ABC: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics (Artifact Document)

1 Introduction

The accompanying paper describes our static analysis framework for Java, `Tai-e`. For anonymity, in our submission paper, we changed the framework's name to "`ABC`". Since the reviewing process of artifact evaluation is single-blind, we revert the frameworks' name back to "`Tai-e`" in this artifact. Before diving into this artifact, we want to point out that the best artifact of `Tai-e` is its GitHub repository (`https://github.com/pascal-lab/Tai-e`), where we actively develop and maintain `Tai-e`. Currently, we have released `Tai-e` for a year, and it has obtained 600+ stars.

This artifact is provided to reproduce the results of RQ4 in Section 6 of our companion paper, i.e., the data in:

- Table 1 (for pointer analysis)
- Table 2 (for data flow analysis)

Note that the data in Table 1 and Table 2 are the *average* numbers summarized from detailed evaluation results for all benchmarks under each analysis tool. Actually, **when we submitted our paper, we also submitted a separate supplementary material which includes all the detailed evaluation results for RQ4 of the paper. For your convenience, we copied those detailed results in the supplementary material to Table 1.0.1 and Table 1.0.2, which contain the experimental results for each benchmark under pointer analysis and data flow analysis, respectively. This artifact supports reproducing not only the summarized results but also the detailed results.**

To thoroughly evaluate the effectiveness of `Tai-e`, we compared it with other five analysis tools, `Qilin`, `Doop`, `Soot`, `WALA` and `SpotBugs`, which make our evaluation complex. So for your convenience, we compose easy-to-use scripts with different options to easily reproduce the data in our paper and to run any analysis that you are particularly interested in.

Research questions RQ1 - RQ3 in Section 6 involve in a survey as we explained in the paper. However, because the survey is not conducted in English, and the participants did not give us permission to share their answers in the survey (including for review), they cannot be included in the artifact.

In this document, we will first introduce how to setup our artifact and run basic tests in Section 2, and then explain how to use the artifact with different options and reproduce experimental results in Section 3.

2 Getting Started

2.1 Basic Requirements

- **Machine.** In our paper, all experiments were carried out on a machine with an Intel Xeon 2.2GHz processor and 128GB of memory. **Since pointer analysis can be very memory-consuming, using a machine with a smaller RAM may cause some analyses to be more slower or even to the point of being unscalable.** If the user's memory resource is limited, we recommend you concentrate on cheap analyses, i.e., data flow anlaysis and context-insensitive pointer analysis (`ci` for short), and avoid heavy context-sensitive analyses (like `2-call`). **Also note that the results concerning execution time of analysis may vary in different running environments.**
- **Docker.** To ease the setup of our artifact and make it cross-platform, we packed it as a Docker image with the experimental environment completely setup (e.g., JDK 17, Python 3, and Souffle 1.5.1 have been installed).

2.2 Experimental Setup

Firstly, please install Docker on your system (users who have already installed Docker can skip this step). If you are a Mac or Windows user, please follow the instructions on https://docs.docker.com/get-docker/ to install the Docker Desktop. If you are a Linux user, we recommend you install the Docker engine by following instructions on https://docs.docker.com/engine/install/.

Table 1.0.1: Pointer analysis results in terms of recall and analysis time. “Total”, “Recall” and “R/T ” mean the real results collected by running dynamic analysis, the real results resolved by a pointer analysis, and Recall/Total (recall rate), respectively. “#varpt”, “#reach” and “#edges” mean the total number of points-to relations for all variables, reachable methods and call graph edges respectively. “2-obj” and “2-call” represent two widely adopted context-sensitive pointer analysis algorithms (2 levels of object sensitivity and call-site sensitivity). As `Soot` does not support context sensitivity, its results for “2-obj” and “2-call” are all not application (“N/A”). “–” means the analysis cannot finish running within time budget or run out of memory.

