. The cost is computed by evaluating how precise and efficient the abstract interpreter is for the given recipe. This cost is used by the optimization engine to keep track of the best recipe so far, i.e., the one that proves the most properties in the least amount of time. tAIlor repeats this process for a given number of iterations to sample multiple recipes and returns the recipe with the lowest cost.

Zooming in on the evaluator, a recipe is processed by invoking the abstract interpreter for each ingredient. After each analysis (i.e., one ingredient), the evaluator collects the new verification results, that is, the verified assertions. All verification results that have been achieved so far are subsequently shared with the analyzer when it is invoked for the next ingredient. Verification results are

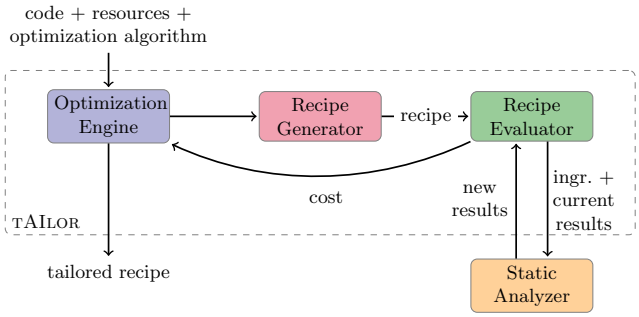

Figure 1: Overview of our framework.

shared by converting all verified assertions into assumptions. After processing the entire recipe, the evaluator computes a cost for the recipe, which depends on the number of unverified assertions and the total analysis time.

In general, there might be more than one recipe tailored to a particular usage scenario. Naïvely, finding one requires searching the space of all recipes. Sect. 4.3 discusses several optimization algorithms for performing this search, which tAIlor already incorporates in its optimization engine.

Examples. As an example, let us consider the usage scenario where a user runs the Crab abstract interpreter [25] in their editor for instant feedback during code development. This means that the allowed time limit for the analysis is very short, say, 1 sec. Now assume that the code under analysis is a program file[6] of the multimedia processing tool ffmpeg, which is used to evaluate the effectiveness of tAIlor in our experiments. In this file, Crab checks 45 assertions for common bugs, i.e., division by zero, integer overflow, buffer overflow, and use after free.

Analysis of this file with the default Crab configuration takes 0.35 sec to complete. In this time, Crab proves 17 assertions and emits 28 warnings about the properties that remain unverified. For this usage scenario, tAIlor is able to tune the abstract-interpreter configuration such that the analysis time is 0.57 sec and the number of verified properties increases by 29% (i.e., 22 assertions are proved). Note that the tailored configuration uses a completely different abstract domain than the one in the default configuration. As a result, the verification results are significantly better, but the analysis takes slightly longer to complete (although remaining within the specified time limit). In contrast, enabling the most precise analysis in Crab verifies 26 assertions but takes over 6 min to complete, which by far exceeds the time limit imposed by the usage scenario.

While it takes tAIlor 4.5 sec to find the above configuration, this is time well invested; the configuration can be re-used for several subsequent code versions. In fact, in our experiments, we show that generated configurations can remain tailored for at least up to 50 subsequent commits to a file under version control. Given that changes in the editor are typically much more incremental, we expect that no re-tuning would be necessary at all during an editor session. Re-tuning

[6] https://github.com/FFmpeg/FFmpeg/blob/master/libavformat/idcin.c

may be beneficial after major changes to the code under analysis and can happen offline, e.g., between editor sessions, or in the worst case overnight.

As another example, consider the usage scenario where Crab is integrated in a CI pipeline. In this scenario, users should be able to spare more time for analysis, say, 5 min. Here, let us assume that the analyzed code is a program file[7] of the curl tool for transferring data by URL, which is also used in our evaluation. The default Crab configuration takes 0.23 sec to run and only verifies 2 out of 33 checked assertions. tAIlor is able to find a configuration that takes 7.6 sec and proves 8 assertions. In contrast, the most precise configuration does not terminate even after 15 min.

Both scenarios demonstrate that, even when users have more time to spare, the default configuration cannot take advantage of it to improve the verification results. At the same time, the most precise configuration is completely impractical since it does not respect the resource constraints imposed by these scenarios.

3 Background: A Generic Abstract Interpreter

Many successful abstract interpreters (e.g., Astrée [5], C Global Surveyor [53], Clousot [17], Crab [25], IKOS [6], Sparrow [46], and Infer [8]) follow the generic architecture in Fig. 2. In this section, we describe its main components to show that our approach should generalize to such analyzers.

Memory domain. Analysis of low-level languages such as C and LLVM-bitcode requires reasoning about pointers. It is, therefore, common to design a *memory domain* [42] that can simultaneously reason about pointer aliasing, memory contents, and numerical relations between them.

Pointer domains resolve aliasing between pointers, and *array domains* reason about memory contents. More specifically, array domains can reason about individual memory locations (cells), infer universal properties over multiple cells, or both. Typically, reasoning about individual cells trades performance for precision unless there are very few array elements (e.g., [22,42]). In contrast, reasoning about multiple memory locations (*summarized cells*) trades precision for performance. In our evaluation, we use *Array smashing* domains [5] that abstract different array elements into a single summarized cell. *Logico-numerical domains* infer relationships between program and *synthetic* variables, introduced by the pointer and array domains, e.g., summarized cells.

Next, we introduce domains typically used for proving the absence of runtime errors in low-level languages. *Boolean domains* (e.g., flat Boolean, BDDApron [1]) reason about Boolean variables and expressions. *Non-relational domains* (e.g., Intervals [11], Congruence [23]) do not track relations among different variables, in contrast to *relational domains* (e.g., Equality [35], Zones [41], Octagons [43], Polyhedra [16]). Due to their increased precision, relational domains are typically less efficient than non-relational ones. *Symbolic domains* (e.g., Congruence closure [9], Symbolic constant [44], Term [21]) abstract complex expressions (e.g., non-linear) and external library calls by uninterpreted functions.

