SyGuS-Comp 2018: Results and Analysis

Rajeev Alur
University of Pennsylvania, USA
alur@cis.upenn.edu

Dana Fisman
Ben-Gurion University, Israel
dana@cs.bgu.ac.il

Saswat Padhi
University of California, Los Angeles, USA
padhi@cs.ucla.edu

Rishabh Singh
Google Brain, USA
rising@google.com

Abhishek Udupa
Microsoft, USA
abudup@microsoft.com

Syntax-Guided Synthesis (SyGuS) is the computational problem of finding an implementation f that meets both a semantic constraint given by a logical formula φ in a background theory T, and a syntactic constraint given by a grammar G, which specifies the allowed set of candidate implementations. Such a synthesis problem can be formally defined in the SyGuS Input Format (SyGuS-IF), a language that is built on top of SMT-LIB.

The *Syntax-Guided Synthesis competition (SyGuS-Comp)* is an effort to facilitate, bring together and accelerate research and development of efficient solvers for SyGuS by providing a platform for evaluating different synthesis techniques on a comprehensive set of benchmarks. In the 5th SyGuS-Comp, five solvers competed on over 1600 benchmarks across various tracks. This paper presents and analyses the results of this year's (2018) SyGuS competition.

1 Introduction

The *Syntax-Guided Synthesis* competition (SyGuS-Comp) is an annual competition aimed to provide an objective platform for comparing different approaches for solving the syntax-guided synthesis problem. A SyGuS problem takes as input a logical specification φ for what a synthesized function f should compute, and a grammar G providing syntactic restrictions on the implementation for the function f to be synthesized. Formally, a solution to a SyGuS instance (φ, G, f) is a function f_{imp} that is expressible in the grammar G such that the formula $\varphi[f/f_{imp}]$ obtained on replacing f by f_{imp} in the logical specification φ is valid. SyGuS instances are formulated in SyGuS-IF [9], a format built on top of SMT-LIB2 [4].

We report here on the 5th SyGuS competition that took place in July 2018, in Oxford, UK as a satellite event of CAV'18 (The 30th International Conference on Computer-Aided Verification) and SYNT'18 (The 7th Workshop on Synthesis). As in the previous year's competition, there were four tracks: the general track, the conditional linear integer arithmetic (CLIA) track, the invariant synthesis (Inv) track, and the programming by examples (PBE) track. We assume that most readers of this report would already be familiar with the SyGuS problem and the SyGuS-Comp tracks, and thus refer unfamiliar readers to the report on last year's competition [2].

The report is organized as follows.

- Section 2 briefly describes the benchmarks and the key idea behind the submitted solvers.
- Section 3 provides details on the experimental setup.
- Section 4 gives an overview of the results per track.
- Section 5 provides details on the results, given from a single benchmark perspective.
- Section 6 concludes the report with some key takeaway points.

2 Submitted Benchmarks and Solvers

In addition to the benchmarks from the last year's competition, we received over 100 new benchmarks this year, across various competition tracks, which we summarize below in Table 1.

Track	Benchmarks	Contributors
CLIA	15	Kangjing Huang (Purdue University)
General	29	Qinheping Hu and Loris D' Antoni (University of Wisconsin-Madison)
Invariants	21 + 32	Saswat Padhi (UCLA) + Kangjing Huang (Purdue University)
PBE-Strings	10	Woosuk Lee (University of Pennsylvania)

Table 1: New benchmarks contributed to various tracks

Five solvers were submitted to this year's competition: **(1)** CVC4_{2018}, an improved version of CVC4, **(2)** DRYADSYNTH, a solver specialized for conditional linear integer arithmetic, **(3)** EUSOLVER_{2017}, an improved version of EUSOLVER, **(4)** HORNDINI, a solver specialized for constrained horn clauses (CHCs) and **(5)** LOOPINVGEN, a solver specialized for invariant generation problems. Table 2 lists the submitted solvers along with their authors, and Table 3 shows the tracks in which each solver participated.

The CVC4_{2018} solver is based on an approach for program synthesis that is implemented inside an SMT solver [10]. This approach extracts solution functions from unsatisfiability proofs of the negated form of synthesis conjectures, and uses counterexample-guided techniques for quantifier instantiation (CEGQI) that make finding such proofs practically feasible. CVC4_{2018} also combines enumerative techniques, and symmetry breaking techniques [11].

The DRYADSYNTH solver combines enumerative and symbolic techniques. It considers benchmarks in conditional linear integer arithmetic theory (LIA), and can therefore assume all have a solution in some pre-defined decision tree normal form. It then tries to first get the correct height of a normal form decision tree, and then tries to synthesize a solution of that height. It makes use of parallelization, using as many cores as are available, and of optimizations based on solutions of typical LIA SyGuS problems.

The EUSOLVER_{2017} solver uses a divide-and-conquer strategy [1] to find different expressions that satisfy different subsets of the input space, and then unifies them into a solution that works well for the entire space of inputs. Subexpressions are typically found using enumeration techniques and are then unified into the final solutions using decision tree learning [3].

Solver	Authors
CVC4_{2018}	Andrew Reynolds (Univ. of Iowa), Haniel Barbosa (Univ. of Iowa), Andrez Notzli (Stanford), Cesare Tinelli (Univ. of Iowa), and Clark Barrett (Stanford)
DRYADSYNTH	KangJing Huang (Purdue Univ.), Xiaokang Qiu (Purdue Univ.), Qi Tan (Nanjing Univ.), and Yanjun Wang (Purdue Univ.)
EUSOLVER_{2017}	Arjun Radhakrishna (Microsoft) and Abhishek Udupa (Microsoft)
HORNDINI	Deepak DSouza (IISc, Bangalore), P. Ezudheen (IISc, Bangalore), P. Madhusudan (UIUC), Pranav Garg (Amazon), Daniel Neider (MPI-SWS), and Shubham Ugare (IIT, Guwahati)
LOOPINVGEN	Saswat Padhi (UCLA), Rahul Sharma (Microsoft Research), and Todd Millstein (UCLA)

Table 2: List of registered solvers

The HORNDINI solver extends the classical IC3 decision-tree algorithm with the Horn implication counterexamples (Horn-ICE) framework [5], which extends the ICE-learning model. The authors describe a decision-tree learning algorithm that learns from Horn-ICE samples, works in polynomial time, and uses statistical heuristics to learn small trees that satisfy the samples.

The LOOPINVGEN solver [8] for invariant synthesis extends the data-driven approach to inferring sufficient loop invariants from a set of program states [7]. Previous approaches to invariant synthesis were restricted to using a fixed set, or a fixed template for features, e.g., ICE-DT [6] requires the shape of constraints (such as octagonal) to be fixed apriori. Instead LOOPINVGEN starts with no initial features, and automatically learns features as necessary using program synthesis. It reduces the problem of loop invariant inference to a series of precondition inference problems, and uses a counterexample-guided inductive synthesis (CEGIS) loop to revise the current candidate.

	Solver				
Track	CVC4$_{2018}$	DRYADSYNTH	EUSOLVER$_{2017}$	HORNDINI	LOOPINVGEN
CLIA	1	1	1	0	0
INV	1	1	1	1	1
General	1	0	1	0	0
PBE-Strings	1	0	1	0	0
PBE-BV	1	0	1	0	0

Table 3: Participating solvers

3 Experimental Setup

The solvers were run on the StarExec platform [12] with a dedicated cluster of 12 nodes, where each node consisted of two 4-core 2.4 GHz Intel processors with 256 GB RAM and 1 TB hard-disk space. The memory usage limit for each solver run was set to 128 GB, and the wall-clock time limit is set to 3600 seconds (thus, a solver that used all 4 cores could consume at most 14400 seconds of CPU time). The solutions that the solvers produced were checked for both syntactic and semantic correctness. That is, a first postprocessor checked that the produced expression adhered to the grammar specified in the given benchmark, and if this check passes, a second postprocessor checked that the solution adhered to semantic constraints given in the benchmark (by invoking an SMT solver).

