TSE Final Project Report (Team APR)

GYMPaG – A Patch Generator, Ranker and Classifier For Java

1. George Cherian (UNI: gc2920)

2. Mavis Athene Chen (UNI: mu2288)

3. Yin Zhang (UNI: yz4053)

1. Synopsis

For our final project we created an APR tool called GYMPaG which generated a list of patches ranked by usefulness and classified based on correct vs plausible

We hope that this tool can be used by software developers to help find patches for programs which are ranked from best to worst and classified as correct to present users with the best possible correct patch without the developers having to dig through a list of patches and choosing the best one, we also aimed to create a tool which could be used without much setup and tweaking of code which is needed in almost all of the existing tools.

Our tool combines multiple recently developed approaches as well as a new ranking mechanism followed by the classification of patches to give users a easy to use patch which saves developers time in going through incorrect patches.

This tool is fairly efficient in terms of time taken to run although it does need a fair amount of memory to run all three of our chosen approaches in parallel, we believe they generate a good set of patches which after ranking and classification can be easily used to correct a program.

The tool can be easily accessed on our GitHub (https://github.com/TeamAPR/tool) for use and replication.

Our approach is demonstrated in Fig 1. (In the appendix)

The input to the tool is the Defects4J Charts Patch numbers to run for, we have configured it to take this in as a default set, but this can be configured to take in any other faulty set of code as well by simply changing the configuration files for the base level code as needed.

Once the input is provided a bash script makes sure to compile most of the needed code and then calls the Java tool which has been created.

This Java tool acts as an orchestrator running all the sections of the tool and moving from one stage to the next upon completion.

The Java tool in Step #1 runs the GZoltar fault localization method on the passed in program, we chose GZoltar since it had good capabilities to find the faulty statements accurately and all the tools could be modified to accept it as an input. Once fault localization is completed the output is dumped into a JSON file containing the information needed by the APR tools.

Next in Step #2 the tool calls the three patch generation methods to execute in parallel. All three of them first read the input from the JSON file created by the fault localization method run in Step 1 and then start working on the generation of a patch. For this step we leveraged code from Cardumen(https://github.com/SpoonLabs/astor/tree/master/src-cardumen), jGenProg (https://github.com/SpoonLabs/astor/tree/master/src-jgenprog)and ARJA(https://github.com/yyxhdy/arja). Our initial plan was to use TBar (https://github.com/SerVal-DTF/TBar) however when we tried executing TBar on The first defect of Defects4J (Chart #1) it took 12 hours to run and generate a patch as opposed to the approximately 50 minutes the other 3 tools combined took to execute for the same patch.

We did however leverage some of the shell utilities and other code from TBar to build our fault localization and orchestrator.

Step #2 as mentioned earlier runs all three patch-generation approaches in parallel, We modified all three approaches in Step #3 to generate a standard JSON file which is dumped into a folder with the following structure (BugName/Approach/output.json). The orchestrator waits until all 3 approaches are done executing and have created the output JSON if successful.

Once this is done then the orchestrator calls the ranking code in Step #4. In this step, we compile all the plausible patches generated by the three approaches and rank them according to our own ranking mechanism devised through heuristics from literature. Our ranking system is based on the number of mutations and the types of mutations. After all the patches have been ranked, we dump the results into a folder with the following structure (BugName/BugName_ranked.json).

The output of Step #4 is passed to the classifier in Step #5. The patch classifier will identify the patch correctness based on the approach described in the paper Identifying patch correctness in test-based program repair [REF NUM]. First, the classifier takes a patch ranked by the ranking system as an input, passes it to the patch distance measurer which calculates a vector of the sequence distances between its executions on the original and patched program. Then the calculated results will be sent to the patch classifier to determine whether the patch is correct or plausible and generates a classification result into the ranked json file for each patch. Approach overview of the patch classifier is demonstrated in Fig 2. (In the appendix)

Finally, the tool generates an output file which contains ranking metrics such as NumOfMutation, NumOfDeleteMutations, NumOfInsertMutations, NumOfReplaceMutations, ranking number, patch differences, tool name and correctness classification result.

