Z3str3: A String Solver with Theory-aware Heuristics

Murphy Berzish and Vijay Ganesh
University of Waterloo
Waterloo
Canada

Yunhui Zheng
IBM Research
Yorktown Heights
USA

***Abstract*—We present a new string SMT solver, Z3str3, that is faster than its competitors Z3str2, Norn, CVC4, S3, and S3P over a majority of three industrial-strength benchmarks, namely, Kaluza, PISA, and IBM AppScan. Z3str3 supports string equations, linear arithmetic over length function, and regular language membership predicate. The key algorithmic innovation behind the efficiency of Z3str3 is a technique we call theory-aware branching, wherein we modify Z3's branching heuristic to take into account the structure of theory literals to compute branching activities. In the traditional DPLL(T) architecture, the structure of theory literals is hidden from the DPLL(T) SAT solver because of the Boolean abstraction constructed over the input theory formula. By contrast, the theory-aware technique presented in this paper exposes the structure of theory literals to the DPLL(T) SAT solver's branching heuristic, thus enabling it to make much smarter decisions during its search than otherwise. As a consequence, Z3str3 has better performance than its competitors.**

I. Introduction

String SMT solvers are increasingly becoming important for security applications and in the context of analysis of string-intensive programs [6], [9], [11], [15], [16], [18], [22]. Many string SMT solvers, such as Z3str2 [23], [24] (and its predecessor Z3str [25]), CVC4 [12], Norn [2], S3 [20] (and its successor S3P [21]), and Stranger (and its successor ABC [4]) have been developed to address these challenges and applications. We have developed the Z3str3 string solver as a native first-class theory solver directly integrated into the Z3 SMT solver [7]. Z3str3 is the primary string solver in the official Z3 codebase. Our tool is competitive with respect to its predecessor Z3str2 and the CVC4 solver, and much faster than Norn, S3, and S3P. Having direct access to the core solver of Z3 has allowed us to develop and implement novel theory-aware DPLL(T) techniques, described below. We follow the latest string SMT language standard supported by all major string solvers, and published on the CVC4 website [12].

A. Contributions

1) **Theory-aware branching:** We leverage the integration between the Z3 SMT solver's DPLL(T) SAT layer (henceforth referred to as the "core solver") and the string solver to guide the search and prioritize certain branches of the search tree over others. In particular, we modify the activity computations of the branching heuristic of the Z3 core solver, making it aware of the structure of the theory literals underlying the Boolean abstraction of the input formula such that "simpler" theory literals are prioritized over more complex ones. The question of whether branching can be made theory-aware was first posed in a paper by Roberto Sebastiani [17]. However, to the best of our knowledge we are the first to propose a theory-aware branching technique which prioritizes certain branches over others in a DPLL(T) setting.
2) **Theory-aware case-split:** We add an optimization to Z3's core solver that enables efficient representation of mutually exclusive Boolean variables in the Boolean abstraction of the input theory formula.
3) **Experimental evaluation:** To validate the effectiveness of our techniques, we present a comprehensive and thorough evaluation of Z3str3, and compare against Z3str2, CVC4, S3, and Norn on several large industrial-strength benchmarks. We could not directly compare against S3P since its source is not available, but summarize the results from their CAV 2016 paper and compare against Z3str3. We also could not compare against Stranger/ABC because they do not produce models, do not support dis-equations over arbitrary string terms, and have correctness issues as noted in their paper [4].

II. Theory-Aware Branching

Several of the key enhancements we make in Z3str3 over Z3str2 involve changes to the Z3 core solver, which handles the Boolean structure of the formula and performs propagation and branching. The first of these enhancements is referred to as **theory-aware branching**. We modify the Z3 core solver to allow theory solvers to give certain literals increased or decreased priority during the search. Consider the case where the solver learns the equality $X \cdot Y = A \cdot B$ for non-constant terms X, Y, A, B. Z3str3, in line with Z3str2, handles this equality by considering a disjunction of three possible arrangements [23], [24]:

Arrangement 1: $X = A$ and $Y = B$
Arrangement 2: $X = A \cdot s_1$ and $s_1 \cdot Y = B$ for a fresh non-empty string variable s_1
Arrangement 3: $X \cdot s_2 = A$ and $Y = s_2 \cdot B$ for a fresh non-empty string variable s_2

Of the three possible arrangements, the first is the simplest to check because it does not introduce any new variables and

only asserts equalities between existing terms. Therefore, we would like Z3's core solver to prioritize checking this arrangement before the others. The advantage gained by theory-aware branching is the ability to give the core solver information regarding the relative importance of each branch, allowing the theory solver to exert additional control over the search. We always prioritize simpler branches over more complex ones.