[7] `https://github.com/curl/curl/blob/master/lib/cookie.c`

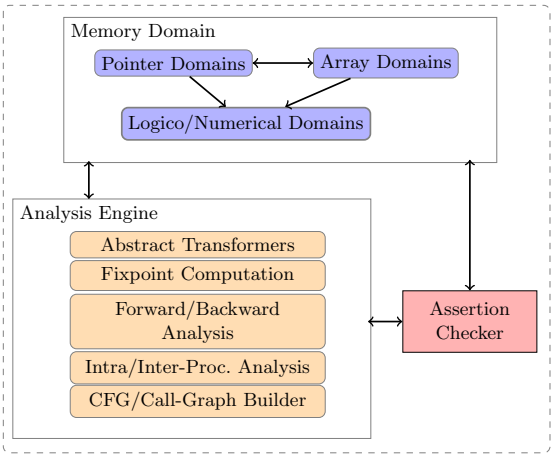

Figure 2: Generic architecture of an abstract interpreter.

Non-convex domains express disjunctive invariants. For instance, the DisInt domain [17] extends Intervals to a finite disjunction; it retains the scalability of the Intervals domain by keeping only non-overlapping intervals. On the other hand, the Boxes domain [24] captures arbitrary Boolean combinations of intervals, which can often be expensive.

Fixpoint computation. To ensure termination of the fixpoint computation, Cousot and Cousot introduce *widening* [12,14], which usually incurs a loss of precision. There are three common strategies to reduce this precision loss, which however sacrifice efficiency. First, *delayed widening* [5] performs a number of initial fixpoint-computation iterations in the hope of reaching a fixpoint before resorting to widening. Second, *widening with thresholds* [37,40] limits the number of program expressions (thresholds) that are used when widening. The third strategy consists in applying *narrowing* [12,14] a certain number of times.

Forward and backward analysis. Classically, abstract interpreters analyze code by propagating abstract states in a *forward* manner. However, abstract interpreters can also perform *backward* analysis to compute the execution states that lead to an assertion violation. Cousot and Cousot [13,15] define a *forward-backward refinement* algorithm in which a forward analysis is followed by a backward analysis until no more refinement is possible. The backward analysis uses invariants computed by the forward analysis, while the forward analysis does not explore states that cannot reach an assertion violation based on the backward analysis. This refinement is more precise than forward analysis alone, but it may also become very expensive.

Intra- and inter-procedural analysis. An *intra-procedural* analysis analyzes a function ignoring the information (i.e., call stack) that flows into it, while an *inter-procedural* analysis considers all flows among functions. The former is much more efficient and easy to parallelize, but the latter is usually more precise.

Algorithm 1: Optimization engine.

```
1  Function Optimize(P, r_max, l_max, i_dom, i_set, rec_init, GenerateRecipe,
     Accept) is
2      // Phase 1 (optimize domains)
3      rec_best := rec_curr := rec_init
4      cost_best := cost_curr := Evaluate(P, r_max, rec_best)
5      for l := 1 to l_max do
6          for i := 1 to i_dom · l do
7              rec_next := GenerateRecipe(rec_curr, l)
8              cost_next := Evaluate(P, r_max, rec_next)
9              if cost_next < cost_best then
10                 rec_best, cost_best := rec_next, cost_next
11             if Accept(cost_curr, cost_next) then
12                 rec_curr, cost_curr := rec_next, cost_next
13     // Phase 2 (optimize settings)
14     for i := 1 to i_set do
15         rec_mut := MutateSettings(rec_best)
16         cost_mut := Evaluate(P, r_max, rec_mut)
17         if cost_mut < cost_best then
18             rec_best, cost_best := rec_mut, cost_mut
19     return rec_best
```

4 Our Technique

This section describes the components of tAIlor in detail; Sects. 4.1, 4.2, 4.3 explain the optimization engine, recipe evaluator, and recipe generator (Fig. 1).

4.1 Recipe Optimization

Alg. 1 implements the optimization engine. In addition to the code P and the resource limit r_{max}, it also takes as input the maximum length of the generated recipes l_{max} (i.e., the maximum number of ingredients), a function to generate new recipes GenerateRecipe (i.e., the recipe generator from Fig. 1), and four other parameters, which we explain later.

A tailored recipe is found in two phases. The first phase aims to find the best abstract domain for each ingredient, while the second tunes the remaining analysis settings for each ingredient (e.g., whether backward analysis should be enabled). Parameters i_{dom} and i_{set} control the number of iterations of each phase. Note that we start with a search for the best domains since they have the largest impact on the precision and performance of the analysis.

During the first phase, the algorithm initializes the best recipe rec_{best} with an initial recipe rec_{init} (line 3). The cost of this recipe is evaluated with function Evaluate, which implements the recipe evaluator from Fig. 1. The subsequent

nested loop (line 5) samples a number of recipes, starting with the shortest recipes ($l := 1$) and ending with the longest recipes ($l := l_{max}$). The inner loop generates i_{dom} ingredients for each ingredient in the recipe (i.e., $i_{dom} \cdot l$ total iterations) by invoking function GenerateRecipe, and in case a recipe with lower cost is found, it updates the best recipe (lines 9–10). Several optimization algorithms, such as hill climbing and simulated annealing, search for an optimal result by mutating some of the intermediate results. Variable rec_{curr} stores intermediate recipes to be mutated, and function Accept decides when to update it (lines 11–12).

As explained earlier, the purpose of the first phase is to identify the best sequence of abstract domains. The second phase (lines 13–18) focuses on tuning the other settings of the best recipe so far. This is done by randomly mutating the best recipe via MutateSettings (line 15), and updating the best recipe if better settings are found (lines 17–18). After exploring i_{set} random settings, the best recipe is returned to the user (line 19).

4.2 Recipe Evaluation

The recipe evaluator from Fig. 1 uses a cost function to determine the quality of a fresh recipe with respect to the precision and performance of the abstract interpreter. This design is motivated by the fact that analysis imprecision and inefficiency are among the top pain points for users [10].