2. Research Questions

- **RQ1: How many unique bugs does GYMPaG generate patches for?**

We ran GYMPaG and 7 other tools on the 26 bugs which were a part of the Defects4J benchmark and came up with the results listed in Table 1 in the appendix. The last row in bold shows the results from GYMPaG.

We have also shown the details of running the other tools from our previous experiment in tables 5-8.

As demonstrated by the results GYMPaG is able to fix 14 out of the 26 bugs in the Defects4J benchmark which was more than the total number of bugs fixed by any other approach that we ran on our system, Based on Table 4 we can also see that the changes made to the tools(Henceforth named ARJA-M , Cardumen-M and JGenProg-M) to all take in the input from GZoltar helped the tools solve more bugs than their generic unmodified counterparts.

With these results we can be sure that the three methods that we chose to solve the bugs are effective in generating patches for the code passed to them and that the usage of GZoltar as a Fault localization methodology is effective in helping the underlying approaches generate a patch for a bug,

We know that our approach fixes 14/26 bugs in Defects4J Charts library which is more than 50% of the bugs being passed through it which is an improvement on existing systems.

- **RQ2: What effect does the removal of approaches in Step 2 have on the results of the system?**

We conducted an ablation study to find out which combination of approaches had the maximum ability to solve the largest number of bugs as well as trying to identify the time taken for each set of approaches to solve the bugs and what was the most efficient set of approaches that could be employed.

The results of this experiment are in Table 3 in the appendix, As the table demonstrates removing of any one mode (Especially ARJA-M) might lead to an increase in the time efficiency of the tool, but this comes at the cost of the capability of fixing a larger number of bugs.

Hence, we conclude that the selection of approaches for Step 2 is ideal in solving the largest set of bugs and removal of any approach has a negative impact on the number of bugs the tool can patch.

- **RQ3: How much time does the system take on average to produce a fix for a bug.**

Since we ran GYMPaG on the 26 bugs in Defects 4J we were able to come up with a rough estimate on the amount of time the tool would take to fix a bug on average, Given more time we could have conducted more runs to determine a more accurate number but we did run the ablation study as shown by Table 3 in the appendix which did give us an idea that the time of execution does not vary too wildly across runs.

We did see that ARJA-M runs faster than the base version of ARJA and this is probably due to the modifications we made to ARJA to allow fault localization to happen outside of ARJA and some optimizations we made within the code for ARJA which did help speed up the process of execution for ARJA and the entire system as a result.

GYMPaG takes around 36 hours to fix 276 bugs and with this we can see that the average time that GYMPaG takes per bug fix is around 1.4 hours. While this time is definitely more than approaches like Cardumen and JMutRepair it is also much better than ARJA and NOPOL and we believe it is a good ratio of time to bug fixes.

If we divide the number of bugs for which patches were found by the time taken then as shown by Table 3 the only approach which has a better ratio than GYMPaG's 2.5 is JGenProg's 1.6

- **RQ3: How much time does the ranking mechanism save in terms of providing the user the best possible patch?**

We first need to define what the best possible patch is for the user. Our approach to devising our own ranking mechanism is to first examine if any of these approaches have an existing ranking approach. ARJA does not implement a ranking system but it does have a fitness evaluation metric for each patch which factors in the patch size (number of edits) and the weighted failure weight. Essentially, their metric is achieved by minimizing both the parameters. Although there is no explicit ranking present, we can employ their heuristic of minimizing patch size. In both Cardumen and jGenProg, the default ranking is in chronological order of when the patch is produced. They provide an extension point ("SolutionVariantSortCriterion") for implementing custom sort for patches or post-processing. Although not implemented in the original codebase, patched minimization was mentioned by the authors as one potential post-processing step. Once again, we see another heuristic of minimizing patch size. In addition, we also looked into TBar and SimFix since there is more documentation on the effect of the types of mutations/edits on patch sorting. Both TBar and SimFix agree on prioritizing types of mutations in this manner: replacement > insertion > deletion. Furthermore, they both agree on minimizing the size of patches as well. TBar and SimFix also mentioned metrics like consistent modifications (ie. same variable) and minimizing distances between buggy code and donor code. However, we were unable to extract information from all three approaches relevant to these two metrics and therefore not implemented in our ranking mechanism. Hence, our definition of the best possible patch for the user, in terms of ranking, is defined to have small mutations with the types of mutations in the order presented above.