4 Results Overview

The primary criterion for winning a track was the number of benchmarks solved, but we also analyzed the time to solve and the the size of the generated expressions. The overall score for each solver was computed as $5N + 3S + F$. Here N denotes the number of benchmarks solved by the solver, F denotes the number of benchmarks solved among the fastest, and S denotes the number of benchmarks for which the size of the generated solution was among the shortest. We used a pseudo-logarithmic scale for F and S. For time to solve, the scale is: $[0,1)$, $[1,3)$, $[3,10)$, $[10,30)$, $[30,100)$, $[100,300)$, $[300,1000)$, $[1000,3600)$, $\geqslant 3600$. That is, the first "bucket" refers to termination in less than one second, the second to termination in one to three seconds and so on. We say that a solver solved a certain benchmark *among the fastest* if the time it took to solve that benchmark is in the same bucket as that of the solver which solved that benchmark in minimum time. Similarly, for expression sizes, the pseudo-logarithmic scale we use is: $[1,10)$, $[10,30)$, $[30,100)$, $[100,300)$, $[300,1000)$, $\geqslant 1000$, where expression size is the number of nodes in the SyGuS parse-tree. We also report on the number of benchmarks *solved uniquely* by a solver, *i.e.* the number of benchmarks which no solver other than the particular solver could solve.

In Figure 1, we show the number of benchmarks solved, the number of benchmarks solved among the fastest, and the number of synthesized expressions among the smallest size; per solver per track.

Figure 1: The number of benchmarks solved by different solvers across all tracks, the number of benchmarks a solver solved among the fastest, and the number of benchmarks for which a solver generated an expression among the smallest size.

		Compiler Optimizations and Bit Vectors	Let and Motion Planning	Invariant Generation with Bounded Ints	Invariant Generation with Unbounded Ints	Multiple Functions	Arrays	Hackers Delight	Integers	Program Repair	ICFP	Cryptographic Circuits	Instruction Selection	**Total**
Number of benchmarks		32	30	28	28	32	35	69	34	18	50	214	28	598
Solved	CVC4$_{2018}$	16	17	24	24	13	31	62	34	17	50	160	0	448
	EUSOLVER$_{2017}$	16	10	24	23	18	31	53	33	14	50	148	0	420
	CVC4$_{2017}$	15	15	24	24	12	31	62	34	17	48	116	0	398
Fastest	CVC4$_{2018}$	15	15	22	24	9	31	59	33	16	23	119	0	366
	EUSOLVER	13	1	12	11	14	5	29	15	12	45	109	0	266
	CVC4$_{2017}$	12	9	16	14	9	24	60	33	6	20	49	0	252
Uniquely	CVC4$_{2018}$	1	2	0	0	0	0	0	0	2	0	7	0	12
	EUSOLVER	3	0	0	0	6	0	0	0	0	0	0	0	9
	CVC4$_{2017}$	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4: The performance of various solvers across all categories of the general track

General Track In the general track, CVC4$_{2018}$ solved the most number of benchmarks (448), and EUSOLVER$_{2017}$ came second, solving 420. We note that the new version CVC4$_{2018}$ is significantly better than the previous version CVC4$_{2017}$, which could only solve 398 benchmarks. The same order appears in the number of benchmarks solved among the fastest: CVC4$_{2018}$ with 366, EUSOLVER$_{2017}$ with 266, and CVC4$_{2017}$ with 252. Finally, we note that CVC4$_{2018}$ is able to solve 12 benchmarks that no other solver could solve, and similarly there are 9 benchmarks that only EUSOLVER could solve.

We partitioned the benchmarks of the general track to a number of categories, each containing a set of related benchmarks. The results per category are given in the Table 4. We observe that EUSOLVER$_{2017}$ preformed significantly better in the "Multiple Functions" and "ICFP" categories. While the CVC4$_{2018}$ solver preformed better in the other categories, none of the solvers could solve any of the benchmarks from the "Instruction Selection" category.

Conditional Linear Arithmetic Track In the CLIA track, CVC4$_{2018}$ and DRYADSYNTH had a close competition. CVC4$_{2018}$ solved 85 out of 88 benchmarks, DRYADSYNTH solved 84 benchmarks, and EUSOLVER$_{2017}$ solved 81 benchmarks. In terms of the time to solve, DRYADSYNTH solved 79 benchmarks among the fastest, CVC4$_{2018}$ solved 74, followed by EUSOLVER$_{2017}$ which solved 29 among the fastest. There were two benchmarks that were solved uniquely by DRYADSYNTH, and one that was solved uniquely by CVC4$_{2018}$.

Invariant Generation Track In the invariant generation track, the LOOPINVGEN solver solved 115 out of 127 benchmarks, CVC4$_{2018}$ solved 109, DRYADSYNTH solved 103, HORNDINI solved 68 and

EUSOLVER$_{2017}$ solved 61 benchmarks. In terms of the time to solve, LOOPINVGEN solved 102 benchmarks among the fastest, followed by CVC4$_{2018}$ which solved 82, DRYADSYNTH which solved 81, HORNDINI which solved 21, and EUSOLVER$_{2017}$ which solved 15. There was one benchmark that was solved by a unique solver – the `fib_17n.sl` benchmark solved by LOOPINVGEN.

Programming By Example (Bit Vectors) Track In the PBE track on the theory of bit vectors, the CVC4$_{2018}$ solver solved 724 out of 750 benchmarks and EUSOLVER$_{2017}$ solved 677 benchmarks. In terms of the time to solve, CVC4$_{2018}$ solved 508 benchmarks among the fastest, and EUSOLVER$_{2017}$ solved 468. However, EUSOLVER$_{2017}$ generates shorter expressions than CVC4$_{2018}$ in significantly many cases. There were four benchmarks that were solved uniquely by CVC4$_{2018}$, and one benchmark that was solved uniquely by EUSOLVER$_{2017}$.

Programming By Example (Strings) Track In the PBE track on the theory of strings, the CVC4$_{2018}$ solver solved 102 out of 118 benchmarks, and EUSOLVER$_{2017}$ solved 79 benchmarks. In terms of the time to solve, CVC4$_{2018}$ solved 98 benchmarks among the fastest, and EUSOLVER$_{2017}$ solved 79. We note again that EUSOLVER$_{2017}$ generates shorter expressions than CVC4$_{2018}$ in several cases. There were 21 benchmarks that were solved uniquely by CVC4$_{2018}$.

5 Detailed Results

In this section we show the results of the competition from the benchmark's perspective. For a given benchmark we would like to know: **(1)** how many solvers solved it **(2)** what are the minimum and maximum times required to solve **(3)** what are the minimum and maximum sizes of solutions generated **(4)** which solver solved the benchmark the fastest, and **(5)** which solver produced the smallest expression.

We present the results in groups organized per track and category. For instance, the top plot in Figure 6 presents the details for program repair benchmarks from the general track. The black bars above the *y*-axis show the range of time taken to solve across the various solvers, in our pseudo logarithmic scale. Inspect for instance benchmark `t2.sl`. The black bar indicates that the fastest solver takes less than 1 second, and the slowest one takes between 100 to 300 seconds. The black number above the black bar indicates the exact number of seconds (floor-rounded to the nearest second) it took the slowest solver to solve a benchmark (and ∞ if at least one solver exceeded the time bound). Thus, we can see that for `t2.sl`, the slowest solver took 138 seconds. The white number at the lower part of the bar indicates the time taken by the fastest solver. Thus, we can see that for `t2.sl`, the fastest solver required less than 1 second. The colored squares/rectangles below the black bar indicate which solvers were among the fastest to solve that benchmark (according to the solvers' legend at the top). For instance, we can see that CVC4$_{2018}$ and EUSOLVER$_{2017}$ were the fastest to solve `t2.sl`, solving it in less than a second, and that among the solvers that solved `t4.sl` only EUSOLVER$_{2017}$ solved it in less than a second.

Similarly, the gray bars below the *y*-axis indicate the range of expression sizes in pseudo-logarithmic scales, where the size of an expression is determined by the number of nodes in its parse tree. The black number at the bottom of the gray bar indicates the exact size of the largest solution (or ∞ if it exceeded 1000), and the white number at the top of the gray bar indicates the exact size of the smallest solution. When the smallest and largest size of expressions are in the same pseudo-logarithmic bucket, as is the case in `t2.sl`), we indicate the expression size only in black. The colored squares/rectangles above the gray bar indicate which solvers were amongst the ones that produced the smallest expression (according to the solvers' legend at the top). For instance, for `t20.sl` the smallest expression produced had size 3, which is produced only by EUSOLVER$_{2017}$.

