Data-Driven Design and Evaluation of SMT Meta-Solving Strategies: Balancing Performance, Accuracy, and Cost

Malte Mues
TU Dortmund University
Dortmund, Germany
malte.mues@tu-dortmund.de

Falk Howar
TU Dortmund University
Dortmund, Germany
falk.howar@tu-dortmund.de

Abstract—**Many modern software engineering tools integrate SMT decision procedures and rely on the accuracy and performance of SMT solvers. We describe four basic patterns for integrating constraint solvers (earliest verdict, majority vote, feature-based solver selection, and verdict-based second attempt) that can be used for combining individual solvers into meta-decision procedures that balance accuracy, performance, and cost – or optimize for one of these metrics. In order to evaluate the effectiveness of meta-solving, we analyze and minimize 16 existing benchmark suites and benchmark seven state-of-the-art SMT solvers on $17k$ unique instances. From the obtained performance data, we can estimate the performance of different meta-solving strategies. We validate our results by implementing and analyzing two strategies. As additional results, we obtain (a) the first benchmark suite of unique SMT string problems with validated expected verdicts, (b) an extensive dataset containing data on benchmark instances as well as on the performance of individual decision procedures and several meta-solving strategies on these instances, and (c) a framework for generating data that can easily be used for similar analyses on different benchmark instances or for different decision procedures.**

I. Introduction

Modern software analysis tools are complex and rely on SMT solvers for automated reasoning. Since the introduction of Z3 [1] and CVC4 [2], many research solvers have been developed and specialize in different aspects of certain SMT theories. Researchers and industry frequently use these solvers in their own tools. Decision procedures for constraints over string variables and string operations, e.g., are one key enabler for the formal analysis of Web-applications that rely heavily on string variables for passing URLs and request parameters (e.g., [3], [4], [5], [6], [7]). The development of security analysis tools and advances in the area of string theory solving (e.g., [8], [9], [10], [11], [12], [13]) are closely connected. In the field of Java Web-application analysis the Java String Analyzer [5] was presented more than 15 years ago and evolved into the integration of string method encoding based on modern SMT solvers in symbolic execution engines like Symbolic PathFinder [14]. Nevertheless, until today, newer Java analyzers [7], [15], [16] still report about problems and challenges in the encodings of string operations while exploring Java code.

Recent advances in string theory solving have been accompanied by new tools for testing and fuzzing SMT solvers (e.g., [17], [18], [19], [20]). These tools uncovered various bugs in the implementation of decision procedures. Moreover, at SMT-COMP 2020 the only two competing solvers in the string related categories, CVC4 [10] and Z3str4[1], had a couple of disagreements on the correct solutions for the provided benchmark tasks[2].

While this is natural for cutting edge research tools, users of this technology need strategies for integrating such components without jeopardizing the validity of obtained research results and for minimizing the harmful potential of bugs — or at least methods for analyzing potential bias and confidence in obtained analysis results. Existing approaches that are employed by individual tool developers or research communities are constructive and analytical and include the development of proof/witness-producing analyses (e.g., [21], [22], [23]), portfolio-techniques (e.g., [24]), tool integration platforms [25], and benchmarking (e.g., [26]).

We propose to increase accuracy and validity through integration of multiple decision procedures. In this paper, we demonstrate that a data-driven design of such meta-solving strategies is possible. We discuss a set of four basic patterns for the integration of multiple solvers, balancing response time, performance, accuracy, and cost for obtaining the results. We use these basic patterns for constructing integrated analyses from seven constraint solvers. We validate the design patterns by evaluating the performance of these seven individual solvers and four integrated solvers. To run the evaluation, we pre-processed and prepared existing string benchmarks leading to the first string benchmark suite of SMT string problems with validated expected verdicts. We describe a framework for analyzing data that can be used for the design and evaluation of meta-solving strategies. We propose four different meta-solving strategies using the framework and have implemented two of them. We simulated the other two with the available data and evaluate all four strategies against the

[1]https://z3str4.github.io
[2]https://smt-comp.github.io/2020/disagreements/qf-slia-single-query

seven individual solvers on the benchmark set.

Our evaluation shows that all tested solvers have individual performance profiles and that no solver dominates on all benchmark instances. Nevertheless, CVC4 is the best single solver in the evaluation and can only be outperformed by more expensive meta-solving strategies. Moreover, the *earliest verdict* strategy, which is often touted as a viable solution, was prone to incorrect verdicts in our experiments. Using the introduced *verdict-based second attempt* pattern allows us to increase performance and reliability at moderate costs.

We accompany our paper with a corresponding artifact that consists of all data produced in the experiments (in a database) along with all the scripts used for generating the data reported in tables and figures in this paper as well as of the infrastructure that was used for conducting the experiments. It contains the per instance performance of each solver allowing better comparison between solvers in the future and make it easy to extend the comparison by other solvers. This data is available for others interested in detailed performance data of a solver [27].

Outline. In the next section, we present a selection of relevant related work for this paper. Section III presents solver integration pattern and Section IV describes our preparation of the benchmarks used for the data-driven design of concrete meta-solving strategies presented in Section V and their evaluation in Section VI. We discuss the results and draw conclusions in Section VII and Section VIII.

II. Related Work

The presented approach builds on ideas from three areas: Integration of formal methods, meta SMT-Solving, and string theory solving.

A. Integration of Formal Methods

The integration of formal methods is still challenging as many formal methods tend to be designed as single method, but the recent history shows that integration becames more important. Hähnle et al. [28] discuss this challenge for deductive verification tools. Damiani et al. [29] demonstrate how tool refinement can be used to orchestrate multiple verification tools in a divide-and-conquer style. The electronic tool integration platform [25] was an early approach to standardize how tools could be combined. The SV-COMP uses competition tools to confirm produced witnesses enforcing some kind of provability of the verdict [22].

B. Meta Solving

The idea of solver abstraction layers and meta solving support have been proposed for the first time more than a decade ago (c.f. [30]). The solver abstractions are often developed hand in hand with a verification tool. They allow to access the solvers as a library in the language of the tool and support features the developer of the software verification tool have been interested in. CPACHECKER,e.g., comes along with JAVASMT [31], JDART introduced the abstraction library JCONSTRAINTS [32], and Cok developed JSMTLIB [33] along with the development of SMT-Lib. In the Python ecosystem, PYSMT [34] is very popular as an abstraction layer and for C++, METASMT [35] formed around the KLEE tool. MachSMT [36] is a recently introduced approach for machine learning based selection of an SMT solver based on expected performance.

METASMT supports parallel solving in the KLEE solving chain [37] and is used for speeding up KLEE. In the original METASMT paper [35], the authors described and already noticed the importance of cross checking solver results a decade ago. They crosschecked all the results obtained from different solver backends of METASMT against Z3. It does not support any kind of inter solver checking or unsatisfiable core validation on the fly. PYSMT has a parallel solving support as well, but no other advanced meta solving strategies comparable to the CVCSEQCORES solving strategy we propose in this paper (c.f. Section III).

A unique feature of JCONSTRAINTS is that it supports the validation of SMT-Lib models in the JAVA semantics. This is useful in the case of ground truth label validation and we adapted the code for model validation to match the complete Unicode theory of SMT-Lib 2.6.

ZaligVinder [26] is a framework executing multiple solvers on different benchmarks and visualises the results per benchmark. While this is useful tool and similiar to parts of the proposed workflow in this paper, the ZaligVinder's authors have not run any analysis of the results nor the benchmarks.

C. String Theory Solver

ABC Aydin et al. [38] presented a model counting string theory solver called ABC in 2015 and still maintain it. This research project is embedded into Bultan et al.'s book on string analysis for software verification [13]. The tool is not distributed as a pre-built release binary, so we have built the master[3] branch and used the resulting binary without any further adoptions.