We implement theory-aware branching as a modification of the branching heuristic in Z3. The default branching heuristic in Z3 is activity-based, similar to VSIDS [13]. The core solver will branch on the literal with the highest activity that has not yet been assigned. Activity is increased additively when a literal appears in a conflict clause, and decayed multiplicatively at regular intervals. Although we are not aware of any other work on theory-aware branching, there has been some work in taking domain-specific knowledge into account in the context of branching heuristics and custom decision strategies [10], [14], [8].

The theory-aware branching technique computes the activity of a literal A as the sum of two terms A_b and A_t, wherein the term A_b is the "base activity", which is the standard activity of the literal as computed and handled by Z3's core solver. The term A_t is the "theory-aware activity". The value of this term is provided for individual literals by theory solvers, and is taken to be 0 if no theory-aware branching information has been provided. This modification causes the core solver to branch on the literal with the highest activity A, taking into account both the standard activity value and the theory-aware activity. Therefore, assigning a (small) positive theory-aware activity to a literal will cause it to have higher activity than usual, making it more likely for the core solver to choose it to branch on. Conversely, assigning a (small) negative theory-aware activity will deter the core solver from choosing that literal. Theory-aware branching in Z3str3 modifies the activities of theory literals as follows:

1) Literals corresponding to arrangements that do not create new variables (as in Arrangement 1 above) are given a large (0.5) A_t. Other arrangements in the same case are given a small (0.1) A_t.
2) Arrangements that allow a variable to become equal to a constant string are given a small (0.2) A_t.
3) When searching for length of strings, literals corresponding to longer length values have small negative (-0.1) A_t.

The values of A_t were chosen to be similar in scale to the initial activity values assigned to literals by the default branching heuristic. Although this technique is currently used by the string solver component, theory-aware branching is also useful in many other contexts where new search paths may have unequal importance, such as non-linear arithmetic.

III. Theory-Aware Case-split

During the search, a theory solver can create terms which encode a disjunction of Boolean literals that are pairwise mutually exclusive, i.e., exactly one of the literals must be assigned true and the others must be assigned false. We refer to this as a **theory-aware case-split**. As an example, consider the case where the string solver learns that a concatenation of two string variables X and Y is equal to a string constant $c = c_1c_2 \dots c_n$ of length n, where each c_i is a character in c. There are $n + 1$ possible ways in which we can split the constant c over X and Y resulting in different arrangements:

- $X = \epsilon, Y = c_1c_2 \dots c_n$
- $X = c_1, Y = c_2c_3 \dots c_n$
- $\dots$
- $X = c_1c_2 \dots c_n, Y = \epsilon$

Note that each of these arrangements represents a case that can be explored by the solver, and also that all of these cases are mutually exclusive (as clearly X cannot be equal to both ϵ and c_1 simultaneously, etc.). Thus, this represents a theory-aware case-split. However, the Boolean abstraction constructed over theory literals hides the fact that these are mutually exclusive cases. A naïve solution is to encode $O(n^2)$ extra mutual exclusion Boolean clauses over these variables. Unfortunately, this would result in very poor performance because of the quadratic blowup in formula size. Another option is to let the congruence closure solver in the Z3 core discover the mutual exclusivity of these Boolean variables. This can result in unnecessary backtracking, unnecessary calls to congruence closure, and, in the worst case, reduces to the same set of mutual exclusion clauses being learned in the form of conflict clauses.

The means of handling such cardinality constraints efficiently has been well-studied; previous work has investigated the possibility of alternate encodings, e.g. totalizers [5] and lazy cardinality [3]. Our implementation, by contrast, shows a way to handle these constraints in the inner loop of the SAT solver in a theory-aware manner. This means that theory solvers do not have to perform rewriting or assert extra clauses to enforce mutual exclusivity of choices. Instead, they can provide this information directly to the core solver, which can use these facts during the search. This saves on the propagation effort of the DPLL(T) framework. Our implementation of this technique is as follows:

1) The theory solver provides the core solver with a set S of mutually exclusive literals that correspond to a theory case-split. This set is maintained by the core solver in a list of all such sets.
2) During branching, the core solver checks if the current branching literal belongs to some such set S. If yes, the current branching literal is assigned true and all other theory case-split literals in S are assigned false. Otherwise, the default branching behaviour is used.
3) During propagation, the core solver may assign a truth value to a literal l in some set S of theory case-split literals. If so, the theory case-split check is invoked, i.e., the core solver checks whether two literals l_1, l_2 in the same set S have been assigned the value true. If this is the case, the core solver immediately generates the conflict clause $(\neg l_1 \vee \neg l_2)$.