Finally, the top x-axis indicates the number of solvers that solved a particular benchmark. For instance, in Figure 6, only one solver solved `t6.sl`, two solvers solved `t14.sl`, three solvers solved `t2.sl`, and no solver solved `t13.sl`. Note that for the benchmarks that no solver is able to solve, the black bars indicate the range of time taken by solvers to terminate. When no solver produces a correct result, there are no colored squares/rectangles below the black bars, as is the case for `t13.sl`.

6 Summary

This year's competition consisted of over 1600 benchmarks, 107 of which where contributed this year. Five solvers competed this year, one of which was submitted by developers creating a tool for SyGuS-Comp for the first time. All tools preformed remarkably well, on both existing and new benchmarks. In particular, more than 74% of the current set of benchmarks from the general track are now solved. However, there are several classes of problems that are still challenging for the current solvers, especially the "Instruction Selection" and "Multiple Functions" categories; and we hope the developers would continue to improve their SyGuS techniques and advance the state of the art.

References

[1] Rajeev Alur, Pavol Cerný & Arjun Radhakrishna (2015): *Synthesis Through Unification*. In: *Computer Aided Verification - 27th International Conference, CAV, Proceedings, Part II*, pp. 163–179.

[2] Rajeev Alur, Dana Fisman, Rishabh Singh & Armando Solar-Lezama (2017): *SyGuS-Comp 2017: Results and Analysis*. In: *Proceedings of the Sixth Workshop on Synthesis, SYNT@CAV, EPTCS* 260, pp. 97–115.

[3] Rajeev Alur, Arjun Radhakrishna & Abhishek Udupa (2017): *Scaling Enumerative Program Synthesis via Divide and Conquer*. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Proceedings, Part I*, pp. 319–336.

[4] Clark Barrett, Aaron Stump & Cesare Tinelli: *The SMT-LIB Standard Version 2.0*.

[5] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg & P. Madhusudan (2018): *Horn-ICE Learning for Synthesizing Invariants and Contracts*. *PACMPL* 2(OOPSLA), pp. 131:1–131:25.

[6] Pranav Garg, Daniel Neider, P. Madhusudan & Dan Roth (2016): *Learning Invariants using Decision Trees and Implication Counterexamples*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pp. 499–512.

[7] Saswat Padhi, Rahul Sharma & Todd D. Millstein (2016): *Data-driven precondition inference with learned features*. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pp. 42–56.

[8] Saswat Padhi, Rahul Sharma & Todd D. Millstein (2017): *LoopInvGen: A Loop Invariant Generator based on Precondition Inference*. *CoRR* abs/1707.02029.

[9] Mukund Raghothaman & Abhishek Udupa (2014): *Language to Specify Syntax-Guided Synthesis Problems*. *CoRR* abs/1405.5590.

[10] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli & Clark W. Barrett (2015): *Counterexample-Guided Quantifier Instantiation for Synthesis in SMT*. In: *Computer Aided Verification - 27th International Conference, CAV, Proceedings, Part II*, pp. 198–216.

[11] Andrew Reynolds & Cesare Tinelli (2017): *SyGuS Techniques in the Core of an SMT Solver*.

[12] Aaron Stump, Geoff Sutcliffe & Cesare Tinelli (2014): *StarExec: A Cross-Community Infrastructure for Logic Solving*. In: *Automated Reasoning - 7th International Joint Conference, IJCAR, Proceedings*, pp. 367–373.

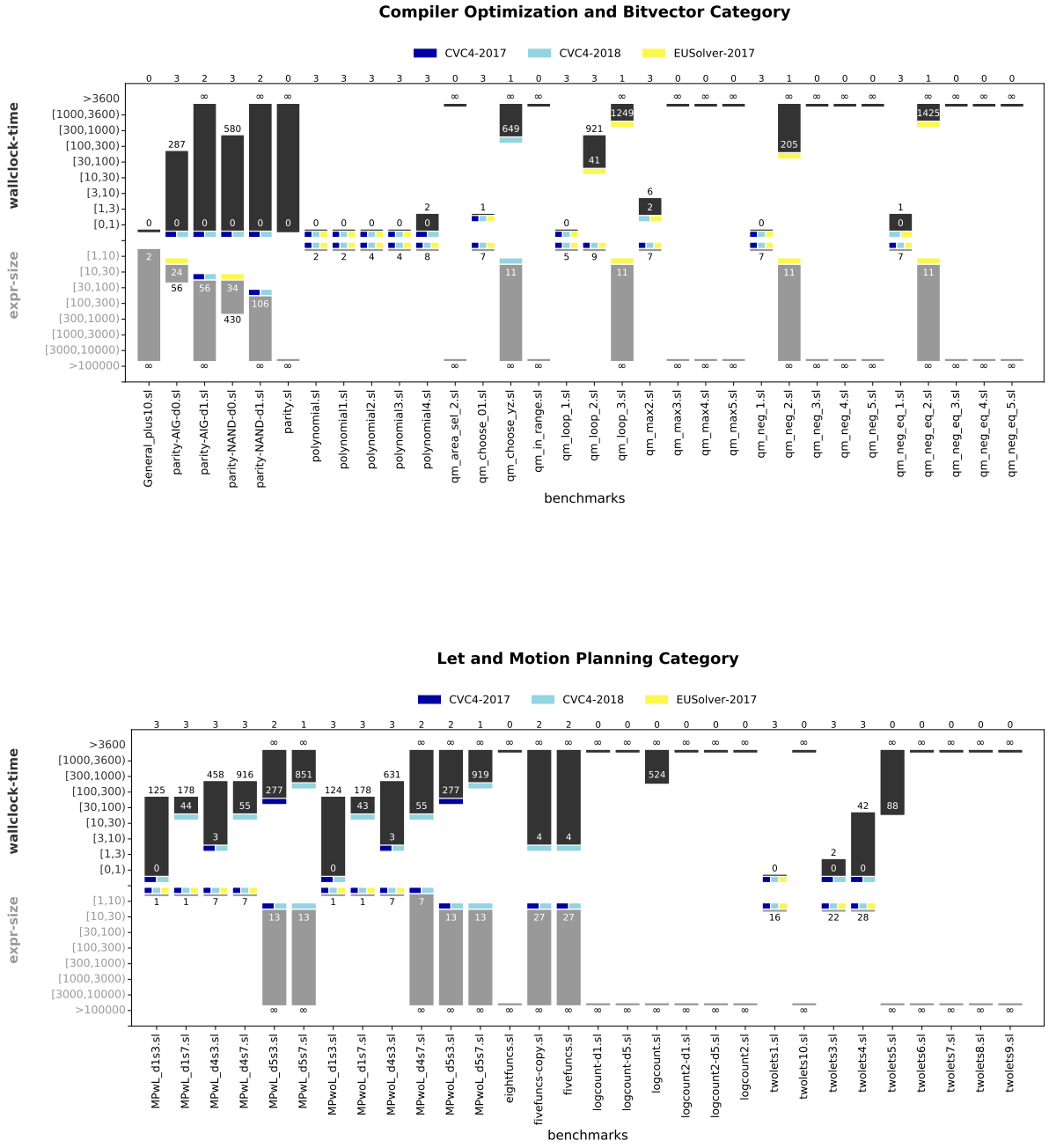


Figure 2: Evaluation of compiler optimizations, bitvectors, let and motion planning, and program repair categories of the General track.