Ostrich Chen et al. presented OSTRICH [12] more recently in 2019 and the development is still active. The solver is supposed to be easily extensible and has one solving strategy that relies on simple functions. For more complex functions and nondeterministic string operations, the solver incorporates the SLOTH [39] solver in the background. SLOTH relies on model checking algorithms like IC3. In this comparison, we used OSTRICH 's official release version 1.0.1[4]. Since ostrich appears to include SLOTH and is presented as the more efficient string solver in the paper on OSTRICH, we excluded SLOTH from our experiments.

Norn Abdulla et al. introduced NORN in 2014 [40]. The paper also established the `Norn` benchmark used until today. One focus of the Norn solver are regular membership queries and this is also dominant in the `Norn` benchmark. While NORN shines on the regular membership queries in SMT problems, the older versions of CVC4 and the Z3STR family used for

[3]on: https://github.com/vlab-cs-ucsb/ABC/commit/8b10049

[4]https://github.com/uuverifiers/ostrich/releases/tag/v1.0.1

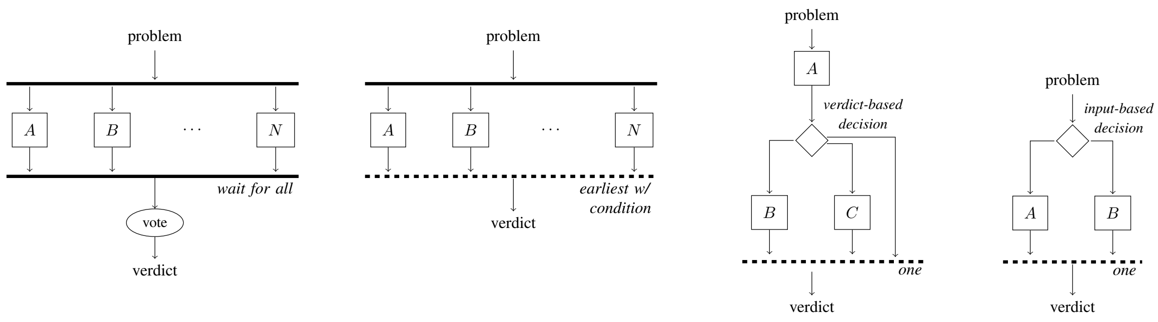

Fig. 1: Basic constraint-solver integration patterns: majority vote, earliest verdict, verdict-based second attempt or validation, feature-based / capability-based solver selection (from left to right).

comparison in the original paper have already been better than NORN back in 2014 on the `Kaluza` benchmark. However, we excluded NORN from the experiments. NORN might have its place in a a meta-solver tackling workloads with many regular expression membership constraints.

Trau Abdulla et al. [11] presented TRAU in 2018. It is a CEGAR based solver, but is not actively maintained[5]. Therefore, we excluded it from the comparison in this paper in favor of actively maintained solvers, but data for Trau is included in the artifact.

Z3 sequence solver (SEQ) Z3 [1] with its own sequence solver [9] is one of the SMT solvers that have been around for over a decade now. As there are a couple of other string backends in Z3 available that have been integrated over time, we refer in the paper to the sequence solver backend as SEQ. We used Z3 in a slightly newer version than the official 4.8.10 release build from its master branch[6].

Z3str-Family Ganesh et al. [8], [41], [42] developed the Z3str solver family. The newest member in the family is Z3STR4[7], while Z3STR3 is part of the official Z3 distribution. Whenever we run Z3STR3, it is the above mentioned Z3 binary configured to use the appropriate backend.

CVC4 Barett et al. [2] introduced CVC4 a very competitive main stream SMT solver that is often used as Z3's counterpart in evaluations. CVC4 is a DPLL(T) style solver with many different specialized theory backends, also one for strings [10]. We used the official 1.8 release version[8].

Princess Rümmer et al. presented PRINCESS [43] a full SMT solver written in Scala. While the others are written in C++. Princess is based on Presburger arithmetic. We used a recent official release[9].

S3 Trinh et al. [4] presented S3, the symbolic string solver for vulnerability analysis with a competitive performance. With the introduction of NORN, some soundness issues [40] have been reported. There was S3P [44] as a follow up version but it is not actively maintained at the moment. We excluded it from our experiments. There was only one update version in 2020 requiring an outdated Ubuntu[10].

[5]https://github.com/diepbp/z3-trau

[6]https://github.com/Z3Prover/z3/commit/4c3c15c

[7]https://z3str4.github.io

[8]https://github.com/CVC4/CVC4/releases/tag/1.8

[9]2020-03-12: http://www.philipp.ruemmer.org/princess-sources.shtml

[10]https://trinhmt.github.io/home/S3/

III. Solver Integration Patterns

The goal of the integration of multiple SMT solvers is usually an increase in the number of solved tasks or some kind of cross-validation between solvers. Depending on the chosen strategy, this increase in performance and reliability is either paid for by longer response times or increased hardware demands. We derive four basic patterns from portfolio solvers proposed in literature (c.f. [30], [32], [35], [45]). Figure 1 sketches them. The remainign section explains these integration in detail. These four basic patterns can be combined hierarchically as a basis for more complex meta-solver or multi-solver decision procedures.

A. Majority Vote

The *Majority Vote* pattern (left of Figure 1) integrates different constraint solvers (or multiple instances of the same solver) A to N by analyzing a problem instance with all solvers. After the return of all solvers or the exhaustion of the resource share per solver (e.g., time limit), a majority vote (or any other aggregation of the obtained individual verdicts) is computed as a final verdict.

Benefits vs. Cost. This pattern can increase the confidence in obtained results and can help with identifying potential bugs in underlying constraint solvers. But it is very expensive as using n solvers can lead to an n-fold increase in response time (if solvers are executed sequentially) and/or resources (in case solvers are executed in parallel). Solvers that solve a problem fast or fail early are optimal candidates for this pattern. Solvers that tend to timeout in the given resource limit without reaching a result are less helpful as they increase the cost without hardening the result in the worst case.

B. Earliest Verdict

The *Earliest Verdict* pattern (center-left of Figure 1) integrates constraint solvers A to N by analyzing a problem instance with all solvers. The first obtained verdict is used as the final verdict — errors and unknowns may be disregarded.

Benefits vs. Cost. This pattern optimizes response times and the number of successful analyses at the expense of parallel resource consumption. It is often instantiated by calling the same constraint solver with different random seeds or configuration options in order to profit from effects of randomization[11] [30]. As we will show in Figure 3, this pattern is not suited for improving correctness and requires sufficient trust in the involved solvers. The invested resources pay off if the total number of solved instances is more important than correctness or if all solvers are known to be reliable upfront.

C. Verdict-based second attempt or validation

This pattern (center-right of Figure 1) integrates multiple constraint solvers by first calling solver A and then deciding on a next step on the basis of the obtained verdict. Potential next steps can be (a) using the verdict of A as final verdict, (b) invoking another solver on the original problem (e.g., in the case that A did not provide a verdict), or (c) using another solver or tool to validate the result computed by A. Strategies for validation comprise (i) evaluating the problem instance with an obtained model, (ii) checking an unsatisfiable core by a second solver, (iii) checking an obtained model and the original problem with another solver, or (iv) switching the encoding or tactic (e.g., the FEAL solver integration [32]).

Benefits vs. Cost. While this pattern can increase confidence in obtained results and the number of successful analyses in a more meaningful way than the previous two patterns, it requires deeper integration between solvers for communicating models and unsatisfiable cores or additional tools for evaluating instances and models. This leads to higher running times and requires support in the solvers for tracking unsatisfiable cores.

D. Feature-based / capability-based solver selection.

This pattern (center-right of Figure 1) integrates multiple solvers by selecting one constraint solver for a problem instance based on features of the problem instance (e.g., used operators and sorts) and based on capabilities of individual solvers. The pattern is, e.g., instantiated in works that train models for selecting a solver that will most likely analyze a problem instance successfully.