From the search through the three approaches and from literature, we were able to devise our own nested ranking mechanism as such (earlier rule indicates higher priority):

- Patches with fewer number of mutations/edits are ranked higher.
- Patches with replacement mutations are ranked higher.
- Patches with insertion mutations are ranked higher.
- Patches with deletion mutations are ranked last.

Table 2 in the Appendix details the metrics of the ranked patches for Chart 7 of the Defects4J by GYMPaG. We can conclude that our method does produce the correct ranking given the metric by producing patches with the smallest number of mutations first before ranking by type of mutation (following priority mentioned above).

Finally, the ranking mechanism we have saves the user almost all their time spent parsing through for the best possible patch. Users would have to parse through the "Patch Difference", which is the typical output format of the three approaches, a convoluted string of information about the modified file, the original code and the patch. Our approach first provides a heuristic for finding the best possible patch and proceeds to rank it so users do not have to search for the best possible patch. Even if we provide the users with the metrics, this is comparing between ranking items manually given four nested sort criteria and having computers perform the same task. The number of patches, if any, produced by GYMPaG for Charts in the Defects4J dataset ranges from 1 to 282 patches. With an average of X patches per bug. Going through all the patches manually for all the bugs is clearly not as efficient as having a ranking mechanism which the user could then decide to take the top "x" patches they want.

- **RQ5: How many of the patches generated can be counted as correct and have been correctly classified as such?**

YIN

…………………………………….

- **RQ6: How much time does it take for a human user to go through incorrect patches?**

YIN

……………………………

3. Deliverable

All of the code we created is in the master branch at https://github.com/TeamAPR/tool we have a readme file which details the pre-requisites and needed instructions to run the code. The code has only been tested on a mac machine so the shell file and scripts may need to be changed to work on a Windows machine and may need some slight tweaking for a Linux machine. We ran the code on an 8 core M1 Mac with 16 GB of ram and a 1 TB SSD.

4. Reuse

As mentioned earlier we have leveraged code from the following sources to build GYMPaG:

- https://github.com/SpoonLabs/astor/tree/master/src-cardumen
- https://github.com/SpoonLabs/astor/tree/master/src-jgenprog
- https://github.com/yyxhdy/arja
- https://github.com/SerVal-DTF/TBar
- https://github.com/Ultimanecat/DefectRepairing
- https://github.com/rjust/defects4j

While using the below list of papers to help guide our design principles:

- René Just et al., "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14), no. https://doi.org/10.1145/2610384.2628055 , July 2014.
- Jiajun Jiang et al., "Shaping program repair space with existing patches and similar code.," Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, no. https://doi.org/10.1145/3293882.3330559 , July 2019.
- Wolfgang Banzhaf et. al., "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," IEEE Transactions on Software Engineering,, no. https://ieeexplore.ieee.org/abstract/document/8485732 , Oct 2020..
- Martin Monperrus et. al., "“Astor: Exploring the Design Space of Generate-and-Validate Program Repair beyond GenProg”.," Journal of Systems and Software, Elsevier, Jan 2019. https://www.sciencedirect.com/science/article/pii/S0164121219300159
- Yingfei Xiong et al, "“Identifying patch correctness in test-based program repair”," Proceedings of ICSE. 2018., May 2018. https://dl.acm.org/doi/10.1145/3180155.3180182
- C. Le Goues et. al., ""GenProg: A Generic Method for Automatic Software Repair,"," EEE Transactions on Software Engineering, vol. 38, October 2011.