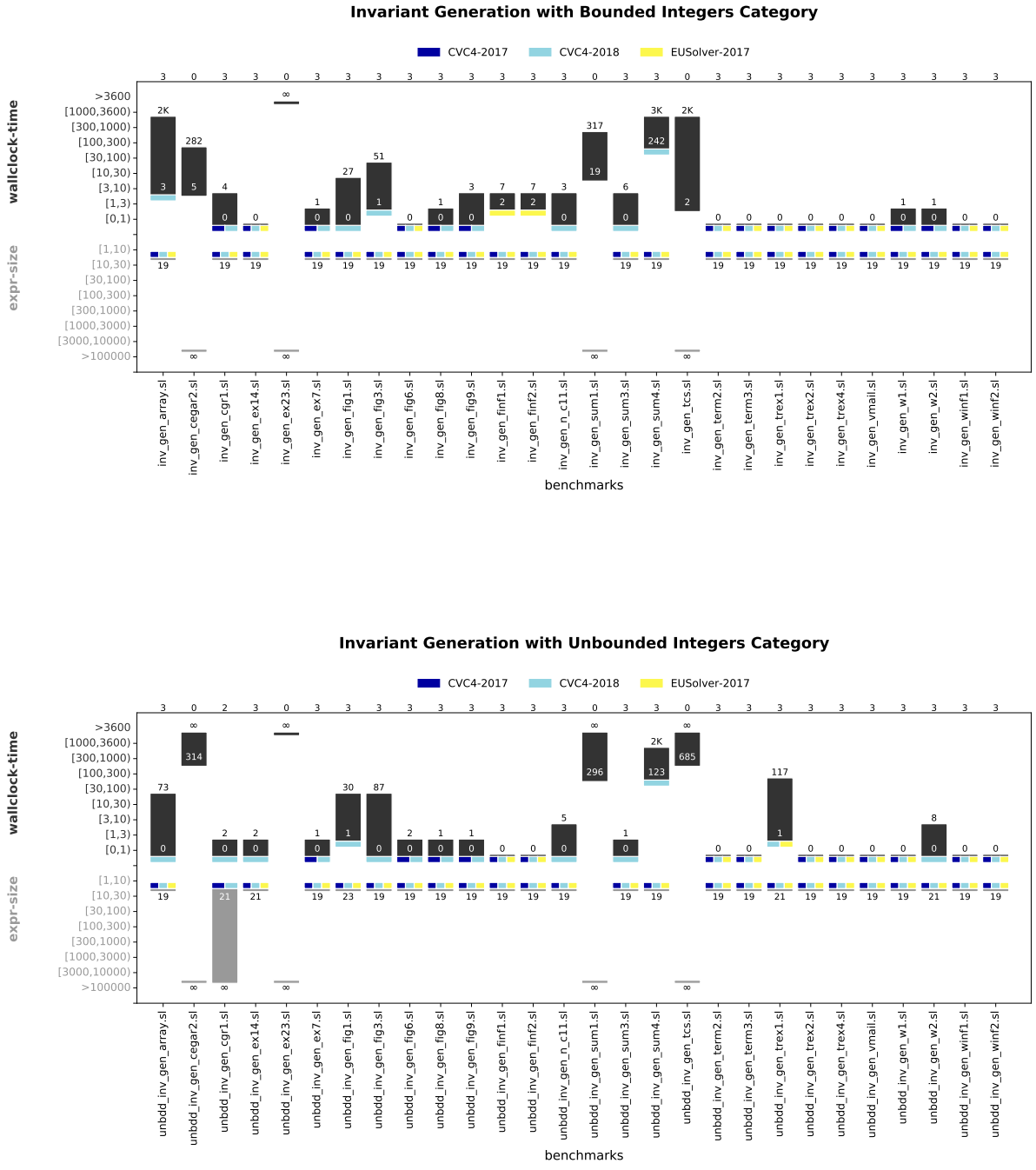


Figure 3: Evaluation of invariant generation categories of the General track.

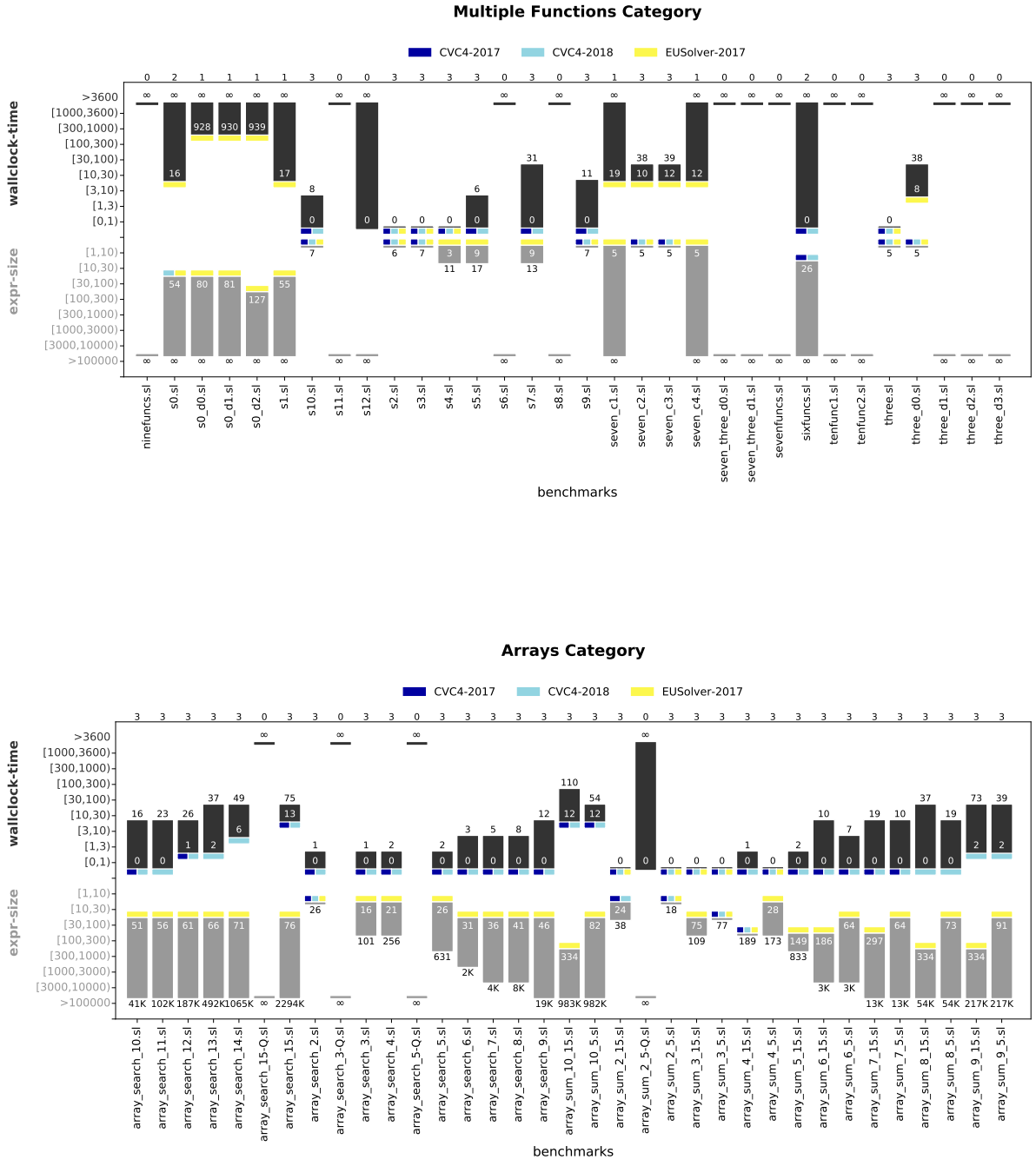


Figure 4: Evaluation of multiple functions and arrays categories of the General track.