Benefits vs. Cost. Feature-based or capability-based solver selection can optimize performance and the number of successful analyses with fewer resources than for example the earliest verdict pattern but relies on the existence and identification of characteristic features in problem instances and/or solvers. This can add additional complexity to the task of designing an integration of constraint solvers and may easily be biased through training sets that are not representative.

IV. SMT String Benchmarks

We need a representative benchmark set that includes expected verdicts as a basis for data-driven design decisions and as a basis for the comparative evaluation of the meta-solving strategies to be developed. Since (a) we have no means of deciding if a given benchmark set is representative of all SMT string problems (or even of some particular application area like program analysis), and since (b) most of the existing benchmark sets from the literature do not include expected verdicts, we simply collected all the benchmarks sets we could find. We use the combined problem instances as the basis for a new benchmark set after removing duplicates to reduce potential bias and optimize resource consumption during experiments. In order to gain at least some confidence in the adequacy of the new benchmark set, we analyze its composition and diversity. Finally, we compute and validate expected verdicts for all instances.

A. Collection and Pre-Processing of Instances

We use the following benchmark suites from the literature. `PyEx`[12] [10],`Pisa` [46],`Norn`[13] [40],`Trau Light`[14] [11], `Leetcode Strings`[15], `IBM Appscan`[16], `Sloth` [39], `Woorpje`[17] [47], `Kaluza`[18] [48], `StringFuzz`[19] [17], `Z3str3`[20], `Cashew` [49], [50], `Joaco` [51], [50], `Stranger` [3], `Kausler` [52], [51], [50], and `BanditFuzz`[21] [18]. We extend this set with SMT problems obtained by running JDART [53], a JAVA program analysis tool, with a modification that exports SMT verification tasks on the JAVA programs of SV-COMP 2021 [54] and refer to the corresponding subset of instances as `SVCOMP`.

Duplicate Removal. Duplicates of the same task in a benchmark inflate the task set artificially without any contribution to the explanatory power of the benchmark set regarding tool performance and, even worse, may introduce bias into conclusions that can be drawn.

We use a form of structural identity for removing duplicates: for a set $\mathbf{x}$ of variables, let $\mathcal{F}_\mathbf{x}$ denote the set of syntactically correct benchmark instances over variables from $\mathbf{x}$. Let $\varphi \in \mathcal{F}_\mathbf{x}$ be a benchmark instance. We can rename variables in φ and write $\varphi[x/y][y/z]$ for the instance that is obtained by replacing all occurrences of variable y with variable x and all occurrences of variable z with variable y simultaneously in φ. A renaming then is a mapping $\pi : \mathbf{x} \mapsto \mathbf{x}$ and we write $\pi(\varphi)$ to denote the application of π to φ, i.e., the instance $\varphi[\pi(x_1)/x_2][\pi(x_2)/x_2][\ldots]$ for $x_1, x_2, \ldots \in \mathbf{x}$.

Definition 1 (Identity up-to renaming). *Two benchmark instances $\varphi, \psi \in \mathcal{F}_\mathbf{x}$ are* identical up-to renaming *iff there exists a bijective renaming $\pi : \mathbf{x} \mapsto \mathbf{x}$ for which $\varphi = \pi(\psi)$.* □

Identity up-to renaming is an equivalence relation on $\mathcal{F}_\mathbf{x}$. Expressions `(assert (= x "abc"))` and `(assert (=`

[11]http://cvc4.cs.stanford.edu/wiki/Tutorials#Parallel_Solving

[12]Taken from: https://cvc4.github.io/papers/cav2017-strings
[13]Taken form: http://user.it.uu.se/~jarst116/norn/
[14]Taken from: https://z3str4.github.io/#_trau_light
[15]Taken from: https://z3str4.github.io/#_leetcode_strings
[16]Taken from: https://z3str4.github.io/#_ibm_appscan
[17]Taken from: https://z3str4.github.io/#_woorpje_word_equations
[18]Taken from: https://z3string.github.io/benchmarks
[19]http://stringfuzz.dmitryblotsky.com/problems/
[20]Taken from: https://z3str4.github.io/#_z3str3_regression
[21]https://github.com/j29scott/BanditFuzz_Public

	Kaluza	*Cashew*	*Stranger*	*Sloth*	*Norn*	*StringFuzz*	*Joaco*	*Appscan*	*SVCOMP*	*Z3str3*	*BanditFuzz*	*PyEx*	*Leetcode*	*Pisa*	*Kausler*	*WWE*	*Trau Light*
Size	47 284	394	4	40	1 027	1 065	94	8	198	243	357	8 414	2 666	12	120	809	100
Unique	2551	393	4	35	1018	913	71	8	154	238	357	8334	2642	12	120	786	100
General																	
`=`	60 701	421	7	1	44	34 050	1 506	10	23	194	–	655 513	171 640	26	286 520	–	–
`not`	28 287	1 171	7	9	968	40	97	19	96	58	207	1 124 039	240 688	13	–	–	–
`Type Cast`	–	–	–	–	–	–	–	–	364	–	–	–	–	–	–	–	–
`ITE`	3 521	–	–	–	–	–	–	8	23	1	–	585 571	133 919	16	–	–	–
`=>`	–	–	–	–	–	–	–	4	–	–	–	–	–	–	–	–	–
`or`	3	–	8	5	–	–	39	3	–	9	–	–	–	5	489	–	–
`and`	35 128	–	–	–	478	–	–	4	319	1	–	3 777 965	166 377	15	–	–	–
`exists`	–	–	–	–	44	–	–	–	–	–	–	–	–	–	–	–	–
`assert`	104 447	5 276	80	86	5 399	94 151	1 410	54	550	641	1 985	8 334	2 646	51	143 141	18 167	300
String																	
`=`	68 298	3 670	77	56	–	58 551	1 153	41	43	298	–	182 964	36 407	50	143 392	17 396	300
`++`	42 239	5 350	52	18	2 832	33 409	785	14	11	278	36	136 868	7 803	13	23 746	34 772	1 200
`to_re`	81 800	1 108	312	30	15 234	44 084	1 964	76	3	79	1	–	–	–	–	–	–
`in_re`	9 597	905	4	33	4 882	1 465	68	8	1	56	242	–	–	–	–	–	–
`at`	–	–	–	–	–	35 790	–	–	23	7	843	151 122	67 454	–	–	–	–
`substr`	–	–	–	–	–	–	–	4	4	9	749	3 287 647	24 533	10	1 232	–	–
`prefixof`	–	–	–	–	–	–	–	4	4	6	200	4 611	–	–	–	–	–
`suffixof`	–	–	–	–	–	–	–	9	–	5	212	–	–	–	–	–	–
`contains`	–	–	–	–	–	–	–	–	1	71	268	327 387	914	27	–	–	–
`indexof`	–	–	–	–	–	–	–	8	–	26	1 181	3 301 028	26 804	7	–	–	–
`replace`	–	–	–	11	–	–	–	–	–	16	790	136 868	3	10	–	–	–
`replace_all`	–	–	–	7	–	–	–	–	–	–	–	–	–	–	–	–	–
`to_int`	–	–	–	–	–	–	5	–	–	–	24	–	–	–	–	–	–
`from_int`	–	–	–	–	–	–	91	–	–	–	–	–	–	–	–	–	–
`lower`	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–
`upper`	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–
Regular Expressions																	
`none`	–	–	–	–	95	–	–	–	–	–	–	–	–	–	–	–	–
`allchar`	–	–	–	6	–	–	150	–	1	–	275	–	–	–	–	–	–
`++`	72 203	195	176	8	6 337	22 655	1 464	8	9	18	18	–	–	–	–	–	–
`union`	–	8	156	–	6 029	19 964	665	60	1	8	–	–	–	–	–	–	–
`inter`	–	–	–	–	–	–	–	–	–	–	16	–	–	–	–	–	–
`*`	–	–	20	19	7 025	20 001	263	8	1	72	82	–	–	–	–	–	–
`+`	–	16	8	5	–	20 110	151	–	–	1	117	–	–	–	–	–	–
`range`	–	–	24	5	1 919	–	83	–	7	3	–	–	–	–	–	–	–
Numeric																	
`str.len`	15 987	701	–	1	1 109	34 135	12	7	667	128	1 022	3 342 014	118 099	8	238	771	–
`neg`	9 071	–	–	–	372	–	19	–	24	–	–	17 428	251	–	–	–	–
`/`	9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
`*`	–	–	–	–	144	–	–	–	–	–	–	–	2	–	–	771	–
`>=`	2 896	276	–	–	–	–	6	–	82	6	207	3 201 201	39 096	–	–	384	–
`>`	–	–	–	1	–	–	1	9	–	18	300	115	11 602	–	–	–	–
`<`	3 042	–	–	–	–	–	1	–	260	9	288	79	382	–	–	–	–
`<=`	1 868	4	–	–	951	85	11	–	324	3	208	–	42 901	–	–	387	–
`-`	1 692	–	–	–	–	–	11	9	39	1	–	5 152 302	122 609	15	238	–	–
`+`	4 703	–	–	–	473	–	2	5	–	5	–	1 868 006	25 562	–	–	–	–
`mod`	–	–	–	–	–	–	–	–	–	–	–	–	15	–	–	–	–