5. Self-Evaluation

1. **George Cherian (UNI: gc2920)**

I worked on Step #1, 2 and 3 of the projects which was the creation of a centralized fault localization methodology, running of all tools in parallel and the creation of a standardized output format. I also took up the task of creating an orchestration engine to allow all the different steps of the tools to be able to execute as one large entity rather than three separate projects.

For me, the hardest part of the project was modifying the existing code bases to first get them to work and then secondly to get them to work well with each other taking in a common input and spitting out a common format. We also had a tough time coming up with metrics to use for the ranking mechanism which could actually be derived from all three tools because of the differences in how the tools worked there were issues pulling a set of common metrics and so we had to scale down the metrics we were looking to rank all three tools. Finally, the execution time of these tools to run over a set of bugs is so long that any small mistake in the code especially at the output stage would mean hours of wasted effort.

2. **Mavis Athene Chen (UNI: mu2288)**

I worked mainly on Step #4 and contributed to Step #3 as well, which was the whole workflow of the ranking mechanism involving searching through literature, looking for current ranking mechanisms in each approach and its respective codebase. In addition, I also worked on integrating the ranking mechanism to the full codebase.

The challenging parts were deciding what metrics to use to commonly rank the three approaches, since not all the approaches employ a ranking technique to begin with, and the heuristics of ranking the patches. I learned from looking thoroughly through each approach and their papers, what kind of information I could use to devise our own ranking measurements. In addition, looking through the codebase and figuring out where the approximate ranking metrics are also took some time since each approach has different configurations as well. Overall, I learned a lot more about the actual implementation of these APR approaches since I mainly focused on the theoretical side in the midterm paper and learned how to put the findings from literature into real practice like “translating” theory to code.

3. **Yin Zhang (UNI: yz4053)**

I was responsible for Step #5 of the project, which is to apply the classification methodology described in literature to identify the patch correctness vs. plausibility. Based on the PATCH-SIM theory proposed in [REF], the patch correctness identifier takes several steps to generate a final classification result. Including calculating patch distances vector through the distance measurer, normalizing the results, and then passing the results into the classifier to get our final classification output. Besides that, I also manually go through 20 buggy programs from the Defects4j dataset in order to provide human judges on how much time our approach has saved for an actual user.

There are several challenges I need to overcome when applying the patch classifier to the project:

1. When deciding which tool to use for the classification, I found that there are limited researches on identifying patch correctness. I tried to explore several

different approaches that are described in literature, the available options are either too complicated for us to adopt, not suited for test-based program repair, or there's no existing open source codebase for me to test their method. [examples] Thus, under the conditions that we only have a limited amount of time and resources for the project, I came to the conclusion that the method used in [REF] is most suitable for our test-based program repair's patches.

2. The patch classifier proposed in the paper has existing code implementation, but is concatenated with another method called test classifier which is an implementation of their other theory called "TEST-SIM". The test classifier checks for passing or failing tests and then passes the result to the patch classifier. But in our case, all patches are plausible since they've passed all tests in the test suite during previous steps. So I have to recode the method described in the paper to use only the patch classifier for our tool.

What I have learned from contributing to this project is how a test-based program repair works in a more detailed way. From the literature, I learned a new technique to classify patches in a heuristic way without knowing the entire buggy code repair workflow. Meanwhile, I also learned a lot from converting "theory" into actual code that would work for the entire tool. It is a rewarding experience for me to first read through papers, then checkout the open source code and see how the authors actually implemented their proposed methods. I also learned how to adjust the existing tool in order for it to work in a desired way. Besides, this project gives me an impression of a real world research process, even the time period is relatively short. And I am sure I will continue the learning process of automatic program repair and related topics after the end of this course.