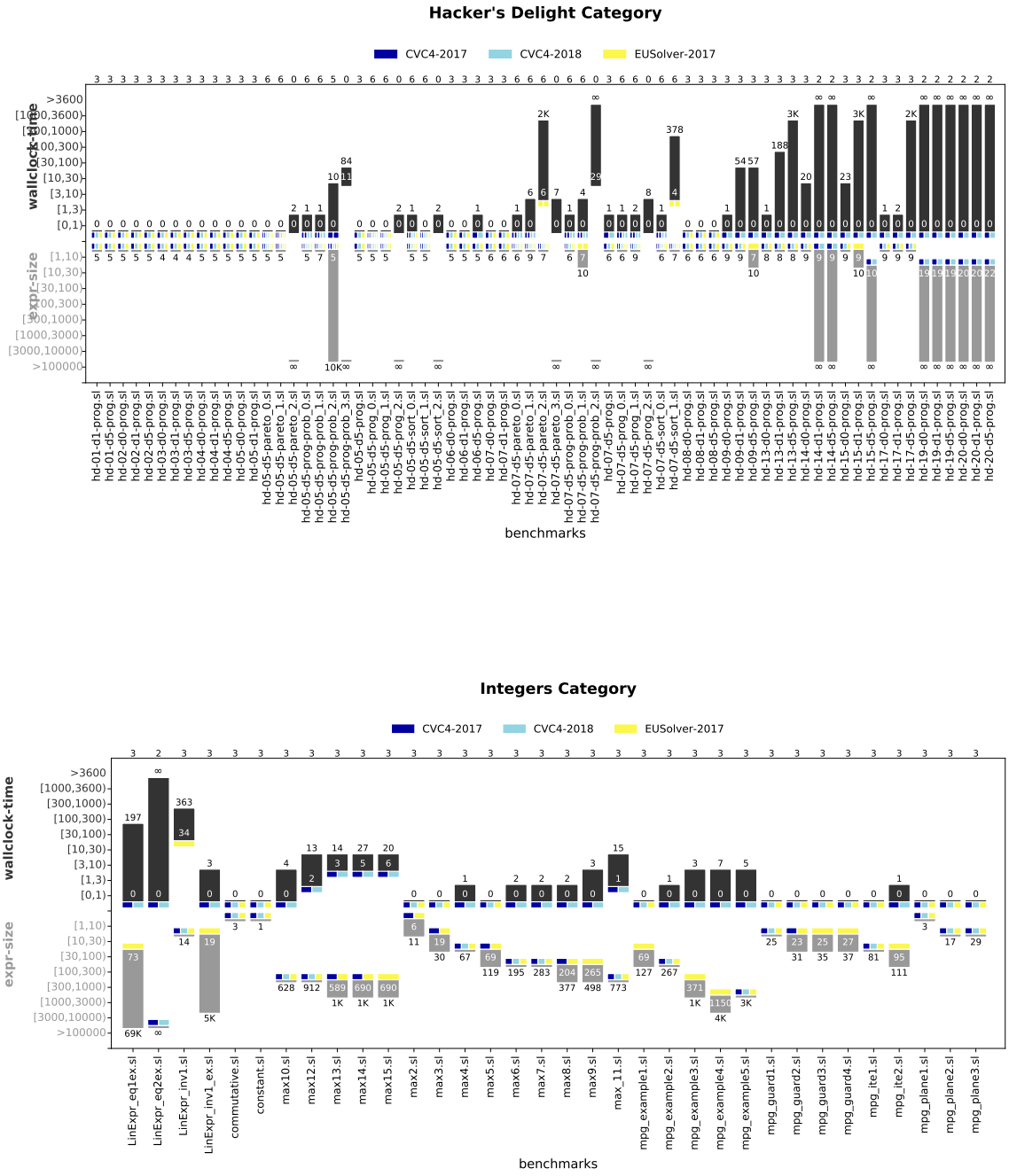


Figure 5: Evaluation of hacker's delight and integers categories of the General track.

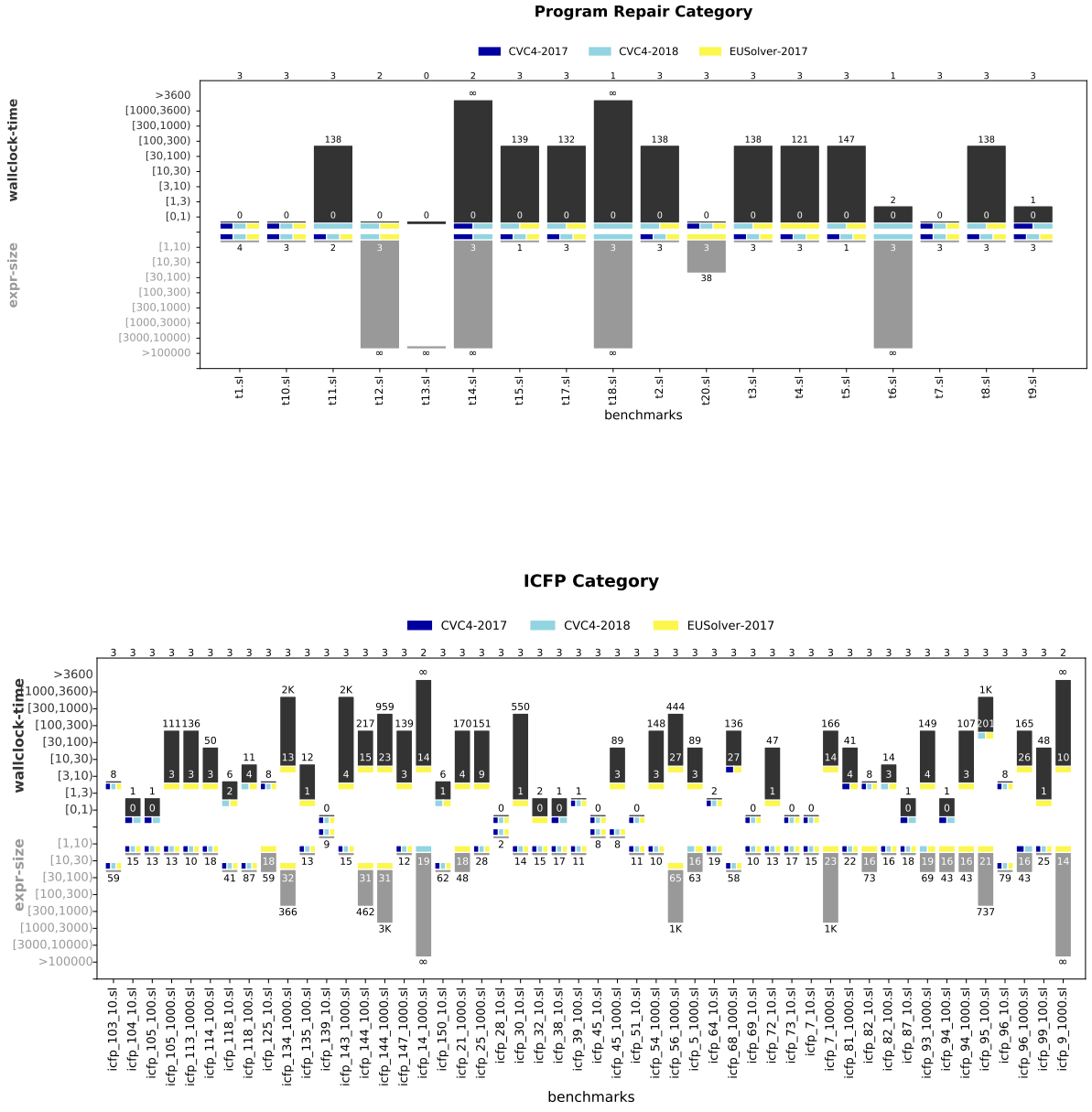


Figure 6: Evaluation of program repair and ICFP categories of the General track.