TABLE I: No benchmark suite contained operations `str.<`, `str.<=`, `re.all`, `str.replace_re`, `str.replace_re_all`, `re.comp`, `re.diff`, `re.^`, `re.loop`, `str.is_digit`, `str.to_code`, `str.from_code`, `re.opt`

`y "abc"))`, e.g., are identical up-to renaming since we can rename `x` to `y` and vice versa.

We have removed all duplicate instances based on identity up-to renaming from the original benchmark sets (reducing them from a total of 62 835 to 17 737 tasks). Detailed results are reported in the first two rows of Table I (Size and Unique). Four benchmark suites shrunk significantly in size (more than 10 % reduction): `Kaluza` (reduced by 95 %), `Joaco` (reduced by 24 %), and `StringFuzz` (reduced by 14 %), and `SVCOMP` (reduced by 22 %). For the `Kaluza` set, we found more than 2000 identical instances in one extreme case. A similar observation has been made by Brennan et al. in their work on string normalization [49]. They report that the subset of the Kaluza benchmark used for their research on constraint caching, contains many binary duplicates, i.e., identical files.

After removing duplicates, the benchmark sets still contains many symmetries, e.g., instances `(assert (= x "abc"))` and `(assert (= "abc" x))` or `(assert (< x 5))` and `(assert (> 5 x))`. We did not attempt to reduce symmetry since structure may have an influence on the performance of search heuristics in solvers.

Character Encoding. One problem encountered in the existing benchmarks are inconsistencies in the Unicode encoding. The Unicode theory requires, e.g., using `\u{09}` for encoding the tab character, while some solvers also accept the escaping `\t`, and Z3 traditionally used a hex encoding for characters (`\x09` for tab). In order to avoid inconsistent results as a consequence from these differences, we prepare a

		UNSAT							
		0	1	2	3	4	5	6	7
SAT	0	**18** 3+0	**7** 0+3	**78** 0+7	**78** 0+37	**186** 0+116	**359** 0+352	**448** 0+446	**1235** 0+1235
	1	**228** 131+0	**22** 1+0	**101** 0+6	**125** 0+107	**871** 0+864	**1017** 0+1017	**195** 0+194	
	2	**558** 447+0	**2** 2+0		**1997** 0+1997		**1** 0+1		
	3	**947** 934+0	**20** 17+1	**1** 0+1					
	4	**4125** 4076+0	**6** 6+0						
	5	**1311** 1305+0	**27** 27+0						
	6	**1307** 1305+0							
	7	**2467** 2467+0							

Fig. 2: The figure shows verdict combinations for seven solvers on all benchmark instances. Each cell in row i and column j has i satisfiable verdicts and j unsatisfiable verdicts. The heatmap (yellow) highlights clustering of instances. Every cell shows number of contained instances (bold) as well as number of confirmed models + checked unsatisfiable cores. Green and red coloring indicates the voting-based expected verdict (sat and unsat) in cases where verdicts could not be validated.

corresponding variant of benchmark instances and used these where appropriate without further mention.

B. *Features in Benchmark Suites*

We analyze the composition and diversity in the combined benchmark set by counting occurrences of relevant operations in instances from every source. Table I lists the used operators organized into four groups: general SMT, string theory, regular expressions, and numeric. We arranged the benchmarks into groups as well, focusing on string concatenation with regular expression membership queries on the left to pure word equation benchmarks on the right.

We observe that the group between `Appscan` and `Kausler` use string operations `str.prefixof`, `str.suffixof`, and `str.contains`, which from our experience occur frequently in program security analysis. Benchmark sets right of and including `PyEx` do not use any regular expressions. None of the benchmarks contain the string order operations `str.<` or `str.<=`. Only `SVCOMP` has instances that use `str.lower` and `str.upper`, which are not yet included in the official SMT-Lib v2.6 standard but are already supported by some solvers.

The benchmark sets contain relatively few instances that combine numeric values and strings: the functions for conversion between code points and strings (`str.to_code` and `str.from_code`) do not appear in any benchmark suite and `Joaco` is the only benchmark that contains the `str.from_int` and `str.to_int` operations. Since parsing Unicode byte patterns into string characters and numeric values from strings occurs frequently in Web-applications, we conjecture that the set of presented benchmarks is not adequate for predicting solver performance for analyses in this domain.

C. *Generating Expected Verdicts.*

While expected verdicts are essential as ground truth when comparing the performance of decision procedures, verdicts can only be found for the `Cashew`, `Joaco`, `Kaluza`, `Kausler`, `Stranger`, and `Z3str3` benchmark sets in the literature. We compute expected verdicts for the combined benchmark set in three steps: First, we compute the verdicts of seven SMT solvers (ABC, CVC4, OSTRICH, PRINCESS, SEQ, Z3STR3, Z3STR4) for every problem. As CVC4 and SEQ return most definitive verdicts (cf. Section VI), we try to validate the verdicts of these solvers in a second step: *satisfiable* verdicts from either solver are validated by evaluating corresponding models on problem instances using the JCONSTRAINTS library. *Unsatisfiable* verdicts from CVC4 are validated by checking the corresponding unsatisfiable core with SEQ. Validated verdicts are used as expected verdicts. For satisfiable verdicts, a model with confirming evaluation of the model in the context of the problem is a proof of the verdict. For unsatisfiable verdicts a confirmed unsatisfiable core is at least an argument that two solvers agree on the unsatisfiable part of the problem. For the cases in which validation was not successful, we determine the (likely correct) expected verdict by majority vote. We have not observed a confirmed satisfiable model and a confirmed unsatisfiable core on the same problem. If it happens, the confirmed satisfiable model should define the expected verdict. The test of a satisfiable model is easier from a theoretical point of view than the proof of unsatisfiability and we have implemented the conformance test independent of a solver. In any case, a manual root cause analysis on the instance to identify the reasons for the disagreeing verdicts is recommended.