Therefore, the cost function depends on the number of generated warnings w (that is, the number of unverified assertions), the total number of assertions in the code w_{total}, the resource consumption r of the analyzer, and the resource limit r_{max} imposed on the analyzer:

$$cost(w, w_{total}, r, r_{max}) = \begin{cases} w + \dfrac{r}{r_{max}} \\ \dfrac{}{w_{total}}, & \text{if } r \leq r_{max} \\ \infty, & \text{otherwise} \end{cases}$$

Note that w and r are measured by invoking the abstract interpreter with the recipe under evaluation. The cost function evaluates to a lower cost for recipes that improve the precision of the abstract interpreter (due to the term w/w_{total}). In case of ties, the term r/r_{max} causes the function to evaluate to a lower cost for recipes that result in a more efficient analysis. In other words, for two recipes resulting in equal precision, the one with the smaller resource consumption is assigned a lower cost. When a recipe causes the analyzer to exceed the resource limit, it is assigned infinite cost.

4.3 Recipe Generation

In the literature, there is a broad range of optimization algorithms for different application domains. To demonstrate the generality and effectiveness of tAIlor, we instantiate it with four adaptations of three well-known optimization algorithms, namely random sampling [38], hill climbing (with regular restarts) [48],

and simulated annealing [39,36]. Here, we describe these algorithms in detail, and in Sect. 5, we evaluate their effectiveness.

Before diving into the details, let us discuss the suitability of different kinds of optimization algorithms for our domain. There are algorithms that leverage mathematical properties of the function to be optimized, e.g., by computing derivatives as in Newton's iterative method. Our cost function, however, is evaluated by running an abstract interpreter, and thus, it is not differentiable or continuous. This constraint makes such analytical algorithms unsuitable. Moreover, evaluating our cost function is expensive, especially for precise abstract domains such as Polyhedra. This makes algorithms that require a large number of samples, such as genetic algorithms, less practical.

Now recall that Alg. 1 is parametric in how new recipes are generated (with GenerateRecipe) and accepted for further mutations (with Accept). Instantiations of these functions essentially constitute our search strategy for a tailored recipe. In the following, we discuss four such instantiations. Note that, in theory, the order of recipe ingredients matters. This is because any properties verified by one ingredient are converted into assumptions for the next, and different assumptions may lead to different verification results. Therefore, all our instantiations are able to explore different ingredient orderings.

Random sampling. Random sampling (rs) just generates random recipes of a certain length. Function Accept always returns *false* as each recipe is generated from scratch, and not as a result of any mutations.

Domain-aware random sampling. rs might generate recipes containing abstract domains of comparable precision. For instance, the Octagons domain is typically strictly more precise than Intervals. Thus, a recipe consisting of these domains is essentially equivalent to one containing only Octagons.

Now, assume that we have a partially ordered set (poset) of domains that defines their ordering in terms of precision. An example of such a poset for a particular abstract interpreter is shown in Fig. 3. An optimization algorithm can then leverage this information to reduce the search space of possible recipes. Given such a poset, we therefore define domain-aware random sampling (dars), which randomly samples recipes that do not contain abstract domains of comparable precision. Again, Accept always returns *false*.

Simulated annealing. Simulated annealing (sa) searches for the best recipe by mutating the current recipe rec_{curr} in Alg. 1. The resulting recipe (rec_{next}), if accepted on line 12, becomes the new recipe to be mutated. Alg. 2 shows an instantiation of GenerateRecipe, which mutates a given recipe such that the poset precision constraints are satisfied (i.e., there are no domains of comparable precision). A recipe is mutated either by adding new ingredients with 20% probability or by modifying existing ones with 80% probability (line 2). The probability of adding ingredients is lower to keep recipes short.

When adding a new ingredient (lines 4–5), Alg. 2 calls RandomPosetLeastIncomparable, which considers all domains that are incomparable with the domains in the recipe. Given this set, it randomly selects from the domains with the least precision to avoid adding overly expensive domains. When modifying

Algorithm 2: A recipe-generator instantiation.

```
1  Function GenerateRecipe(rec, l_max) is
2      act := RandomAction({ADD: 0.2, MOD: 0.8}))
3      if act = ADD ∧ Len(rec) < l_max then
4          ingr_new := RandomPosetLeastIncomparable(rec)
5          rec_mut := AddIngredient(rec, ingr_new)
6      else
7          ingr := RandomIngredient(rec)
8          act_m := RandomAction({GT: 0.5, LT: 0.3, INC: 0.2})
9          if act_m = GT then
10             ingr_new := PosetGreaterThan(ingr)
11         else if act_m = LT then
12             ingr_new := PosetLessThan(ingr)
13         else
14             rec_rem := RemoveIngredient(rec, ingr)
15             ingr_new := RandomPosetLeastIncomparable(rec_rem)
16         rec_mut := ReplaceIngredient(rec, ingr, ingr_new)
17     if ¬PosetCompatible(rec_mut) then
18         rec_mut := GenerateRecipe(rec, l_max)
19     return rec_mut
```

a random ingredient in the recipe (lines 7–16), the algorithm can replace its domain with one of three possibilities: a domain that is immediately more precise (i.e., not transitively) in the poset (via PosetGreaterThan), a domain that is immediately less precise (via PosetLessThan), or an incomparable domain with the least precision (via RandomPosetLeastIncomparable). If the resulting recipe does not satisfy the poset precision constraints, our algorithm retries to mutate the original recipe (lines 17–18).

For simulated annealing, Accept returns *true* if the new cost (for the mutated recipe) is less than the current cost. It also accepts recipes whose cost is higher with a certain probability, which is inversely proportional to the cost increase and the number of explored recipes. That is, recipes with a small cost increase are likely to be accepted, especially at the beginning of the exploration.

Hill climbing. Our instantiation of hill climbing (HC) performs regular restarts. In particular, it starts with a randomly generated recipe that satisfies the poset precision constraints, generates 10 new valid recipes, and restarts with a random recipe. Accept returns *true* only if the new cost is lower than the best cost, which is equivalent to the current cost.

5 Experimental Evaluation

To evaluate our technique, we aim to answer the following research questions:

RQ1: Is our technique effective in tailoring recipes to different usage scenarios?

RQ2: Are the tailored recipes optimal?
RQ3: How diverse are the tailored recipes?
RQ4: How resilient are the tailored recipes to code changes?