Program	Tool	Reachable methods			Call graph edges			Context insensitivity				2-obj	2-call
		Recall	Total	R/T	Recall	Total	R/T	Time (s)	#varpt	#reach	#edges	Time (s)	Time (s)
findbugs	`Tai-e`	5,808	5,857	99.2%	13,899	14,075	98.7%	15.2	7,355,719	17,353	107,339	1419.1	2114.5
	`Qilin`	5,663		96.7%	13,323		94.7%	23.5	7,360,275	16,988	106,998	2033.0	4214.6
	`Doop`	4,835		82.6%	11,121		79.0%	46.0	5,239,275	13,660	83,699	638.0	5292.0
	`Soot`	5,379		91.8%	12,548		89.2%	44.4	9,959,179	15,425	116,642	N/A	N/A
soot	`Tai-e`	4,288	4,372	98.1%	16,233	16,452	98.7%	107.7	84,628,666	32,918	415,728	–	–
	`Qilin`	4,225		96.6%	16,120		98.0%	173.9	82,179,258	32,780	420,243	–	–
	`Doop`	4,253		97.3%	16,156		98.2%	454.0	74,564,948	32,600	413,411	–	–
	`Soot`	3,175		72.6%	10,446		63.5%	47.0	19,829,796	21,072	264,506	N/A	N/A
gruntspud	`Tai-e`	14,360	14,543	98.7%	41,426	42,099	98.4%	69.0	48,179,683	39,800	274,872	–	–
	`Qilin`	14,279		98.2%	41,261		98.0%	113.9	50,073,466	39,247	273,794	–	–
	`Doop`	9,508		65.4%	25,363		60.2%	159.0	22,309,125	25,077	154,763	–	–
	`Soot`	12,835		88.3%	35,683		84.8%	111.4	54,498,294	31,968	265,090	N/A	N/A
columba	`Tai-e`	6,673	6,757	98.8%	14,368	14,689	97.8%	135.8	107,856,623	56,787	425,149	–	–
	`Qilin`	6,617		97.9%	14,260		97.1%	170.0	101,298,328	56,726	416,791	–	–
	`Doop`	4,983		73.7%	9,798		66.7%	616.0	70,315,191	42,665	286,990	–	–
	`Soot`	5,710		84.5%	11,635		79.2%	183.9	131,367,643	49,081	433,320	N/A	N/A
antlr	`Tai-e`	2,697	2,786	96.8%	10,086	10,341	97.5%	9.0	2,003,409	8,674	59,190	37.1	654.0
	`Qilin`	2,568		92.2%	9,531		92.2%	11.5	2,066,405	8,363	58,486	71.0	2009.9
	`Doop`	2,510		90.1%	9,401		90.9%	31.0	2,385,993	8,250	57,560	282.0	1472.8
	`Soot`	2,407		86.4%	8,774		84.8%	13.4	2,291,874	7,491	63,279	N/A	N/A
bloat	`Tai-e`	3,339	3,421	97.6%	11,794	12,029	98.0%	11.7	3,270,010	9,936	69,219	452.3	1107.0
	`Qilin`	3,292		96.2%	11,722		97.4%	13.1	3,363,502	9,631	68,922	1253.8	2691.2
	`Doop`	3,231		94.4%	11,297		93.9%	26.0	3,393,681	9,509	67,604	1394.0	–
	`Soot`	3,122		91.3%	11,592		96.4%	15.9	3,604,082	8,902	75,226	N/A	N/A
xalan	`Tai-e`	3,826	3,966	96.5%	9,254	9,614	96.3%	9.8	2,805,148	12,942	72,661	1854.0	1221.5
	`Qilin`	3,650		92.0%	8,571		89.2%	16.5	2,697,455	12,526	70,612	869.3	2546.6