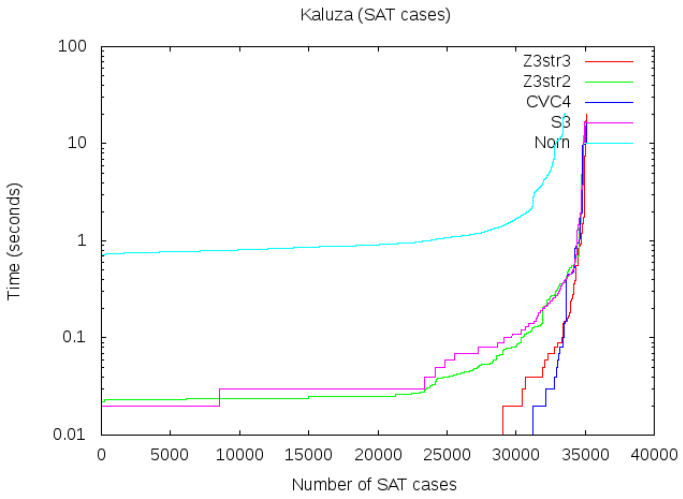

Fig. 1. Cactus plot of string solvers over the Kaluza benchmark (SAT cases).

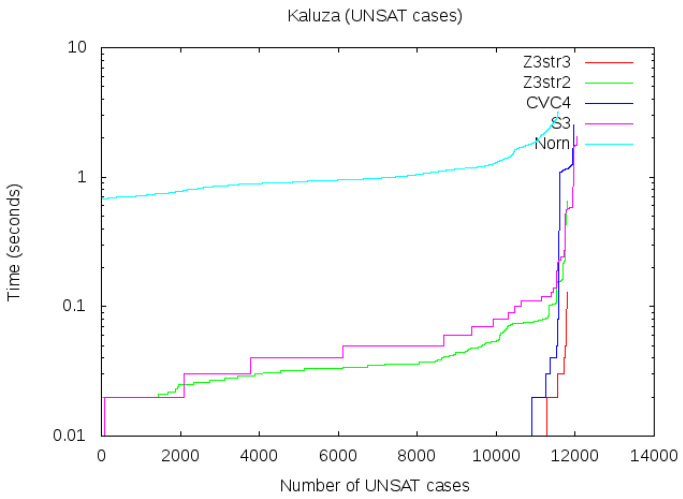

Fig. 2. Cactus plot of string solvers over the Kaluza benchmark (UNSAT cases).

IV. Experimental Results

In this section, we describe the experimental evaluation of the Z3str3 solver to validate the effectiveness of the techniques presented in this paper. We compare Z3str3 against four other state-of-the-art string solvers, namely, Z3str2 [24], [23], CVC4 [12], S3 [20], and Norn [2], across industrial benchmarks obtained from Kaluza [16], PISA [19], and AppScan Source [1]. Each of these benchmark suites draw from real-world applications with diverse characteristics. All experiments were performed on a workstation running Ubuntu 15.10 with an Intel i7-3770k CPU and 16GB of memory. Also, we cross-verified the models generated by Z3str3 against Z3str2 and CVC4, and vice-versa.

Table I shows the summary of results for the Kaluza benchmark. As can be seen from the cactus plots over SAT and UNSAT cases from the Kaluza benchmark, in Figures 1 and 2, Z3str3 (red series) outperforms competing solvers. On SAT cases, Z3str3 is competitive with CVC4 and significantly outperforms all other solvers. On UNSAT cases, Z3str3 is the fastest solver over all cases it can complete. As binaries for S3P are not publicly available, we report the aggregate results presented for this benchmark in the most recent S3P paper [21]. From Table I and Figures 1 and 2 it is clear that Z3str3 is highly competitive with respect to CVC4, and is much faster than other tools. Z3str3 solves more SAT instances than any other tool we benchmarked except S3P, and has the lowest total solving time on non-timeout cases. Notably, over all instances where both solvers finish, Z3str3 solves more cases in total than Z3str2 and completes 30% faster. The unknowns in Z3str3 are because it lacks the feature to handle string equations with overlapping variables, similar to Z3str2. However, Z3str3 has far fewer unknowns than Z3str2.