Results are summarized in Figure 2. Cells group problem instances according to the respective number of obtained *satisfiable* and *unsatisfiable* votes. Cells show the number of corresponding problems (bold) and the number of validated models and validated unsatisfiable cores (m+c). We highlighted accumulations of verdicts with increase in the intensity of yellow as background color in the cell. The biggest cluster containing 4 125 instances got 4 satisfiable votes and 0 unsatisfiable votes. In the data, a cluster with 1 997 instances received 2 satisifiable votes and 3 unsatisfiable votes catching attention. The other larger cluster manifested with clear satisfiable or unsatisfiable votes. Overall, we can provide validated verdicts for 97.48% of all benchmark problems using cross-checking for unsastisfiable cores and model tests for provided models with JCONSTRAINTS. We cannot provide expected verdicts for the 36 instances on which validation failed and voting led to a tie (cells 0/0 and 1/0). Please note that validated models reported in these cells are not an error but were obtained by using SEQ in a mode that tracks unsatisfiable cores and performs differently from the default configuration of SEQ used for voting.

V. DATA-DRIVEN DESIGN OF META-SOLVING STRATEGIES

Using the benchmark suite described in the previous section and the performance data of the seven individual SMT solvers

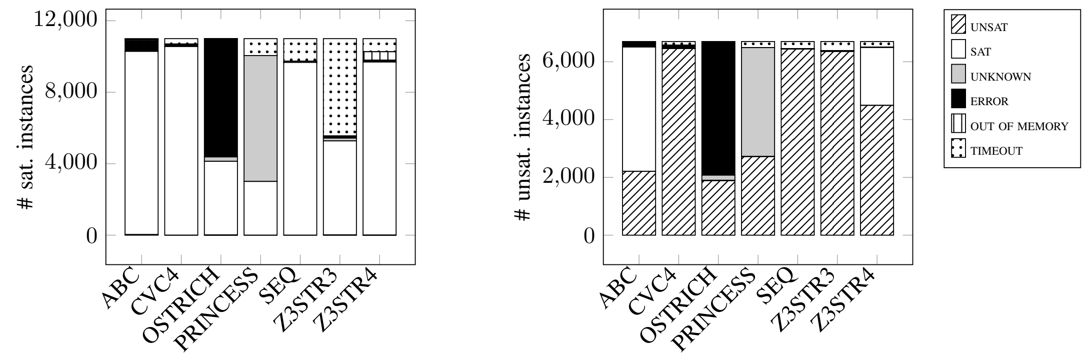

Fig. 3: Verdicts and inconclusive results of SMT solvers ABC, CVC4, Ostrich, Princess, SEQ, Z3str3, and Z3str4 on the benchmark suite presented in Section IV for satisfiable instances (left) and unsatisfiable instances (right).

TABLE II: Performance of individual SMT solvers and meta-solving strategies: (likely) correct verdicts, (likely) incorrect verdicts, unknowns, errors, timeouts, and CPU time (estimated for simulated meta-solving strategies). Vote is omitted as it is used as basis for computing expected verdicts. The table contains 17 701 tasks for wich we could establish a ground truth of the 17 737 tasks in the benchmark.

	Earliest	EarliestTrusted	CvcSeqCores	CvcSeqEval	ABC	CVC4	SEQ	Z3str3	Z3str4	Princess	Ostrich
correct	17 255	**17 449**	16 953	17 290	12 462	17 020	16 117	11 629	14 182	5 730	6 016
unknown	3	247	37	0	0	0	1	163	14	10 814	426
error	0	0	62	2	851	260	76	137	583	1	11 239
timeout	0	0	649	407	53	421	1 505	5 759	920	1 156	6
incorrect	443	5	**0**	2	4 335	**0**	2	13	2 002	**0**	14
CPU time (s)	173 058 s	819 873 s	259 390 s	182 625 s	736 680 s	178 688 s	574 027 s	1 749 373 s	395 750 s	430 048 s	153 314 s

ABC, CVC4, Ostrich, Princess, SEQ, Z3str3, and Z3str4, on this benchmark, we design five meta-solving strategies. Let us start by analyzing the performance data before discussing the resulting design decisions.

A. *Performance Data from Benchmarks*

We focus primarily on analyzing correct verdicts in this section and will analyze resource consumption in Section VI. Figure 3 summarizes the performance of the individual solvers, split into satisfiable instances (left) and unsatisfiable instances (right). We can make two observations:

1) ABC and Z3str4 produce many incorrect verdicts (compared to our expected verdicts) for unsatisfiable benchmark instances.[22] We perform an online transformation of character encodings, as detailed in the previous section. Moreover, the ground truth labels are validated but not proven. Both may contribute to the observed behavior. For the data-driven design of meta-solving strategies, we simply observe that these two solvers frequently produce verdicts that deviate from the majority of solvers on instances deemed unsatisfiable, i.e., we will be skeptical about their *satisfiable* verdicts.
2) Princess and Ostrich solve significantly fewer instances than the other solvers. We will not consider these solvers when optimizing for resource consumption and definitive verdicts without analyzing features of benchmark instances.

B. *Designing Meta-Solving Strategies*

For the remainder of this paper, we focus on five meta-solving strategies (Vote, Earliest, EarliestTrusted, CvcSeqEval, and CvcSeqCores), implementing three integration patterns (*majority vote*, *earliest verdict*, and *verdict-based second attempt*). While we do not present a meta-solving strategy based on the *input-based decision* pattern, we provide some results on training input-based predictors for solver performance in Section VI.

Vote. The Vote strategy instantiates the *majority vote* pattern, executing all seven individual solvers in parallel. All obtained results are aggregated into a final verdict, ignoring timeouts, errors, and out of memory failures. We have not actually implemented this design and instead compute its expected performance (i.e., correct verdicts) from the performance data of the individual solvers. The meaningfulness of the results

[22]For Z3str4, we checked the incorrectly satisfiable instances through Z3str4's Java API, which does yield the expected unsatisifable answers. We assume a bug in the SMT-Lib frontend of Z3str4 but were not able to locate it. For ABC, it was not possible to analyze returned models to find a root cause as ABC's API for accessing models is currently not compatible with the model validator in JConstraints. These types of bugs are to expected in research tools and will hopefully be fixed soon and **must not** be taken as an indicator of bad solver performance!

obtained with this strategy is limited within the scope of this paper since it was used to compute expected verdicts on the benchmark suite.

EARLIEST. The EARLIEST strategy instantiates the *earliest verdict* pattern in a straightforward manner: all solvers are run in parallel and the earliest *satisfiable* or *unsatisfiable* verdict is returned immediately. Failing solvers or *unknown* verdicts are ignored. As for VOTE, we have not actually implemented this strategy and only compute its expected performance.

EARLIESTTRUSTED. This strategy refines the EARLIEST strategy, based on the first observation reported in the previous subsection: an earliest *unsatisfiable* verdict is returned immediately. Any earliest *satisfiable* verdict is disregarded from ABC and Z3STR4 and only used if it is reported by one of the other solvers. We have not implemented this strategy and compute its expected performance.

CVCSEQEVAL. The CVCSEQEVAL strategy instantiates the *verdict-based second attempt or validation* pattern and is based on two observations. First, PRINCESS and OSTRICH solve significantly fewer benchmark instances than the other solvers (cf. above). Second, CVC4 and SEQ use different theories internally and we expect complementing performance profiles. Hence, CVCSEQEVAL starts by calling CVC4 with a timeout of one minute. Then, if CVC4 returns an *unsatisfiable* verdict, it simply returns the verdict. If CVC4 returns a *satisfiable* verdict (and a model), CVCSEQEVAL validates the model by evaluating it on the problem in question, using the JCONSTRAINTS library (cf. Section II). In case the model can be validated, the *satisfiable* verdict is returned. In all other cases, CVCSEQEVAL invokes SEQ for a final verdict. As for CVC4, models are checked for *satisfiable* verdicts and, in case the model cannot be validated, reported as *unknown*. We expect that CVCSEQEVAL computes definitive verdicts for most benchmark instances.