6. Appendix

Figures:

Fig 1. GYMPaG Approach Diagram

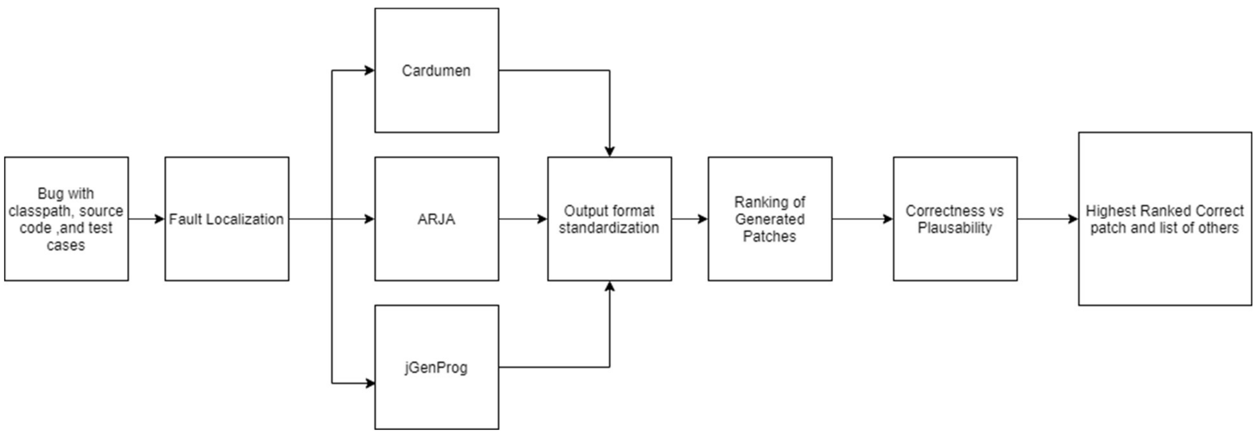
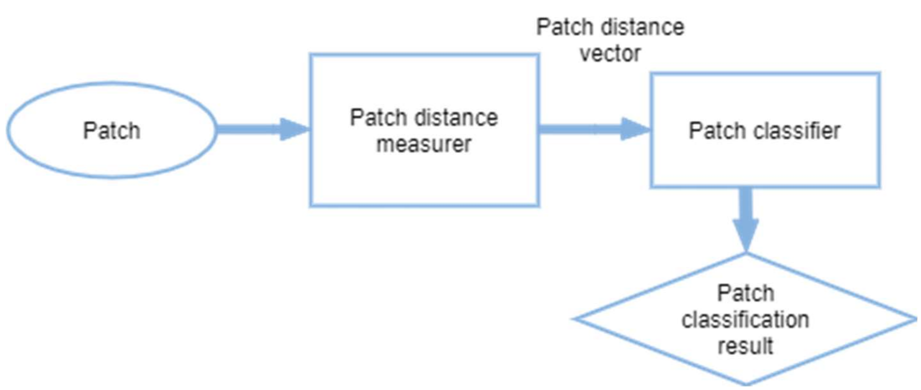


Fig 2. Patch Classifier Approach Overview



Tables

Table 1 Experiment Results Detailed

APR Tool	# of bugs checked	# of bugs Fixed	Time taken per correct patch	Time taken total
ARJA	26	5	12.6 Hrs	63.66 Hrs
Cardumen	26	5	3.4 Hrs	17 Hrs
JKali	26	2	8.25 Hrs	16.5 Hrs
JMutRepair	26	4	3.12 Hrs	12.5 Hrs
jGenProg	26	5	1.6 Hrs	7.9 Hrs
SimFix	26	0	x	73.7 Hrs
Nopol	26	0	x	16.1 Hrs
GYMPaG	**26**	**14**	2.5 Hrs	**36.3 Hrs**

Table 2 Results of ranking Chart 7 from Defects4J.