	Z3str3	Z3str2	CVC4	Norn	S3	S3P
sat	35147	34868	35128	33527	35016	35270
unsat	11799	11799	11957	11568	12049	12014
unknown	223	617	6	1913	0	0
timeout	115	0	0	276	219	0
error	0	0	193	0	0	0
Time (s)	4939.52	3997.63	4851.66	109280.76	10544.06	6972
Time w/o timeouts (s)	2971.02	3997.63	4851.66	97784.00	6164.06	6972

TABLE I
KALUZA BENCHMARK RESULTS. TIMEOUT=20 S. TOTAL TIME INCLUDES ALL SOLVED, TIMEOUT, UNKNOWN, AND ERROR INSTANCES.

Table II shows the results on the PISA benchmark, a set of industrial program analysis instances from IBM. Norn was not able to solve any of the cases as it crashed upon seeing unrecognized string operators (e.g. `indexof`). From Table II we make the following observations. The tools Z3str3, Z3str2, and CVC4 are in agreement on all cases they are able to solve, with CVC4 and Z3str2 timing out on one SAT case which Z3str3 can solve in 0.43 seconds. The results for S3 are significantly worse; it is unable to solve `pisa-009.smt2` while the other three solvers all answer SAT very quickly; and in addition S3 incorrectly answers UNSAT for `pisa-008.smt2`, `pisa-010.smt2`, and `pisa-011.smt2`, on which Z3str3 and (for two of these cases) Z3str2 and CVC4 all return SAT and produce a valid model. The performance of Z3str3 on this benchmark is highly competitive with other solvers, improving on the result from Z3str2.

Table III shows the results on the AppScan benchmark, a second set of industrial instances from IBM. Norn crashed on these cases as well upon seeing unrecognized string operators. From Table III we make the following observations. Z3str3, Z3str2, and CVC4 all agree on all cases they are able to solve. CVC4 performs slightly better than Z3str3 on 3 cases, equally well on 1, and worse on 4, timing out on one case that Z3str3 can solve in 0.73 seconds. In total, on non-timeout cases, CVC4 takes twice as long as Z3str3 (7.89 seconds vs. 4.15 seconds). Z3str2 performs better than Z3str3 on 1 case and worse on 7, taking almost ten times as long on all cases (33.17 seconds vs. 4.15 seconds). S3 returns UNKNOWN on two cases that are solved by the other three tools and produces invalid models which fail cross-validation for four other cases.

input	Z3str3		Z3str2		CVC4		S3	
	result	time (s)	result	time (s)	result	time (s)	result	time (s)
pisa-000.smt2	sat	0.03	sat	0.25	sat	0.08	sat	0.07
pisa-001.smt2	sat	0.05	sat	0.19	sat	0.00	sat	0.07
pisa-002.smt2	sat	0.03	sat	0.10	sat	0.00	sat	0.05
pisa-003.smt2	unsat	0.02	unsat	0.02	unsat	0.01	unsat	0.02
pisa-004.smt2	unsat	0.02	unsat	0.05	unsat	0.39	unsat	0.05
pisa-005.smt2	sat	0.02	sat	0.14	sat	0.02	sat	0.04
pisa-006.smt2	unsat	0.03	unsat	0.05	unsat	0.32	unsat	0.05
pisa-007.smt2	unsat	0.02	unsat	0.05	unsat	0.37	unsat	0.05
pisa-008.smt2	sat	0.43	timeout	20.00	timeout	20.00	unsat X	4.73
pisa-009.smt2	sat	0.60	sat	0.62	sat	0.00	timeout	20.00
pisa-010.smt2	sat	0.02	sat	0.09	sat	0.00	unsat X	0.02
pisa-011.smt2	sat	0.03	sat	0.06	sat	0.00	unsat X	0.02

TABLE II
PISA BENCHMARK RESULTS. TIMEOUT=20 S. X = INCORRECT RESPONSE.

input	Z3str3		Z3str2		CVC4		S3	
	result	time (s)	result	time (s)	result	time (s)	result	time (s)
t01.smt2	sat	0.18	sat	1.31	sat	0.01	sat	0.23
t02.smt2	sat	0.17	sat	0.38	sat	0.01	unknown	0.04
t03.smt2	sat	0.27	sat	9.54	sat	3.82	sat X	0.14
t04.smt2	sat	0.73	sat	4.45	timeout	20.00	sat X	0.10
t05.smt2	sat	0.57	sat	16.84	sat	3.87	sat X	0.55
t06.smt2	sat	0.02	sat	0.15	sat	0.01	sat	0.13
t07.smt2	sat	2.18	sat	0.25	sat	0.00	unknown	0.02
t08.smt2	sat	0.03	sat	0.25	sat	0.17	sat X	0.03

TABLE III
APPSCAN BENCHMARK RESULTS. TIMEOUT=20 S. X = INCORRECT RESPONSE.