CVCSEQCORES. The CVCSEQCORES strategy refines the CVCSEQEVAL strategy in cases were CVC4 returns an *unsatisfiable* verdict: in these cases, CVCSEQCORES tries to validate the verdict by checking the returned unsatisfiable core with SEQ. As for unvalidated *satisfiable* verdicts, unconfirmed unsatisfiable cores are reported as *unknown*. We expect that CVCSEQCORES produces only very few incorrect verdicts at the price of computing slightly fewer definitive verdicts than CVCSEQEVAL since both solvers have to agree on *unsatisfiable* verdicts. As the unsatisfiable core might be a smaller problem than the original task, we expect that CVCSEQCORES solves more tasks than a solver alone as the unsat core validation is less expensive.

While we only consider individual patterns in this work, it would be easy to combine multiple patterns into complexer meta-solving strategies, e.g., for validating earliest verdicts in a second attempt, balancing response time and accuracy.

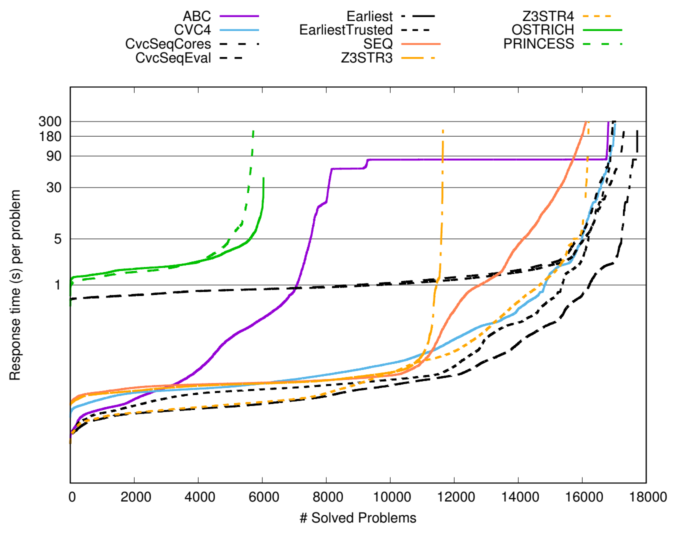

Fig. 4: Number of benchmark instances solved with satisfiable or unsatisfiable answer by individual solvers and meta-solving strategies sorted by increasing response time. EARLIEST and EARLIESTTRUSTED are predictions, other times are measured.

VI. EVALUATION

We conduct a series of experiments in order to evaluate the effectiveness of our data-driven approach to designing meta-solving strategies by addressing the following three concrete research questions.

RQ1. Do meta-solving strategies beat individual solvers in terms of response time, correctness, and cost?

RQ2. What are the observable trade-offs between response time, correctness, and cost?

RQ3. Can we reliably predict if a solver will return a definitive and correct verdict based on features of a problem instance?

We have implemented the CVCSEQEVAL and CVCSEQCORES strategies on top of the JCONSTRAINTS solver abstraction layer in JAVA and have used these implementations to obtain data on performance and resource consumption on the benchmark suite. For the three meta-solving strategies that we did not implement, we estimate resource consumption based on the data recorded in the experiments with the individual solvers and the instantiated integration pattern.

All results were obtained using the BenchExec [55] framework for executing the seven individual solvers (ABC, CVC4, OSTRICH, PRINCESS, SEQ, Z3STR3, and Z3STR4) as well as two meta-solving strategies (CVCSEQEVAL and CVCSEQCORES) on the benchmark suite presented in Section IV. We used an Intel i9-7920X CPU (24 vCores) with 128 GiB RAM running Ubuntu 20.04 LTS. Each run (i.e., one solver on one benchmark problem) was provisioned with 2.5 GB RAM, 1 vCore, and a 5 minute timeout in the BenchExec configuration.

Table II reports the number of correct and incorrect verdicts, other results, and accumulated CPU time for the seven individual constraint solvers and for four meta-solving strategies (analyzing VOTE would not be meaningful since it was used to compute expected verdicts). Figure 4 plots response time (needed time to solve an instance) against correctly solved instances (sorted by increasing response time), showing detailed resource profiles in terms of wall time consumption.

A. *Effectiveness of Meta-Solving (RQ1)*

As can be seen in Table II, the meta-solving strategies outperform the individual constraint solvers with respect to the number of correct verdicts, exceeding the best individual solver (CVC4) by 429 correct verdicts or 2.4% (EARLIESTTRUSTED). Only the CVCSEQCORES meta-solving strategy produces 67 fewer correct verdicts as CVC4.

The results are more mixed for the number of incorrect results: EARLIEST inherits a very high number of incorrect results from ABC and Z3STR4 and while the EARLIESTTRUSTED strategy was explicitly designed to disregard potentially incorrect verdicts from these two solvers, it still produces a greater number of incorrect verdicts than CVC4, SEQ, and PRINCESS. The more conservative CVCSEQEVAL strategy also suffers from 2 incorrect verdicts — which is comparable to SEQ solver and only slightly worse than CVC4. Only the CVCSEQEVAL strategy produces no incorrect verdicts and is on par with CVC4 and the PRINCESS solver in this respect.

With respect to accumulated CPU time, EARLIEST is estimated to be almost as cheap as the cheapest individual solver; EARLIESTTRUSTED, on the other hand, is only better than Z3STR3 and more expensive than most individual solvers. As this strategy require seven cores to run, the paid CPU time masks the fast response time. Figure 4 shows that EARLIEST answers close to $16k$ problems in less than one second response time per problem. EARLIESTTRUSTED still answers over $15k$ problems in less than one second response time. The best single solver answers around $14k$ of the problems in less than one second. The CVCSEQEVAL and CVCSEQCORES meta-solving strategies are almost on par with the faster constraint solvers, where checking of unsatisfiable cores increases resource consumption by 44%. In terms of response times, we observe for these two strategies a constant overhead compared to the solving times of CVC4 and SEQ that vanishes if the response time passes two seconds. This overhead originates from the startup time of the JVM as this strategy is implemented as part of JCONSTRAINTS in JAVA while booth solvers are written in C++. The JVM startup is included in the in measured time presented in the figure. The same overhead is included for OSTRICH and PRINCESS.

B. *Trade-Offs (RQ2)*

Comparing the profiles of the meta-solving strategies, we disregard the EARLIEST meta-solving strategy that seems to be problematic on the concrete benchmark suite and for the concrete selection of solvers as it is strongly affected by incorrect verdicts. The EARLIESTTRUSTED strategy pays for a very competitive response time with a high resource consumption. This may be an interesting performance profile in scenarios were horizontal scaling is cheap if the strategy can be further refined to exclude likely incorrect verdicts. The CVCSEQEVAL strategy balances correct verdicts, resource consumption, and response time very well. The CVCSEQCORES strategy pays for zero incorrect verdicts with a 44% increase in resource consumption (compared to CVCSEQEVAL). The increase in response time seems to be negligible.

C. *Feature-based Solver Selection (RQ3)*

While we do not present an instance of the *input-based decision* integration pattern in this paper, we want to evaluate if the obtained data could help designing such a meta-solving strategy, especially since other works have presented input-based decision strategies. To this end, we have trained random forest classifiers to predict whether a solver can solve a benchmark instance correctly based on the number of occurrences for each SMT primitive in the instance. Training of classifiers was implemented using Python's `sklearn` library and the implementation is available as part of the provided artifact.

Table III shows the achieved precision, recall, and F_1 score, as well as the most influential features for individual solvers. Reported numbers are averages (std. deviations) from a five-fold cross-validation with a randomized $50/50$ split into training set and test set. For individual runs, precision, recall, and F_1 score are computed as weighted averages of the metrics for each label to account for the imbalance in the size of the different classes (some solvers can solve many benchmark instances correctly). As can be seen in the table, the trained classifiers achieve very high values for precision, recall, and F_1 score.