5.1 Implementation

We implemented tAIlor by extending Crab [25], a parametric framework for modular construction of abstract interpreters[8]. We extended Crab with the ability to pass verification results between recipe ingredients as well as with the four optimization algorithms discussed in Sect. 4.3.

Tab. 1 shows all settings and values used in our evaluation. The first three settings refer to the strategies discussed in Sect. 3 for mitigating the precision loss incurred by widening. For the initial recipe, tAIlor uses Intervals and the Crab default values for all other settings (in bold in the table). To make the search more efficient, we selected a representative subset of all possible setting values.

Crab uses a DSA-based [26] pointer analysis and can, optionally, reason about array contents using array smashing. It offers a wide range of logico-numerical domains, shown in Fig. 3. The `bool` domain is the flat Boolean domain, `ric` is a reduced product of Intervals and Congruence, and `term(int)` and `term(disInt)` are instantiations of the Term domain with `intervals` and `disInt`, respectively. Although Crab provides a bottom-up inter-procedural analysis, we use the default intra-procedural analysis; in fact, most analyses deployed in real usage scenarios are intra-procedural due to time constraints [10].

5.2 Benchmark Selection

For our evaluation, we systematically selected popular and (at some point) active C projects on GitHub. In particular, we chose the six most starred C repositories with over 300 commits that we could successfully build with the Clang-5.0 compiler. We give a short description of each project in Tab. 2.

For analyzing these projects, we needed to introduce properties to be verified. We, thus, automatically instrumented these projects with four types of assertions

Table 1: Crab settings and their possible values as used in our experiments. Default settings are shown in bold.

Setting	Possible Values
NUM_DELAY_WIDEN	$\{\mathbf{1}, 2, 4, 8, 16\}$
NUM_NARROW_ITERATIONS	$\{1, \mathbf{2}, 3, 4\}$
NUM_WIDEN_THRESHOLDS	$\{\mathbf{0}, 10, 20, 30, 40\}$
Backward Analysis	$\{\mathbf{OFF}, \mathit{ON}\}$
Array Smashing	$\{\mathit{OFF}, \mathbf{ON}\}$
Abstract Domains	all domains in Fig. 3

[8] Crab is available at https://github.com/seahorn/crab.

Table 2: Overview of projects.

Project	Description
CURL	Tool for transferring data by URL
DARKNET	Convolutional neural-network framework
FFMPEG	Multimedia processing tool
GIT	Distributed version-control tool
PHP-SRC	PHP interpreter
REDIS	Persistent in-memory database

that check for common bugs; namely, division by zero, integer overflow, buffer overflow, and use after free. Introducing assertions to check for runtime errors such as these is common practice in program analysis and verification.

As projects consist of different numbers of files, to avoid skewing the results in favor of a particular project, we randomly and uniformly sampled 20 LLVM-bitcode files from each project, for a total of 120. To ensure that each file was neither too trivial nor too difficult for the abstract interpreter, we used the number of assertions as a complexity indicator and only sampled files with at least 20 assertions and at most 100. Additionally, to guarantee all four assertion types were included and avoid skewing the results in favor of a particular assertion type, we required that the sum of assertions for each type was at least 70 across all files—this exact number was largely determined by the benchmarks.

Overall, our benchmark suite of 120 files totals 1346 functions, 5557 assertions (on average 4 assertions per function), and 667927 LLVM instructions (Tab. 3).

5.3 Results

We now present our experimental results for each research question. We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30GHz machine with 264GB of memory, running Ubuntu 16.04.1 LTS.

RQ1: Is our technique effective in tailoring recipes to different usage scenarios? We instantiated TAILOR with the four optimization algorithms

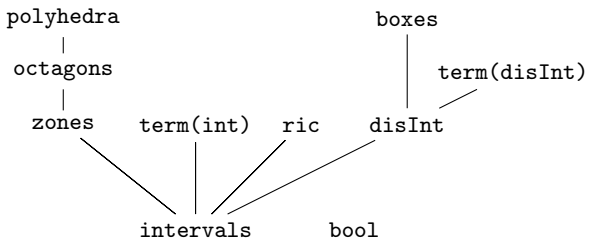

Figure 3: Comparing logico-numerical domains in Crab. A domain d_1 is less precise than d_2 if there is a path from d_1 to d_2 going upward, otherwise d_1 and d_2 are incomparable.

Table 3: Benchmark characteristics (20 files per project). The last three columns show the number of functions, assertions, and LLVM instructions in the analyzed files.

Project	Functions	Assertions	LLVM Instructions
CURL	306	787	50 541
DARKNET	130	958	55 847
FFMPEG	103	888	27 653
GIT	218	768	102 304
PHP-SRC	268	1031	305 943
REDIS	321	1125	125 639
Total	1346	5557	667 927

described in Sect. 4.3: RS, DARS, SA, and HC. We constrained the analysis time to simulate two usage scenarios: 1 sec for instant feedback in the editor, and 5 min for feedback in a CI pipeline. We compare TAILOR with the default recipe (DEF), i.e., the default settings in CRAB as defined by its designer after careful tuning on a large set of benchmarks over the years. DEF uses a combination of two domains, namely, the reduced product of Boolean and Zones. The other default settings are in Tab. 1.

For this experiment, we ran TAILOR with each optimization algorithm on the 120 benchmark files, enabling optimization at the granularity of files. Each algorithm was seeded with the same random seed. In Alg. 1, we restrict recipes to contain at most 3 domains ($l_{max} = 3$) and set the number of iterations for each phase to be 5 and 10 ($i_{dom} = 5$ and $i_{set} = 10$).

The results are presented in Fig. 4, which shows the number of assertions that are verified with the best recipe found by each algorithm as well as by the default recipe. All algorithms outperform the default recipe for both usage scenarios, verifying almost twice as many assertions on average. The random-sampling algorithms are shown to find better recipes than the others, with DARS being the most effective. Hill climbing is less effective since it gets stuck in local

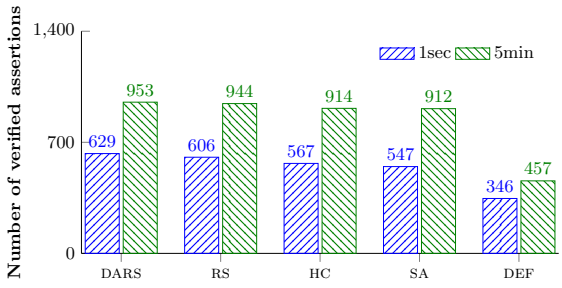

Figure 4: Comparison of the number of assertions verified with the best recipe generated by each optimization algorithm and with the default recipe, for varying timeouts.