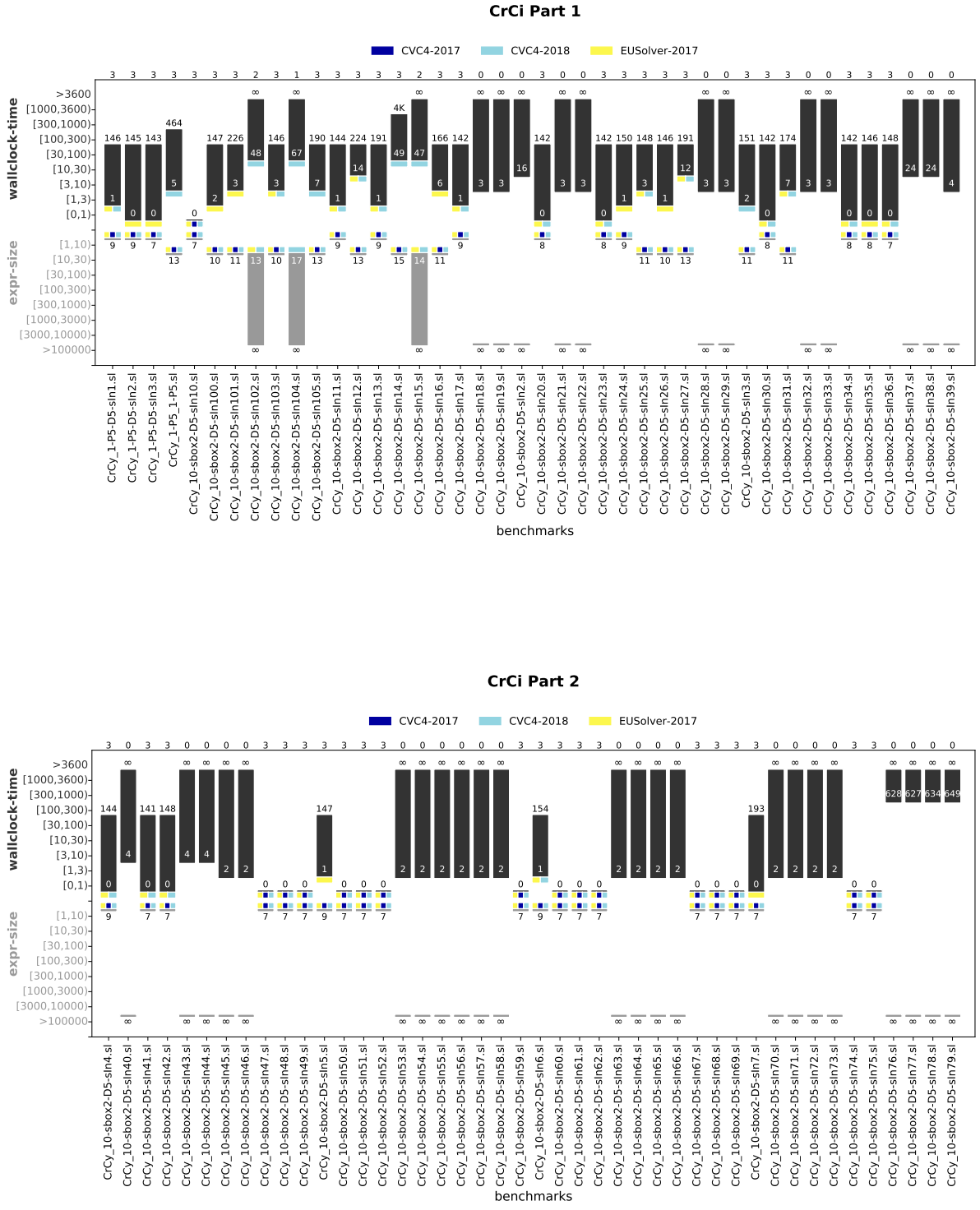


Figure 7: Evaluation of crypto circuits category of the General track (Parts 1 & 2).

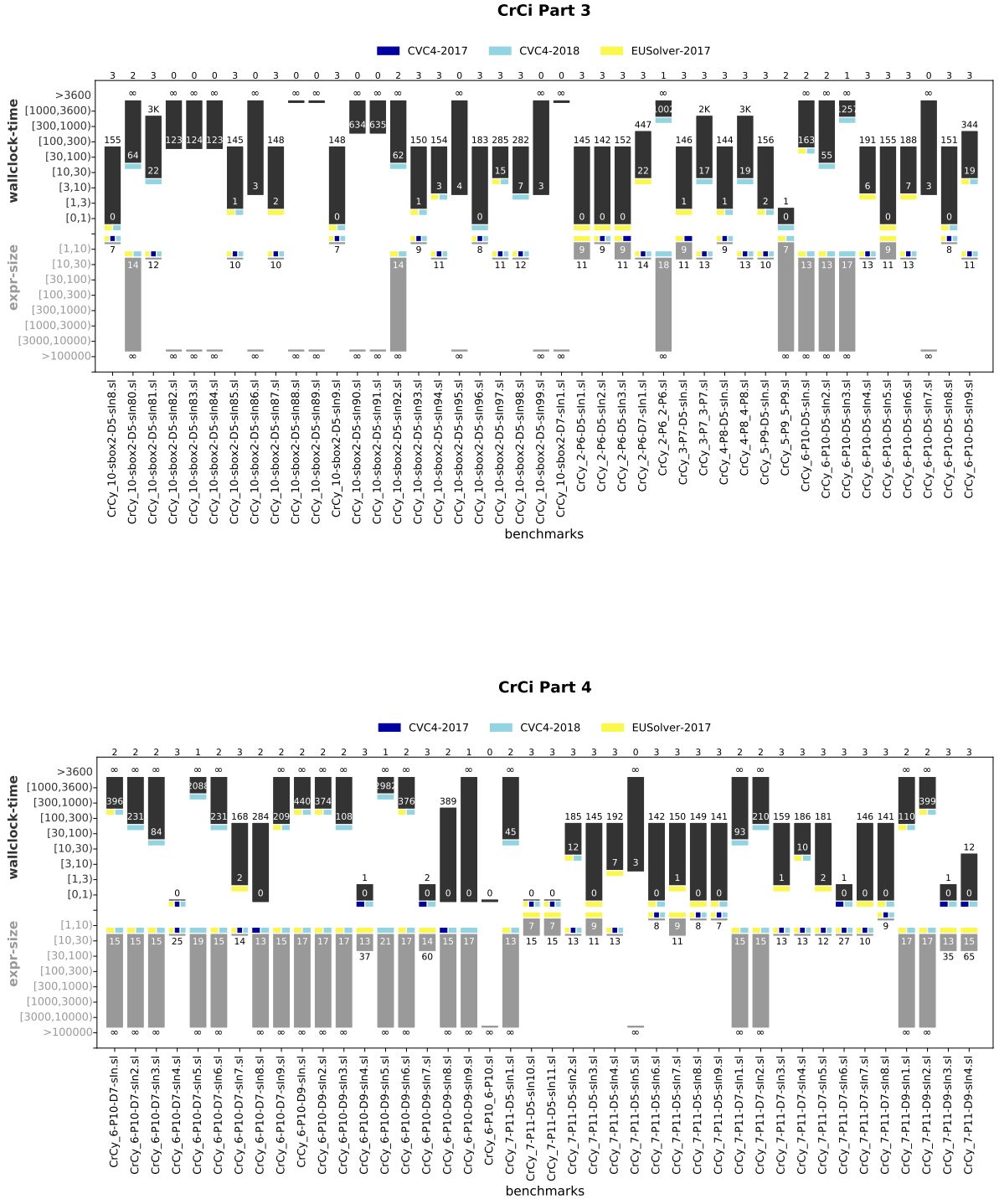


Figure 8: Evaluation of crypto circuits category of the General track (Parts 3 & 4).

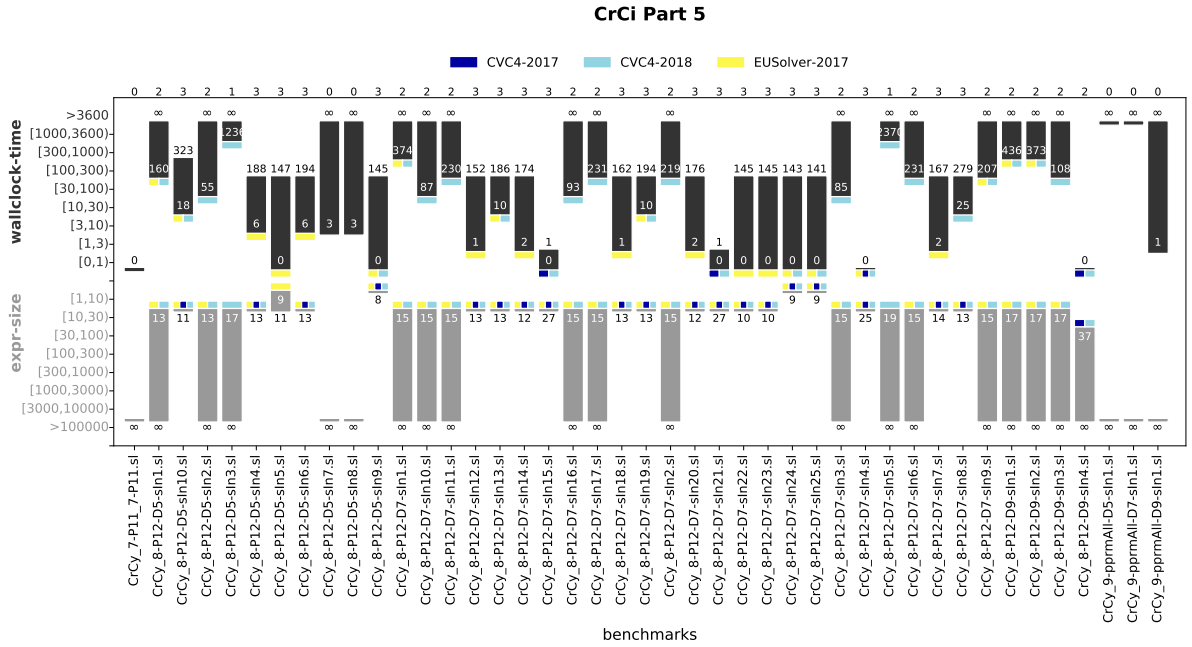


Figure 9: Evaluation of crypto circuits category of the General track (Part 5).