	`Doop`	3,132		79.0%	6,999		72.8%	28.0	1,681,360	9,956	53,278	439.0	1359.4
	`Soot`	3,470		87.5%	8,082		84.1%	22.6	4,030,969	11,479	77,437	N/A	N/A
eclipse	`Tai-e`	7,080	8,093	87.5%	18,050	20,916	86.3%	26.8	22,633,879	23,920	184,294	–	–
	`Qilin`	6,936		85.7%	17,476		83.6%	48.1	24,230,566	23,550	184,290	–	–
	`Doop`	3,378		41.7%	6,860		32.8%	21.0	2,044,456	9,764	56,595	146.0	–
	`Soot`	2,927		36.2%	5,266		25.2%	17.9	2,091,670	8,153	55,881	N/A	N/A
hsqldb	`Tai-e`	2,608	2,733	95.4%	5,856	6,295	93.0%	11.7	1,838,669	11,588	64,393	–	654.6
	`Qilin`	1,707		62.5%	2,996		47.6%	8.8	1,016,624	7,570	42,896	36.2	968.3
	`Doop`	1,626		59.5%	2,841		45.1%	31.0	839,150	6,945	37,414	70.0	2477.4
	`Soot`	2,356		86.2%	5,040		80.1%	18.0	6,341,781	10,195	73,532	N/A	N/A
jython	`Tai-e`	4,243	4,835	87.8%	11,639	35,970	32.4%	17.0	10,885,453	13,076	121,603	–	–
	`Qilin`	4,056		83.9%	10,847		30.2%	25.7	10,101,462	12,650	119,872	–	–
	`Doop`	3,985		82.4%	10,717		29.8%	78.0	11,267,651	12,507	118,706	–	–
	`Soot`	3,879		80.2%	10,393		28.9%	28.3	12,062,834	11,881	132,383	N/A	N/A
luindex	`Tai-e`	2,179	2,266	96.2%	4,461	4,710	94.7%	8.8	829,314	7,899	41,520	18.4	515.8
	`Qilin`	2,052		90.6%	3,907		83.0%	9.1	889,969	7,590	40,828	30.4	1226.4
	`Doop`	1,973		87.1%	3,376		71.7%	14.0	921,998	7,456	39,743	46.0	469.2
	`Soot`	1,859		82.0%	3,689		78.3%	11.1	1,313,139	6,807	44,422	N/A	N/A
lusearch	`Tai-e`	1,736	1,826	95.1%	3,261	3,511	92.9%	9.2	981,532	8,574	44,839	18.9	509.8
	`Qilin`	1,606		88.0%	2,705		77.0%	9.5	1,048,650	8,263	44,166	47.7	1234.2
	`Doop`	1,545		84.6%	2,534		72.2%	15.0	1,099,851	8,136	43,072	49.0	1435.8
	`Soot`	1,440		78.9%	2,319		66.0%	10.8	1,409,394	7,103	46,628	N/A	N/A
pmd	`Tai-e`	3,908	4,025	97.1%	9,287	9,635	96.4%	10.1	2,848,296	13,311	73,890	39.1	1174.2
	`Qilin`	3,735		92.8%	8,613		89.4%	15.8	2,974,963	12,981	73,292	75.0	2136.0
	`Doop`	2,718		67.5%	5,737		59.5%	29.0	1,330,917	8,905	46,862	68.0	1605.9
	`Soot`	3,562		88.5%	8,196		85.1%	23.3	4,802,443	11,898	79,917	N/A	N/A
chart	`Tai-e`	5,197	5,326	97.6%	12,751	13,110	97.3%	17.4	5,755,512	16,455	87,938	184.9	856.7
	`Qilin`	5,024		94.3%	12,064		92.0%	21.8	6,103,008	16,109	87,848	283.6	2281.8
	`Doop`	4,691		88.1%	11,172		85.2%	42.0	4,679,309	13,643	73,106	216.0	3506.0
	`Soot`	4,483		84.2%	10,405		79.4%	31.5	9,994,315	13,802	96,449	N/A	N/A