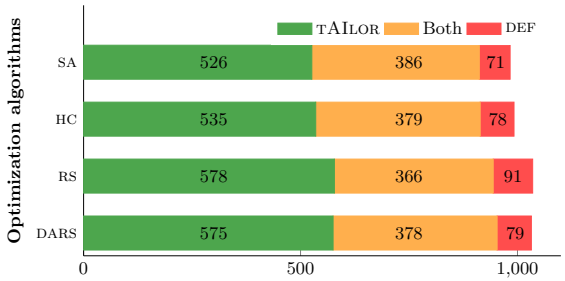

Figure 5: Comparison of the number of assertions verified by a tailored vs. the default recipe.

cost minima despite restarts. Simulated annealing is the least effective because it slowly climbs up the poset toward more precise domains (see Alg. 2). However, as we explain later, we expect the algorithms to converge on the number of verified assertions for more iterations.

Fig. 5 gives a more detailed comparison with the default recipe for the time limit of 5 min. In particular, each horizontal bar shows the total number of assertions verified by each algorithm. The orange portion represents the assertions verified by both the default recipe and the optimization algorithm, while the green and red portions represent the assertions only verified by the algorithm and default recipe, respectively. These results show that, in addition to verifying hundreds of new assertions, TAILOR is able to verify the vast majority of assertions proved by the default recipe, regardless of optimization algorithm.

In Fig. 6, we show the total time each algorithm takes for all iterations. DARS takes the longest. This is due to generating more precise recipes thanks to its domain knowledge. Such recipes typically take longer to run but verify more assertions (as in Fig. 4). On average, for all algorithms, TAILOR requires only 30 sec to complete all iterations for the 1-sec timeout and 16 min for the 5-min timeout. As discussed in Sect. 2, this tuning time can be spent offline.

Fig. 7 compares the total number of assertions verified by each algorithm when TAILOR runs for 40 ($i_{dom} = 5$ and $i_{set} = 10$) and 80 ($i_{dom} = 10$ and $i_{set} =$

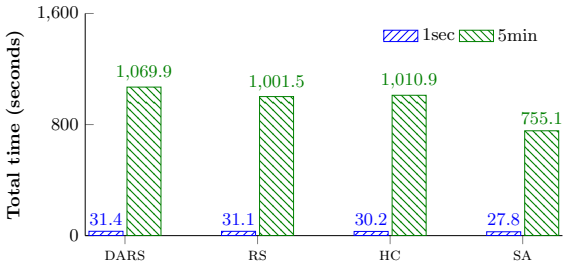

Figure 6: Comparison of the total time (in sec) that each algorithm requires for all iterations, for varying timeouts.

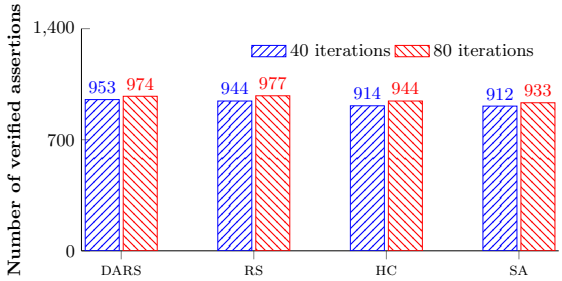

Figure 7: Comparison of the number of assertions verified with the best recipe generated by the different optimization algorithms, for different numbers of iterations.

20) iterations. The results show that only a relatively small number of additional assertions are verified with 80 iterations. In fact, we expect the algorithms to eventually converge on the number of verified assertions, given the time limit and precision of the available domains.

As DARS performs best in this comparison, we only evaluate DARS in the remaining research questions. We use a 5-min timeout.

> **RQ1 takeaway:** TAILOR verifies between $1.6 - 2.1\times$ the assertions of the default recipe, regardless of optimization algorithm, timeout, or number of iterations. In fact, even very simple algorithms (such as RS) significantly outperform the default recipe.

RQ2: Are the tailored recipes optimal? To check the optimality of the tailored recipes, we compared them with the most precise (and least efficient) CRAB configuration. It uses the most precise domains from Fig. 3 (i.e., `bool`, `polyhedra`, `term(int)`, `ric`, `boxes`, and `term(disInt)`) in a recipe of 6 ingredients and assigns the most precise values to all other settings from Tab. 1. We generously gave a 30-min timeout to this recipe.

For 21 out of 120 files, the most precise recipe ran out of memory (264GB). For 86 files, it terminated within 5 min, and for 13, it took longer (within 30 min)—in many cases, this was even longer than TAILOR's tuning time in Fig. 6. We compared the number of assertions verified by our tailored recipes (which do not exceed 5 min) and by the most precise recipe. For the 86 files that terminated within 5 min, our recipes prove 618 assertions, whereas the most precise recipe proves 534. For the other 13 files, our recipes prove 119 assertions, whereas the most precise recipe proves 98.

Consequently, our (in theory) less precise and more efficient recipes prove more assertions in files where the most precise recipe terminates. Possible explanations for this non-intuitive result are: (1) Polyhedra coefficients may overflow, in which case the constraints are typically ignored by abstract interpreters, and (2) more precise domains with different widening operations may result in less precise results [45,2].

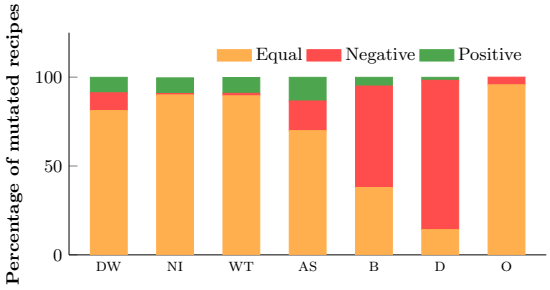

Figure 8: Effect of different settings on the precision and performance of the abstract interpreter. (dw: `NUM_DELAY_WIDEN`, ni: `NUM_NARROW_ITERATIONS`, wt: `NUM_WIDEN_THRESHOLDS`, as: array smashing, b: backward analysis, d: abstract domain, o: ingredient ordering).