When using input-based decisions with the goal of reducing resource consumption, precision is the most relevant metric, as it expresses the percentage of correct "solver can solve instance" predictions. The very high values for the precision of the trained models seems encouraging but has, of course, been achieved with strongly biased training and test sets for the solvers that solve many benchmark instances correctly, which raises the suspicion that simply always predicting that a solver can solve an instance would already yield a high precision.

A more detailed report of the classification results (not included in the paper but can easily obtained from the data and scripts in the artifact) shows, however, that the trained classifiers do not fall in this trap: even for the solvers that can solve many instances, the trained classifiers achieve high precision and reasonable recall for the smaller (i.e., cannot solve) class. For this class individually, recall is the more interesting metric as it indicates how frequently the classifier would be able to prevent us from using a solver that cannot solve an instance successfully. Precision ranges from 75% to 88% and recall ranges from 50% to 75% in these cases, showing that even though the training set is very unbalanced, classifiers identify features that predict performance. For the solvers with reasonably balanced training sets, precision and recall are high for both classes.

The identified important variables provide additional explanation and validation of the achieved precision. As an example,

TABLE III: Prediction of solver behavior (definitive correct verdict or not) based on features of problem instances; features count contained SMT operations (occurrences per type of operation): Precision, Recall, F_1 Score, and five most important features for seven individual solvers.

Solver	Precision	Recall	F_1 Score	Important Variables									
ABC	0.99 (0.00)	0.99 (0.00)	0.99 (0.00)	`str_eq`	0.09 (0.01)	`str_to_re`	0.09 (0.01)	`emptystr`	0.07 (0.01)	`re_concat`	0.07 (0.01)	`str_concat`	0.07 (0.00)
CVC4	0.98 (0.00)	0.99 (0.00)	0.99 (0.00)	`cast`	0.10 (0.01)	`emptystr`	0.09 (0.01)	`str_to_re`	0.05 (0.00)	`and_op`	0.05 (0.01)	`re_union`	0.05 (0.01)
OSTRICH	0.94 (0.00)	0.92 (0.01)	0.93 (0.00)	`indexof`	0.09 (0.02)	`contains`	0.09 (0.02)	`str_concat`	0.08 (0.01)	`not_op`	0.07 (0.00)	`and_op`	0.07 (0.01)
PRINCESS	0.90 (0.01)	0.91 (0.01)	0.90 (0.00)	`str_eq`	0.09 (0.01)	`contains`	0.08 (0.00)	`indexof`	0.07 (0.01)	`not_op`	0.06 (0.00)	`uminus`	0.05 (0.01)
SEQ	0.96 (0.00)	0.98 (0.00)	0.97 (0.00)	`len`	0.09 (0.00)	`str_concat`	0.08 (0.00)	`str_eq`	0.07 (0.01)	`equals`	0.07 (0.00)	`str_to_re`	0.06 (0.00)
Z3STR3	0.95 (0.00)	0.97 (0.00)	0.96 (0.00)	`plus`	0.10 (0.01)	`indexof`	0.08 (0.01)	`str_eq`	0.07 (0.00)	`contains`	0.07 (0.01)	`not_op`	0.06 (0.00)
Z3STR4	0.97 (0.00)	0.98 (0.00)	0.97 (0.00)	`replace`	0.11 (0.01)	`str_concat`	0.11 (0.00)	`plus`	0.05 (0.00)	`len`	0.05 (0.00)	`substr`	0.05 (0.01)

for CVC4, the `cast` operator is identified as an important variable in the experiments (cf. in Table III). This matches our observation that CVC4 is good in casting between data types. For Z3STR3, the `plus` and `indexof` operators are important for the decision as Z3STR3 does not support the combination of integer theory and string theory.

These observations show that the the learned classifiers (to some degree) use features that relate benchmark instances and solver capabilities. This suggests that learned classifiers could be used for instantiating the *input-based decision* and that a proper evaluation of the achievable performance is worthwhile of further investigation. Finally, the information on important features could alternatively be used for the data-driven manual design of input-based decisions.

VII. Discussion and Threats to Validity

We can now summarize our observations, draw some initial conclusions about the application of meta-solving strategies, and discuss some threats to the validity of the obtained results.

Concept Validity. The different performance profiles of the analyzed meta-solving strategies (w.r.t. accuracy, resource consumption, and response time) demonstrate that it is possible to optimize for different application scenarios. We can easily derive ideas for more involved combinations of solvers, based on the four patterns: If horizontal scaling is cheap as sufficiently many CPU cores are available, taking the first returning solver with filter conditions seems the best strategy and might lead to a further improvement over EARLIESTTRUSTED. If less CPU cores are available, adding more solvers to improve the checking of unsatisfiable cores of CVCSEQCORES (so that the results get closer to the ones obtained with CVCSEQEVAL) may be the right strategy. Currently, the timeouts for checking unsatisfiable cores are a limit for this strategy.

Moreover, we were able to draw conclusions from performance data of individual solvers that guided our design decisions. Since we did not actually implement the strategies we simulated and did not simulate the strategies we implemented, we did not actually demonstrate that our analysis of the response time and resource consumption of meta-solving strategies is accurate. Especially for the large group of benchmark instances that can be solved in fractions of a second, forking of processes and inter-process communication may lead to slower response times and higher resource consumption than estimated, especially if many solvers are used in parallel.

Threats to Internal Validity. We identify two threats to internal validity of results obtained for the analyzed meta-solving strategies: seeding of SMT solvers may impact results and expected verdicts may be wrong.

SMT Solvers use heuristics that are traditionally seeded. Z3 and CVC4 tend to be seeded with fix seeds unless they are altered by passing seeds explicitly. Therefore, we have not observed varying results originating from seeding in our experiments. Nevertheless, running the same solver with different seeds might impact the performance. As most problems in the combined benchmark set have only few variables and the run times for many problems are milliseconds, we do not expect this to have a significant impact in obtained results. We do not expect different random choices in these small sets of variables to change this.

Naturally, majority votes do not guarantee correctness and may lead to spurious results. One could argue that majority votes should hence not be used for the computations of ground truth labels. Following such an argument would not change the reported results in a significant way: In the presented data, validated result are computed for $97.5\,\%$ of all benchmark instances. Majority votes are only used for the remaining $2.5\,\%$ of instances and among these a significant fraction are unanimous. On the other hand, disagreement of solvers may provide guidance towards difficult features and bugs. Finally, even validated verdicts can theoretically be spurious but we deem this highly unlikely as models and unsatisfiable cores were confirmed with independently developed tools.

We want to emphasize the importance of cross checking SMT solver results and the method for the evaluation of meta solving strategies rather than establishing a single strategy. The experiments need to be repeated periodically to be valuable in the long run. A consequence is that our ground truth labels may be wrong if a bug in a solver leads to consistently voting for the wrong answers. Due to the model and unsatisfiable core validation applied along with the majority vote, we do not expect this to be the case in a significant amount of cases in this data set.

Threats to External Validity Results obtained in this study on the performance of individual meta-solving strategies may not generalize well for several reasons.

First, we evaluated the multi-solver strategies and solver only on Unicode theory benchmarks, which is only a subset of the whole SMT world. Then, the study does not consider cloud settings where horizontal scaling is achieved by splitting work across the network layer. We do not expect the *Earliest Verdict* based strategies to perform comparably in such a setting due to the short run times in most cases. Third, the string theory solver field advances quickly and the measurements in this paper will be outdated in the future. We report on a snapshot view on the state of implementations. Finally, as shown in Table I, the benchmarks have slightly different profiles but are still quite homogeneous and skip certain parts (in terms of operators) of the SMT-Lib standard. We tried to control for this by using all benchmarks we could find in the literature. Still, the observations made on the used benchmarks set may not extrapolate well to the complete SMT-Lib Unicode string theory.

Summarizing, we are confident that data-driven design of meta-solving strategies is possible and beneficial. Concrete results depend on the benchmarks that are used for analysis and tuning as well as on the distribution of capabilities in solvers.