Rank	# of Mutations	# of Replacement Mutations	# of Insert Mutations	# of Deletion Mutations	Tool
1	1	0	1	0	ARJA
2	1	0	1	0	ARJA
3	1	0	0	1	ARJA
4	2	0	2	0	ARJA
5	2	0	2	0	ARJA
6	2	0	2	0	ARJA
7	2	0	2	0	ARJA
8	2	0	2	0	ARJA
9	2	0	2	0	ARJA

10	2	0	2	0	ARJA
11	2	0	2	0	ARJA
12	2	0	2	0	ARJA
13	2	0	2	0	ARJA
14	2	0	2	0	ARJA
15	2	0	1	1	ARJA
16	2	0	1	1	ARJA
17	2	0	1	1	ARJA
18	2	0	1	1	ARJA
19	2	0	1	1	ARJA
20	2	0	1	1	ARJA
21	2	0	1	1	ARJA
22	2	0	1	1	ARJA
23	2	0	1	1	ARJA
24	2	0	1	1	ARJA
25	2	0	1	1	ARJA
26	2	0	1	1	ARJA
27	2	0	1	1	ARJA
28	2	0	0	2	ARJA

29	3	1	2	0	ARJA
30	3	0	3	0	ARJA
31	3	0	2	1	ARJA

Table 3 Experiment Results For Various Approaches (GYMPaG is ARJA-M + Cardumen -M + jGenProg-M)

APR Tool	# of bugs checked	# of bugs Fixed	Timeouts	Time taken total
ARJA - M	26	8	0	28.2 Hrs
Cardumen - M	26	6	0	19.5 Hrs
jGenProg - M	26	5	0	7 Hrs
ARJA – M + Cardumen - M	26	11	0	33.6 Hrs
ARJA – M + jGenProg - M	26	10	0	29.3 Hrs
Cardumen – M + jGenProg - M	26	9	0	20.5 Hrs
GYMPaG	**26**	**14**	**0**	**36.3 Hrs**

Table 4 Details of fixes provided by each approach for each bug by each approach.

Bug #	ARJA-M Patch	Cardumen-M Patch	jGenProg Patch
Chart-1	Yes	Yes	Yes
Chart-2			
Chart-3	Yes		
Chart-4			
Chart-5		Yes	
Chart-6		Yes	
Chart-7	Yes		
Chart-8			
Chart-9			
Chart-10			
Chart-11		Yes	
Chart-12	Yes		
Chart-13	Yes		Yes
Chart-14	Yes		
Chart-15			Yes
Chart-16			
Chart-17			
Chart-18		Yes	
Chart-19	Yes		

Bug #	ARJA-M Patch	Cardumen-M Patch	jGenProg Patch
Chart-20			
Chart-21			
Chart-22			
Chart-23			
Chart-24			
Chart-25	Yes	Yes	Yes
Chart-26			Yes

Table 5: Results of the Nopol Experiment with Defects4J Charts

Bug #	# of Statements Analyzed	# of Statements with Angelic values	Did Timeout?	Execution Time	Reason for Failure
Chart-1	40	0	No	126 s	No Angelic Value found
Chart-2	21	0	No	86 s	No Angelic Value found
Chart-3	20	1	No	91 s	No Synthesis
Chart-4	282	0	No	865 s	No Angelic Value found
Chart-5	9	2	No	52 s	No Synthesis
Chart-6	26	5	No	83 s	No Synthesis
Chart-7	14	0	No	56 s	No Angelic Value found
Chart-8	0	0	No	59 s	No Angelic Value found
Chart-9	28	7	No	75 s	No Synthesis
Chart-10	0	0	No	36 s	No Angelic Value found
Chart-11	6	0	No	43 s	No Angelic Value found
Chart-12	14	0	No	67 s	No Angelic Value found
Chart-13	41	2	No	86 s	No Synthesis
Chart-14	30	0	No	120 s	No Angelic Value found
Chart-15	112	0	No	256 s	No Angelic Value found
Chart-16	4	0	No	44 s	No Angelic Value found
Chart-17	13	1	No	54 s	No Synthesis
Chart-18	11	0	No	55 s	No Angelic Value found
Chart-19	51	0	No	164 s	No Angelic Value found
Chart-20	0	0	No	34 s	No Angelic Value found
Chart-21	17	1	No	63 s	No Synthesis
Chart-22	21	0	No	59 s	No Angelic Value found
Chart-23	74	0	No	1035 s	No Angelic Value found
Chart-24	0	0	No	35 s	No Angelic Value found
Chart-25	502	12	No	47409 s	No Synthesis