We also evaluated the optimality of tailored recipes by mutating individual parts of the recipe and comparing to the original. In particular, for each setting in Tab. 1, we tried all possible values and replaced each domain with all other comparable domains in the poset of Fig. 3. For example, for a recipe including `zones`, we tried `octagons`, `polyhedra`, and `intervals`. In addition, we tried all possible orderings of the recipe ingredients, which in theory could produce different results. We observed whether these changes resulted in a difference in the precision and performance of the analyzer.

Fig. 8 shows the results of this experiment, broken down by setting. Equal (in orange) indicates that the mutated recipe proves the same number of assertions within ±5 seconds of the original. Positive (in green) indicates that it either proves more assertions or the same number of assertions at least 5 seconds faster. Negative (in red) indicates that the mutated recipe either proves fewer assertions or the same number of assertions at least 5 seconds slower.

The results show that, for our benchmarks, mutating the recipe found by tAIlor rarely led to an improvement. In particular, at least 93% of all mutated recipes were either equal to or worse than the original recipe. In the majority of these cases, mutated recipes are equally good. This indicates that there are many optimal or close-to-optimal solutions and that tAIlor is able to find one.

> **RQ2 takeaway:** As compared to the most precise recipe, tAIlor verified more assertions across benchmarks where the most precise recipe terminated. Furthermore, mutating recipes found by tAIlor resulted in improvement only for less than 7% of recipes.

RQ3: How diverse are the tailored recipes? To motivate the need for optimization, we must show that tailored recipes are sufficiently diverse such that they could not be replaced by a well-crafted default recipe. To better understand the characteristics of tailored recipes, we manually inspected all of them.

tAIlor generated recipes of length greater than 1 for 61 files. Out of these, 37 are of length 2 and 24 of length 3. For 77% of generated recipes, `NUM_DELAY_`-

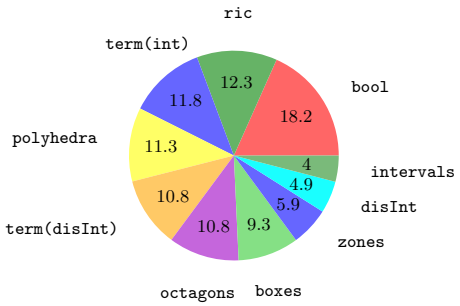

Figure 9: Occurrence of domains (in %) in the best recipes for all assertion types.

`WIDEN` is not set to the default value of 1. Additionally, 55% of the ingredients enable array smashing, and 32% enable backward analysis.

Fig. 9 shows how often (in percentage) each abstract domain occurs in a best recipe found by tAIlor. We observe that all domains occur almost equally often, with 6 of the 10 domains occurring in between 9% and 13% of recipes. The most common domain was `bool` at 18%, and the least common was `intervals` at 4%. We observed a similar distribution of domains even when instrumenting the benchmarks with only one assertion type, e.g., checking for integer overflow.

We also inspected which domain combinations are frequently used in the tailored recipes. One common pattern is combinations between `bool` and numerical domains (18 occurrences). Similarly, we observed 2 occurrences of `term(disInt)` together with `zones`. Interestingly, the less powerful variants of combining `disInt` with `zones` (3 occurrences) and `term(int)` with `zones` (6 occurrences) seem to be sufficient in many cases. Finally, we observed 8 occurrences of `polyhedra` or `octagons` with `boxes`, which are the most precise convex and non-convex domains. Our approach is, thus, not only useful for users, but also for designers of abstract interpreters by potentially inspiring new domain combinations.

> **RQ3 takeaway:** The diversity of tailored recipes prevents replacing them with a single default recipe. Over half of the tailored recipes contain more than one ingredient, and ingredients use a variety of domains and their settings.

RQ4: How resilient are the tailored recipes to code changes? We expect tailored recipes to be resilient to code changes, i.e., to retain their optimality across several changes without requiring re-tuning. We now evaluate if a recipe tailored for one code version is also tailored for another, even when the two versions are 50 commits apart.

For this experiment, we took a random sample of 60 files from our benchmarks and retrieved the 50 most recent commits per file. We only sampled 60 out of 120 files as building these files for each commit is quite time consuming—it can take up to a couple of days. We instrumented each file version with the four

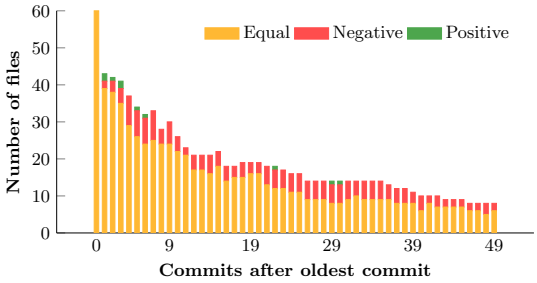

Figure 10: Difference in the safe assertions across commits.

assertion types described in Sect. 5.2. It should be noted that, for some files, we retrieved fewer than 50 versions either because there were fewer than 50 total commits or our build procedure for the project failed on older commits. This is also why we did not run this experiment for over 50 commits.

We analyzed each file version with the best recipe, R_o, found by tAIlor for the oldest file version. We compared this recipe with new best recipes, R_n, that were generated by tAIlor when run on each subsequent file version. For this experiment, we used a 5-min timeout and 40 iterations.

Note that, when running tAIlor with the same optimization algorithm and random seed, it explores the same recipes. It is, therefore, very likely that recipe R_o for the oldest commit is also the best for other file versions since we only explore 40 different recipes. To avoid any such bias, we performed this experiment by seeding tAIlor with a different random seed for each commit. The results are shown in Fig. 10.