Heuristic	Neither	Theory-aware branching	Theory-aware case split	Both
sat	35079	35147	35092	35147
unsat	11799	11799	11799	11799
unknown	221	230	223	223
timeout	185	108	170	115
Time (s)	6252.26	6055.04	5027.35	4939.52

TABLE IV
PERFORMANCE COMPARISON WITH THEORY-AWARE BRANCHING AND THEORY-AWARE CASE SPLIT ENABLED AND DISABLED IN ALL COMBINATIONS. TIMES TAKEN OVER KALUZA BENCHMARK WITH 20 S TIMEOUT. TOTAL TIME INCLUDES ALL SOLVED, TIMEOUT, AND UNKNOWN INSTANCES.

Table IV presents the results of a comparison in which each of the new heuristics in Z3str3, namely theory-aware branching and theory case split, was enabled and disabled in all combinations, in order to measure the change in behaviour of the solver when run over the same benchmark (Kaluza). The experiment clearly shows that both techniques improve the performance of the solver both in isolation and in combination. One intuition for the disparity in performance with respect to each heuristic is that the theory case-split heuristic applies in every instance, due to the frequency of generation of mutually-exclusive options during the search, while the theory-aware branching heuristic is only effective in cases with a large amount of backtracking and search activity, and as such the solver may not benefit from it if the solution is easy to find.

V. DISCUSSION ON EXPERIMENTAL RESULTS, AND CONCLUSIONS

The experimental results discussed here make clear the efficacy of theory-aware branching and case-split. The crucial insight behind these techniques is that biasing the search towards easier branches of the search tree (e.g., an arrangement that doesn't require splitting variables, as opposed to one with overlapping variables) is often very effective since most string constraints obtained from practical applications have the "small model" property. The slogan of theory-aware branching is "bias search towards easy cases first". We also note that Z3str3 and CVC4 do not give any incorrect results, and are more robust than Norn and S3 which sometimes give wrong answers or crash on the benchmarks we used.

REFERENCES

[1] IBM Security AppScan Tool and Source. URL: http://www-03.ibm.com/software/products/en/appscan-source.

[2] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 150–166, 2014.

[3] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and P. J. Stuckey. *To Encode or to Propagate? The Best Choice for Each Constraint in SAT*, pages 97–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[4] A. Aydin, L. Bang, and T. Bultan. *Automata-Based Model Counting for String Constraints*, pages 255–272. Springer International Publishing, Cham, 2015.

[5] O. Bailleux and Y. Boufkhad. *Efficient CNF Encoding of Boolean Cardinality Constraints*, pages 108–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[6] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 307–321, 2009.

[7] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, 2008.

[8] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[9] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[10] A. Erez and A. Nadel. Finding Bounded Path in Graph Using SMT for Automatic Clock Routing. In *Proceedings of the 27th International Conference on Computer Aided Verification*, volume 9207 of *Lecture Notes in Computer Science*, pages 20–36. Springer International Publishing, 2015.

[11] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, 2009.

[12] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 646–662. 2014.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[14] A. Nadel. Routing under constraints. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, pages 125–132, Austin, TX, 2016. FMCAD Inc.

[15] 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*, SAICSIT '12, pages 139–148, 2012.

[16] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, 2010.

[17] R. Sebastiani. Lazy satisfiability modulo theories. In *Journal on Satisfiability, Boolean Modeling and Computation*, volume 3, 2007.

[18] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[19] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, Oct. 2013.

[20] 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*, CCS '14, pages 1232–1243, 2014.

[21] M.-T. Trinh, D.-H. Chu, and J. Jaffar. *Progressive Reasoning over Recursively-Defined Strings*, pages 218–240. Springer International Publishing, Cham, 2016.

[22] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In J. Ferrante and K. McKinley, editors, *PLDI*, pages 32–41. ACM, 2007.

[23] 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*, pages 1–40, 2016.

[24] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. *Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints*, pages 235–254. Springer International Publishing, Cham, 2015.

[25] 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*, ESEC/FSE 2013, pages 114–124, 2013.