Table 1.0.2: Live variable analysis results in terms of efficiency. “#Classes” and “#Methods” mean the number of classes and methods analyzed by each analysis for each program. “Time(s)” gives the analysis time in seconds. “#Methods/s” mean how many methods are analyzed per second.

Program	Tool	#Classes	#Methods	Time(s)	#Methods/s	Program	Tool	#Classes	#Methods	Time(s)	#Methods/s
findbugs	Soot	1,332	9,865	0.19	51,921	eclipse	Soot	379	3,006	0.11	27,327
	Tai-e	1,332	9,865	0.39	25,295		Tai-e	379	3,006	0.28	10,736
	WALA	400	2,201	0.37	5,949		WALA	159	977	0.25	3,908
	SpotBugs	2,274	16,611	5.09	3,263		SpotBugs	6,930	65,200	18.49	3,526
soot	Soot	3,085	31,928	0.43	74,251	hsqldb	Soot	23	255	0.02	12,750
	Tai-e	3,085	31,928	1.64	19,468		Tai-e	23	255	0.24	1,063
	WALA	2,575	25,021	8.98	2,786		WALA	10	54	0.05	1,080
	SpotBugs	3,837	36,673	10.75	3,411		SpotBugs	416	5,240	2.86	1,832
gruntspud	Soot	876	5,539	0.14	39,564	jython	Soot	586	5,907	0.18	32,817
	Tai-e	876	5,539	0.37	14,970		Tai-e	586	5,907	0.4	14,768
	WALA	832	4,902	0.69	7,104		WALA	226	2,075	0.44	4,716
	SpotBugs	1,460	9,534	3.13	3,046		SpotBugs	920	8,549	2.88	2,968
columba	Soot	664	4,843	0.12	40,358	luindex	Soot	181	1,331	0.06	22,183
	Tai-e	664	4,868	0.26	18,723		Tai-e	181	1,331	0.18	7,394
	WALA	4	6	0.01	600		WALA	72	354	0.19	1,863
	SpotBugs	13,783	118,429	28.42	4,167		SpotBugs	349	2,563	1.64	1,563
antlr	Soot	153	1,475	0.06	24,583	lusearch	Soot	220	1,602	0.07	22,886
	Tai-e	153	1,475	0.17	8,676		Tai-e	220	1,602	0.19	8,432
	WALA	102	771	0.2	3,855		WALA	139	732	0.19	3,853
	SpotBugs	228	2,483	2.13	1,166		SpotBugs	349	2,563	1.67	1,535
bloat	Soot	320	3,457	0.14	24,693	pmd	Soot	1,039	8,643	0.2	43,215
	Tai-e	320	3,457	0.32	10,803		Tai-e	1,039	8,643	0.67	12,900
	WALA	266	2,525	0.57	4,430		WALA	257	1,753	0.35	5,009
	SpotBugs	360	3,836	2.28	1,682		SpotBugs	553	3,850	1.77	2,175
xalan	Soot	24	258	0.03	8,600	chart	Soot	554	6,907	0.18	38,372
	Tai-e	24	258	0.24	1,075		Tai-e	554	6,907	0.53	13,032
	WALA	12	57	0.03	1,900		WALA	109	815	0.18	4,528
	SpotBugs	566	5,512	2.31	2,386		SpotBugs	515	6,093	2.13	2,861

With Docker installed, our artifact can be easily setup via following steps:

1. Load the docker image into your system (Note that for Mac or Windows users, you need to first start the Docker Desktop to enable command `docker`, and them type the following command in your terminal):

   ```
   $ docker load --input tai-e-artifact.tar.gz
   ```

   This step may take several minutes since the Docker image is large. (Note: if you are using `finch` rather than `docker`, please add the option `--all-platforms` into the command.)

2. Launch a container from the loaded image, where `tai-e:issta23` is the image name and `tai-e-artifact` is the container name:

   ```
   $ docker run --shm-size=<SIZE> --name tai-e-artifact -it tai-e:issta23
   ```

   The argument `<SIZE>` is to configure size of shared memory, and it is recommended to use the same size of your physical memory (e.g., `--shm-size=32g` for a machine with 32GB of memory).

   After you launched the container, you will enter an interactive shell of the container as shown in Figure 1.