In Fig. 10, we give a bar chart comparing the number of files per commit that have a positive, equal, and negative difference in the number of verified assertions, where commit 0 is the oldest commit and 49 the newest. An equal difference (in orange) means that recipe R_o for the oldest commit proves the same number of assertions in the current file version, f_n, as recipe R_n found by running tAIlor on f_n. To be more precise, we consider the two recipes to be equal if they differ by at most 1 verified assertion or 1% of verified assertions since such a small change in the number of safe assertions seems acceptable in practice (especially given that the total number of assertions may change across commits). A positive difference (in green) means that R_o achieves better verification results than R_n, that is, R_o proves more assertions safe (over 1 assertion or 1% of the assertions that R_n proves). Analogously, a negative difference (in red) means that R_o proves fewer assertions. We do not consider time here because none of the recipes timed out when applied on any file version.

Note that the number of files decreases for newer commits. This is because not all files go forward by 50 commits, and even if they do, not all file versions build. However, in a few instances, the number of files increases going forward in time. This happens for files that change names, and later, change back, which we do not account for.

For the vast majority of files, using recipe R_o (found for the oldest commit) is as effective as using R_n (found for the current commit). The difference in safe assertions is negative for less than a quarter of the files tested, with the average negative difference among these files being around 22% (i.e., R_o proved 22% fewer assertions than R_n in these files). On the remaining three quarters of the files tested however, R_o proves at least as many assertions as R_n, and thus, R_o tends to be tailored across code versions.

Commits can result in both small and large changes to the code. We therefore also measured the average difference in the number of verified assertions per changed line of code with respect to the oldest commit. For most files, regardless of the number of changed lines, we found that R_o and R_n are equally effective, with changes to 1000 LOC or more resulting in little to no loss in precision. In particular, the median difference in safe assertions across all changes between R_o and R_n was 0 (i.e., R_o proved the same number of assertions safe as R_n), with a standard deviation of 15 assertions. We manually inspected a handful of outliers where R_o proved significantly fewer assertions than R_n (difference of over 50 assertions). These were due to one file from GIT where R_o is not as effective because the widening and narrowing settings have very low values.

RQ4 takeaway: For over 75% of files, tAILOR's recipe for a previous commit (from up to 50 commits previous) remains tailored for future versions of the file, indicating the resilience of tailored recipes across code changes.

5.4 Threats to Validity

We have identified the following threats to the validity of our experiments.

Benchmark selection. Our results may not generalize to other benchmarks. However, we selected popular GitHub projects from different application domains (see Tab. 2). Hence, we believe that our benchmark selection mitigates this threat and increases generalizability of our findings.

Abstract interpreter and recipe settings. For our experiments, we only used a single abstract interpreter, CRAB, which however is a mature and actively supported tool. The selection of recipe settings was, of course, influenced by the available settings in CRAB. Nevertheless, CRAB implements the generic architecture of Fig. 2, used by most abstract interpreters, such as those mentioned at the beginning of Sect. 3. We, therefore, expect our approach to generalize to such analyzers.

Optimization algorithms. We considered four optimization algorithms, but in Sect. 4.3, we explain why these are suitable for our application domain. Moreover, tAILOR is configurable with respect to the optimization algorithm.

Assertion types. Our results are based on four types of assertions. However, these cover a wide range of runtime errors that are commonly checked by static analyzers.

6 Related Work

The impact of different abstract-interpretation configurations has been previously evaluated [54] for Java programs and partially inspired this work. To the best of our knowledge, we are the first to propose tailoring abstract interpreters to custom usage scenarios using optimization.

However, optimization is a widely used technique in many engineering disciplines. In fact, it is also used to solve the general problem of algorithm configuration [31], of which there exist numerous instantiations, for instance, to tune hyper-parameters of learning algorithms [3,52,18] and options of constraint solvers [33,32]. Existing frameworks for algorithm configuration differ from ours in that they are not geared toward problems that are solved by sequences of algorithms, such as analyses with different abstract domains. Even if they were, our experience with tAIlor shows that there seem to be many optimal or close-to-optimal configurations, and even very simple optimization algorithms such as rs are surprisingly effective (see RQ2); similar observations were made about the effectiveness of random search in hyper-parameter tuning [4].

In the rest of this section, we focus on the use of optimization in program analysis. It has been successfully applied to a number of program-analysis problems, such as automated testing [19,20], invariant inference [50], and compiler optimizations [49].

Recently, researchers have started to explore the direction of enriching program analyses with machine-learning techniques, for example, to automatically learn analysis heuristics [27,34,47,51]. A particularly relevant body of work is on adaptive program analysis [28,29,30], where existing code is analyzed to learn heuristics that trade soundness for precision or that coarsen the analysis abstractions to improve memory consumption. More specifically, adaptive program analysis poses different static-analysis problems as machine-learning problems and relies on Bayesian optimization to solve them, e.g., the problem of selectively applying unsoundness to different program components (e.g., different loops in the program) [30]. The main insight is that program components (e.g., loops) that produce false positives are alike, predictable, and share common properties. After learning to identify such components for existing code, this technique suggests components in unseen code that should be analyzed unsoundly.

In contrast, tAIlor currently does not adjust soundness of the analysis. However, this would also be possible if the analyzer provided the corresponding configurations. More importantly, adaptive analysis focuses on learning analysis heuristics based on existing code in order to generalize to arbitrary, unseen code. tAIlor, on the other hand, aims to tune the analyzer configuration to a custom usage scenario, including a particular program under analysis. In addition, the custom usage scenario imposes user-specific resource constraints, for instance by limiting the time according to a phase of the software-engineering life cycle. As we show in our experiments, the tuned configuration remains tailored to several versions of the analyzed program. In fact, it outperforms configurations that are meant to generalize to arbitrary programs, such as the default recipe.

7 Conclusion

In this paper, we have proposed a technique and framework that tailors a generic abstract interpreter to custom usage scenarios. We instantiated our framework with a mature abstract interpreter to perform an extensive evaluation on real-world benchmarks. Our experiments show that the configurations generated by tAIlor are vastly better than the default options, vary significantly depending on the code under analysis, and typically remain tailored to several subsequent code versions. In the future, we plan to explore the challenges that an inter-procedural analysis would pose, for instance, by using a different recipe for computing a summary of each function or each calling context.