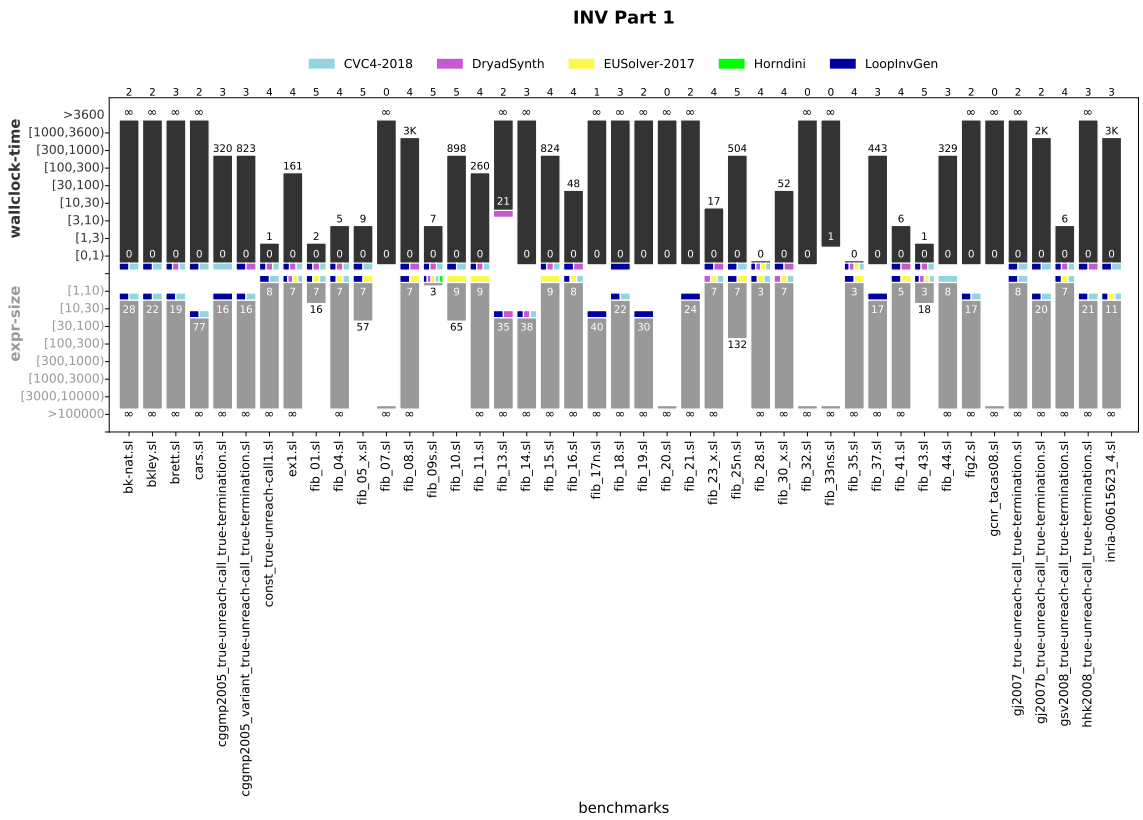


Figure 10: Evaluation of Invariant track benchmarks (Part 1).

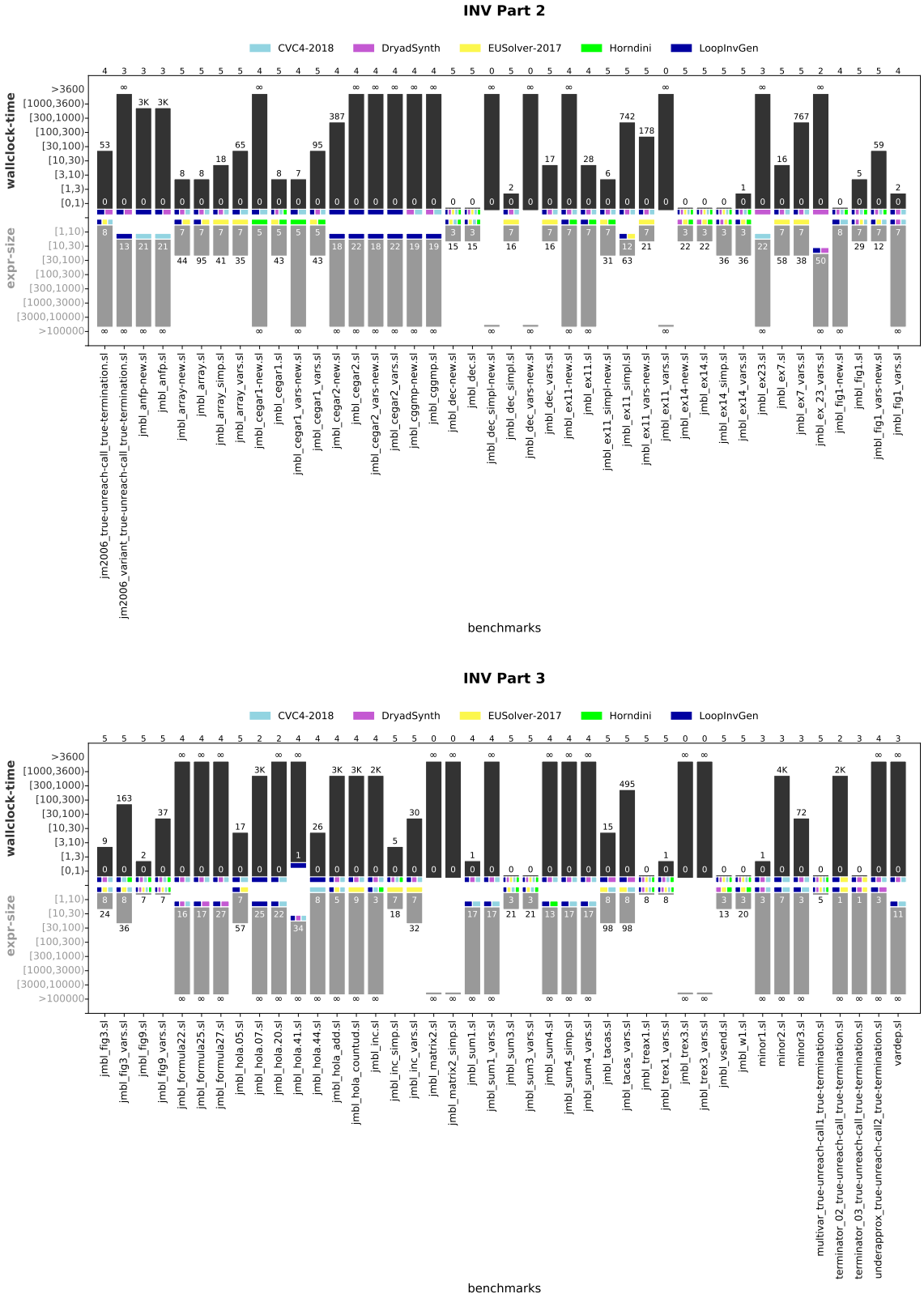


Figure 11: Evaluation of Invariant track benchmarks (Parts 2 & 3).