VIII. Conclusion

In this paper, we evaluated different SMT meta-solving strategies for string theory solvers in terms of performance, accuracy, and costs. To this end, we first collected benchmark sets used in the literature to evaluate the performance of seven SMT solvers on these benchmarks. After collection, we removed duplicates form the benchmarks reducing the overall size from 62 835 to 17 737 tasks. These tasks are analyzed in more detail regarding their homogeneity in the benchmark sets. We used the computed performance data to generate an expected verdict for each task based on either a validated satisfiable model or a confirmed unsatisfiable core. Otherwise, we use majority voting to define a expected result label.

The paper presents four integration patterns for SMT meta-solving strategies. Based on these patterns, we defined four different meta-solving strategies in addition to the vote strategy used to establish the expected result labels. The evaluation demonstrates that the earliest verdict strategies are only suitable in the SMT domain on string problems, if some kind of solver selection is performed upfront. Otherwise, a earliest verdict strategy will return incorrect results compared to the established ground truth. Cheaper meta-solving strategies that combine in this concrete example only CVC4 and SEQ with answer validation require less parallelization power of the CPU compared to the fastest solver approach and archive comparable results. We conclude that for the string theory domain, meta-solving strategies are well suited for boosting the capability of the SMT decision layer in an analysis. The influence of the SMT solver choice should be mentioned in the evaluation of an algorithm implementation more often in the future. Future work should investigate the portability of the results to other SMT domains.

References

[1] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Springer, 2011, pp. 171–177.

[3] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 154–157.

[4] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1232–1243.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, pp. 1–18, available from `http://www.brics.dk/JSA/`.

[6] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.

[7] M. Mues, T. Schallau, and F. Howar, "Jaint: A framework for user-defined dynamic taint-analyses based on dynamic symbolic execution of Java programs," in *Integrated Formal Methods*, B. Dongol and E. Troubitsyna, Eds. Cham: Springer International Publishing, 2020, pp. 123–140.

[8] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," pp. 55–59, 2017.

[9] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 307–321.

[10] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli, "Scaling up DPLL(T) string solvers using context-dependent simplification," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 453–474.

[11] P. A. Abdulla, M. Faouzi Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Trau: SMT solver for string constraints," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, Oct 2018, pp. 1–5.

[12] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, "Decision procedures for path feasibility of string-manipulating programs with complex operations," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[13] T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin, *String Analysis for Software Verification and Security*. Springer, 2017.

[14] G. Redelinghuys, W. Visser, and J. Geldenhuys, "Symbolic execution of programs with strings," in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, 2012, pp. 139–148.

[15] M. Mues and F. Howar, "JDart: Dynamic symbolic execution for Java bytecode (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 398–402.

[16] A. Shamakhi, H. Hojjat, and P. Rümmer, "Towards string support in JayHorn (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, Eds. Cham: Springer International Publishing, 2021, pp. 443–447.

[17] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "StringFuzz: A fuzzer for string solvers," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 45–51.

[18] J. Scott, F. Mora, and V. Ganesh, "Banditfuzz: A reinforcement-learning based performance fuzzer for smt solvers," in *Software Verification*, M. Christakis, N. Polikarpova, P. S. Duggirala, and P. Schrammel, Eds. Cham: Springer International Publishing, 2020, pp. 68–86.

[19] D. Winterer, C. Zhang, and Z. Su, "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428261

[20] ——, "Validating SMT solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 718–730.

[21] F. Besson, P.-E. Cornilleau, and D. Pichardie, "Modular SMT proofs for fast reflexive checking inside Coq," in *Certified Programs and Proofs*, J.-P. Jouannaud and Z. Shao, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 151–166.

[22] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann, "Correctness witnesses: Exchanging verification results between verifiers," in *Pro-*

ceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 326–337.

[23] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig, "Tests from witnesses," in *International Conference on Tests and Proofs*. Springer, 2018, pp. 3–23.

[24] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," in *Data Mining and Constraint Programming*. Springer, 2016, pp. 149–190.

[25] B. Steffen, T. Margaria, and V. Braun, "The electronic tool integration platform: concepts and design," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 9–30, 1997.

[26] M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, "The power of string solving: Simplicity of comparison," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 85–88.

[27] M. Mues and F. Howar, "Reproduction Package for the ASE 2021 AEC Committee," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5226127

[28] R. Hähnle and M. Huisman, *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools*. Cham: Springer International Publishing, 2019, pp. 345–373.

[29] F. Damiani, R. Hähnle, and M. Lienhardt, "Abstraction refinement for the analysis of software product lines," in *Tests and Proofs*, S. Gabmeyer and E. B. Johnsen, Eds. Cham: Springer International Publishing, 2017, pp. 3–20.

[30] C. M. Wintersteiger, Y. Hamadi, and L. De Moura, "A concurrent portfolio approach to SMT solving," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 715–720.

[31] E. G. Karpenkov, K. Friedberger, and D. Beyer, "JavaSMT: A unified interface for SMT solvers in Java," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2016, pp. 139–148.

[32] F. Howar, F. Jabbour, and M. Mues, "JConstraints: a library for working with logic expressions in Java," in *Models, Mindsets, Meta: The What, the How, and the Why Not?* Springer, 2019, pp. 310–325.

[33] D. R. Cok, "jsmtlib: Tutorial, validation and adapter tools for smt-libv2," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 480–486.

[34] M. Gario and A. Micheli, "PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms," in *SMT workshop*, vol. 2015, 2015.

[35] H. Riener, F. Haedicke, S. Frehse, M. Soeken, D. Große, R. Drechsler, and G. Fey, "metaSMT: focus on your application and not on solver integration," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 5, pp. 605–621, 2017.

[36] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, "MachSMT: A machine learning-based algorithm selector for SMT solvers," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, Eds. Cham: Springer International Publishing, 2021, pp. 303–325.

[37] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 53–68.

[38] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 255–272.

[39] L. Holík, P. Janků, A. W. Lin, P. Rümmer, and T. Vojnar, "String constraints with concatenation and transducers solved efficiently," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158092

[40] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "Norn: An SMT solver for string constraints," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 462–469.

[41] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 114–124.

[42] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, "Z3str2: an efficient solver for strings, regular expressions, and length constraints," *Formal Methods in System Design*, vol. 50, no. 2-3, pp. 249–288, 2017.

[43] P. Rümmer, "A constraint sequent calculus for first-order logic with linear integer arithmetic," in *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. LNCS, vol. 5330. Springer, 2008, pp. 274–289.

[44] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "Progressive reasoning over recursively-defined strings," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 218–240.

[45] M. Mues and F. Howar, "Jdart: Portfolio solving, breadth-first search and smt-lib strings (competition contribution)," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 12652, p. 448, 2021.

[46] T. Tateishi, M. Pistoia, and O. Tripp, "Path-and index-sensitive string analysis based on monadic second-order logic," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 1–33, 2013.

[47] J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, "On solving word equations using sat," in *International Conference on Reachability Problems*. Springer, 2019, pp. 93–106.

[48] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 513–528.

[49] T. Brennan, N. Tsiskaridze, N. Rosner, A. Aydin, and T. Bultan, "Constraint normalization and parameterized caching for quantitative program analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 535–546.

[50] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand, "Benchmark suite for "an integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving"," 2018. [Online]. Available: http://dx.doi.org/10.21227/H2ZQ1N

[51] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand, "An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving," *IEEE Transactions on Software Engineering*, vol. 46, no. 2, pp. 163–195, 2020.

[52] S. Kausler and E. Sherman, "Evaluation of string constraint solvers in the context of symbolic execution," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 259–270.

[53] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman, "JDart: A dynamic symbolic analysis framework," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 442–459.

[54] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 12652, p. 401, 2021.

[55] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 1–29, 2019.