Acknowledgements. We are grateful to the reviewers for their constructive feedback. This work was supported by DFG grant 389792660 as part of TRR 248 (see `https://perspicuous-computing.science`). Jorge Navas was supported by NSF grant 1816936.

References

1. The BDDAPRON logico-numerical abstract domains library, `http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron`
2. Amato, G., Rubino, M.: Experimental evaluation of numerical domains for inferring ranges. ENTCS **334**, 3–16 (2018)
3. Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: NIPS. pp. 2546–2554 (2011)
4. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. JMLR **13**, 281–305 (2012)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI. pp. 196–207. ACM (2003)
6. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: A framework for static analysis based on abstract interpretation. In: SEFM. LNCS, vol. 8702, pp. 271–277. Springer (2014)
7. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NFM. LNCS, vol. 6617, pp. 459–465. Springer (2011)
8. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NFM. LNCS, vol. 9058, pp. 3–11. Springer (2015)
9. Chang, B.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: VMCAI. LNCS, vol. 3385, pp. 147–163. Springer (2005)
10. Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: ASE. pp. 332–343. ACM (2016)
11. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: ISOP. pp. 106–130. Dunod (1976)
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
13. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. JLP **13**, 103–179 (1992)

14. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: PLILP. LNCS, vol. 631, pp. 269–295. Springer (1992)
15. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Autom. Softw. Eng. **6**, 69–95 (1999)
16. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. pp. 84–96. ACM (1978)
17. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: FoVeOOS. LNCS, vol. 6528, pp. 10–30. Springer (2010)
18. Falkner, S., Klein, A., Hutter, F.: BOHB: Robust and efficient hyperparameter optimization at scale. In: ICML. PMLR, vol. 80, pp. 1436–1445. PMLR (2018)
19. Fu, Z., Su, Z.: Mathematical execution: A unified approach for testing numerical code. CoRR **abs/1610.01133** (2016)
20. Fu, Z., Su, Z.: Achieving high coverage for floating-point code via unconstrained programming. In: PLDI. pp. 306–319. ACM (2017)
21. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An abstract domain of uninterpreted functions. In: VMCAI. LNCS, vol. 9583, pp. 85–103. Springer (2016)
22. Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J.A., Rinetzky, N., Ryzhyk, L., Sagiv, M.: Simple and precise static analysis of untrusted Linux kernel extensions. In: PLDI. pp. 1069–1084. ACM (2019)
23. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics **30**, 165–190 (1989)
24. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: SAS. LNCS, vol. 6337, pp. 287–303. Springer (2010)
25. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV. LNCS, vol. 9206, pp. 343–361. Springer (2015)
26. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: SAS. LNCS, vol. 10422, pp. 148–168. Springer (2017)
27. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: SAS. LNCS, vol. 9837, pp. 237–256. Springer (2016)
28. Heo, K., Oh, H., Yang, H.: Resource-aware program analysis via online abstraction coarsening. In: ICSE. pp. 94–104. IEEE Computer Society/ACM (2019)
29. Heo, K., Oh, H., Yang, H., Yi, K.: Adaptive static analysis via learning with Bayesian optimization. TOPLAS **40**, 14:1–14:37 (2018)
30. Heo, K., Oh, H., Yi, K.: Machine-learning-guided selectively unsound static analysis. In: ICSE. pp. 519–529. IEEE Computer Society/ACM (2017)
31. Hutter, F.: Automated Configuration of Algorithms for Solving Hard Computational Problems. Ph.D. thesis, The University of British Columbia, Canada (2009)
32. Hutter, F., Babic, D., Hoos, H.H., Hu, A.J.: Boosting verification by automatic tuning of decision procedures. In: FMCAD. pp. 27–34. IEEE Computer Society (2007)
33. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: AAAI. pp. 1152–1157. AAAI (2007)
34. Jeong, S., Jeon, M., Cha, S.D., Oh, H.: Data-driven context-sensitivity for points-to analysis. PACMPL **1**, 100:1–100:28 (2017)
35. Karr, M.: Affine relationships among variables of a program. Acta Inf. **6**, 133–151 (1976)
36. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)

37. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with thresholds for programs with complex control graphs. In: ATVA. LNCS, vol. 6996, pp. 492–502. Springer (2011)
38. Mátyáš, I.: Random optimization. Avtomat. i Telemekh. **26**, 246–253 (1965)
39. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. The Journal of Chemical Physics **21**, 1087–1092 (1953)
40. Mihaila, B., Sepp, A., Simon, A.: Widening as abstract domain. In: NFM. LNCS, vol. 7871, pp. 170–184. Springer (2013)
41. Miné, A.: A few graph-based relational numerical abstract domains. In: SAS. LNCS, vol. 2477, pp. 117–132. Springer (2002)
42. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES. pp. 54–63. ACM (2006)
43. Miné, A.: The Octagon abstract domain. HOSC **19**, 31–100 (2006)
44. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: VMCAI. LNCS, vol. 3855, pp. 348–363. Springer (2006)
45. Monniaux, D., Le Guen, J.: Stratified static analysis based on variable dependencies. ENTCS **288**, 61–74 (2012)
46. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for C-like languages. In: PLDI. pp. 229–238. ACM (2012)
47. Raychev, V., Vechev, M.T., Krause, A.: Predicting program properties from 'big code'. CACM **62**, 99–107 (2019)
48. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education (2010)
49. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS. pp. 305–316. ACM (2013)
50. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: CAV. LNCS, vol. 8559, pp. 88–105. Springer (2014)
51. Singh, G., Püschel, M., Vechev, M.T.: Fast numerical program analysis with reinforcement learning. In: CAV. LNCS, vol. 10981, pp. 211–229. Springer (2018)
52. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In: KDD. pp. 847–855. ACM (2013)
53. Venet, A., Brat, G.P.: Precise and efficient static array bound checking for large embedded C programs. In: PLDI. pp. 231–242. ACM (2004)
54. Wei, S., Mardziel, P., Ruef, A., Foster, J.S., Hicks, M.: Evaluating design tradeoffs in numeric static analysis for Java. In: ESOP. LNCS, vol. 10801, pp. 653–682. Springer (2018)