Bug #	# of Statements Analyzed	# of Statements with Angelic values	Did Timeout?	Execution Time	Reason for Failure
Chart-26	X	X	Yes	> 600 Minutes	Timeout

Table 6 SimFix Results

Bug #	Execution Time	Did Timeout?	Reason for Failure
Chart-1	215 min	No	
Chart-2	327 min	Yes	Timeout
Chart-3	70 min	No	
Chart-4	173 min	No	
Chart-5	315 min	Yes	Timeout
Chart-6	152 min	No	
Chart-7	10 min	No	
Chart-8	11 min	No	
Chart-9	12 min	No	
Chart-10	5 min	No	
Chart-11	39 min	No	
Chart-12	72 min	No	
Chart-13	53 min	No	
Chart-14	38 min	No	
Chart-15	297 min	No	
Chart-16	370 min	Yes	Timeout
Chart-17	330 min	Yes	Timeout
Chart-18	217 min	No	
Chart-19	268 min	No	
Chart-20	307 min	Yes	Timeout
Chart-21	312 min	Yes	Timeout
Chart-22	417 min	Yes	Timeout
Chart-23	173 min	No	
Chart-24	3 min	No	
Chart-25	82 min	No	
Chart-26	73 min	No	

Table 7: JMutRepair Results

Bug #	Execution Time	Patch Found?	Patch found in paper?	Reason for Failure
Chart-1	2 min	Yes	Yes	
Chart-2	11 min			
Chart-3	13 min			
Chart-4	50 min			
Chart-5	12 min			
Chart-6	3 min			
Chart-7	8 min	Yes	Yes	
Chart-8	3 min			
Chart-9	3 min			
Chart-10	2 min			
Chart-11	2 min			
Chart-12	2 min			
Chart-13	3 min			
Chart-14	6 min			
Chart-15	9 min			
Chart-16	62 min			Initial run of test suite fails
Chart-17	35 min			
Chart-18	32 min			
Chart-19	25 min			Initial run of test suite fails
Chart-20	79 min			Initial run of test suite fails
Chart-21	43 min			Initial run of test suite fails
Chart-22	59 min			
Chart-23	57 min			Initial run of test suite fails
Chart-24	61 min			
Chart-25	50 min		Yes	Initial run of test suite fails
Chart-26	111 min		Yes	Initial run of test suite fails

Table 8: JKali Results

Bug #	Execution Time	Patch Found?	Patch found in paper?	Reason for Failure
Chart-1	12 min	Yes	Yes	
Chart-2	12 min			
Chart-3	11 min			
Chart-4	75 min			
Chart-5	69 min			
Chart-6	2 min			
Chart-7	6 min	Yes	Yes	
Chart-8	3 min			
Chart-9	0 min			
Chart-10	15 min			
Chart-11	27 min			
Chart-12	19 min			Initial run of test suite fails
Chart-13	43 min		Yes	
Chart-14	52 min			Initial run of test suite fails
Chart-15	62 min		Yes	Initial run of test suite fails
Chart-16	90 min			Initial run of test suite fails
Chart-17	53 min			Initial run of test suite fails
Chart-18	42 min			Initial run of test suite fails
Chart-19	30 min			Initial run of test suite fails
Chart-20	82 min			
Chart-21	50 min			
Chart-22	35 min			
Chart-23	33 min			Initial run of test suite fails
Chart-24	20 min			
Chart-25	28 min		Yes	Initial run of test suite fails
Chart-26	105 min		Yes	Initial run of test suite fails