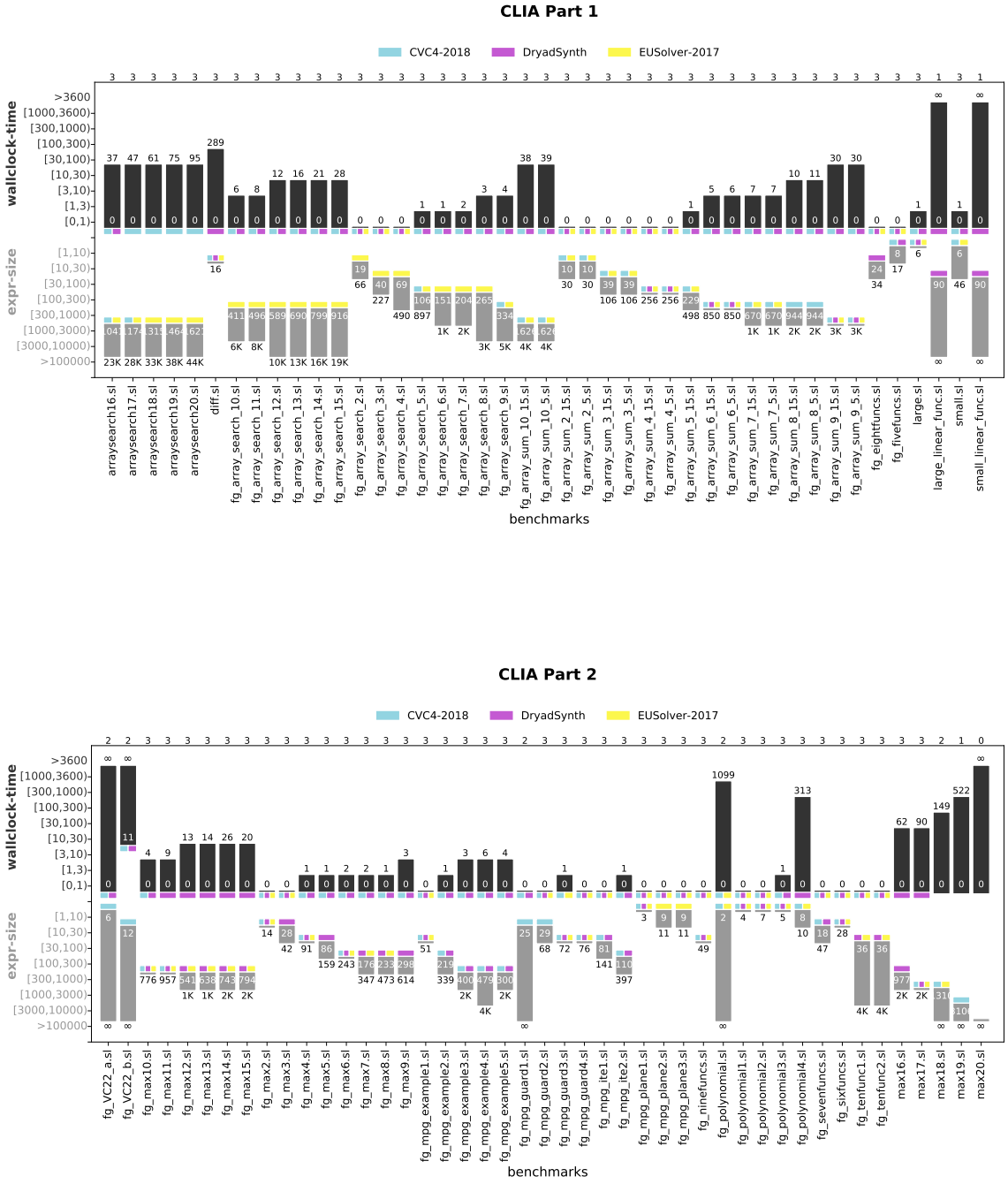
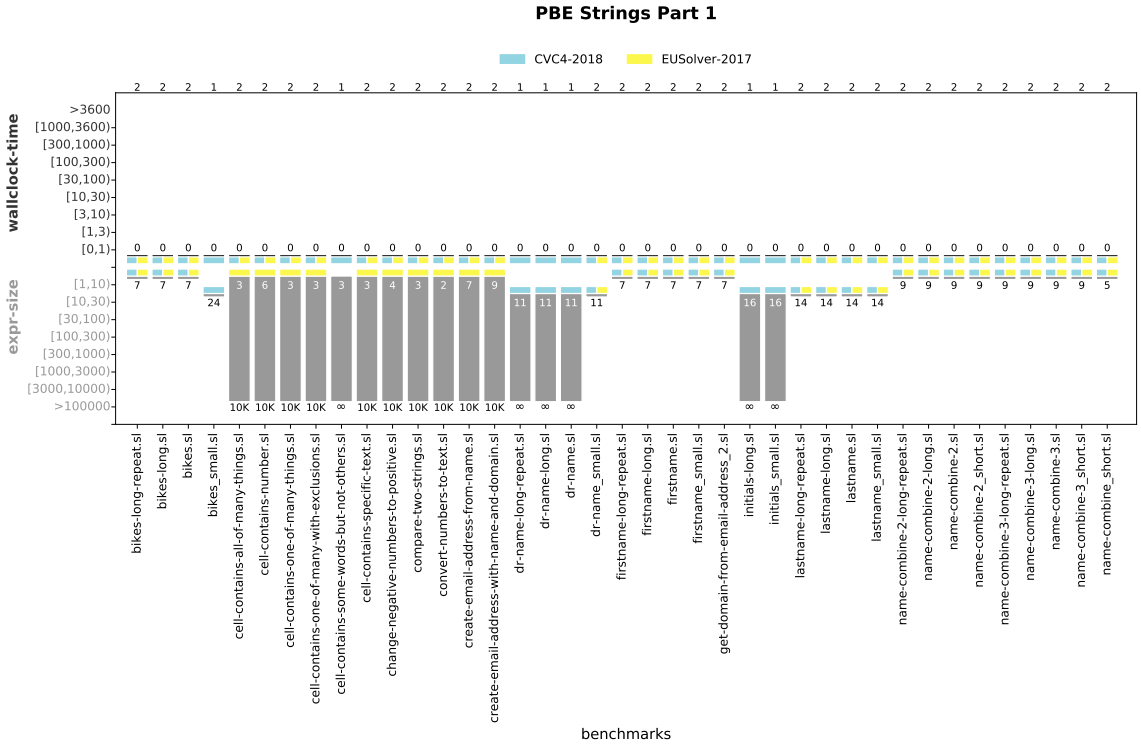


Figure 12: Evaluation of CLIA track benchmarks.



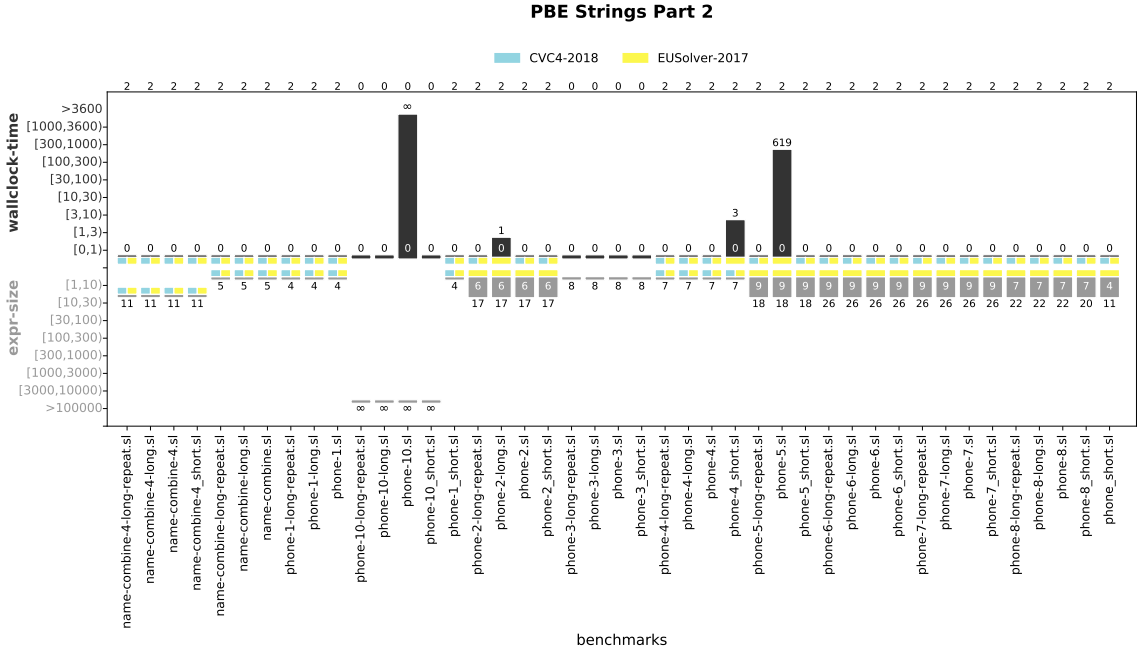


Figure 13: Evaluation of PBE Strings track benchmarks.