   That’s it! Now you can start evaluating our artifact.

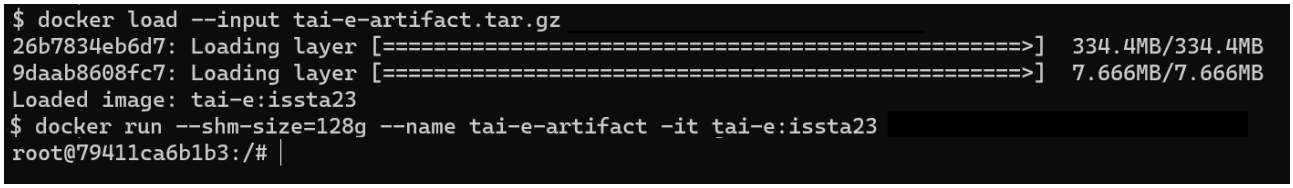

Figure 1: Load the Docker image and launch a Docker container.

To exit the interactive shell, use the command `exit`. If you want to re-enter the container after exiting it, use the following command to restart and enter the same container again:

```
$ docker restart tai-e-artifact
$ docker exec -it tai-e-artifact bash
```

2.3 Basic Testing

This artifact supports reproducing two sets of experiments, one for pointer analysis and the other for data flow analysis, with several different analysis tools. Below we introduce how to test whether these analysis tools have been successfully set up.

Testing Pointer Analysis We compare pointer analysis of four analysis tools, `Tai-e`, `Qilin`, `Doop`, and `Soot`. To test these tools, please first change your current directory to the pointer analysis folder:

```
$ cd /home/pointer
```

Then you can use the following command to run a pointer analysis using the four tools for benchmark `findbugs`:

```
$ ./pointer.py -all findbugs
```

This command will start the four tools to perform context-insensitive pointer analysis for `findbugs`, and may take several minutes to finish. Note that different tools have different output formats, and to ease the result checking, our scripts will collect analysis results of these tools and summarize them in a uniform format at the end of each execution as shown in Figure 2.

If you can observe the outputs for all four analysis tools like in Figure 2, then it means that the four tools for pointer analysis, i.e., `Tai-e`, `Qilin`, `Doop` and `Soot` have been successfully set up.

Testing Data Flow Analysis We compare data flow analysis of four analysis tools, `Soot`, `Tai-e`, `WALA` and `SpotBugs`. To test these tools, please first change your current directory to the data flow analysis folder:

```
$ cd /home/dataflow
```

Then you can use the following command to run a data flow analysis using the four tools for benchmark `findbugs`:

```
$ ./dataflow.py -all findbugs
```

This command will start the four tools to perform live variable analysis for `findbugs`, and may take several minutes to finish. Similar with pointer analysis, our scripts will collect data flow analysis results and statistics of these tools and summarize them in a uniform format at the end of each execution as shown in Figure 3.

If you can observe the outputs for all four analysis tools like in Figure 3, then it means that the four tools for data flow analysis, i.e., `Soot`, `Tai-e`, `WALA` and `SpotBugs` have been successfully set up.

3 Detailed Instructions

We introduce how to run pointer analysis and data flow analysis in Sections 3.1 and 3.2, respectively, and explain how to relate the analysis results to the data in Tables 1 and 2 of the accompanying paper in Section 3.3.

```
Averages of tai-e (with analysis: ci) for findbugs:
Recall-#reach: 99.2%
Recall-#edges: 98.7%
#Time (s): 12.9
#reach: 17353
#edges: 107339

Averages of qilin (with analysis: ci) for findbugs:
Recall-#reach: 96.7%
Recall-#edges: 94.7%
#Time (s): 18.8
#reach: 16988
#edges: 106998

Averages of doop (with analysis: ci) for findbugs:
Recall-#reach: 82.6%
Recall-#edges: 79.0%
#Time (s): 35.0
#reach: 13660
#edges: 83716

Averages of soot (with analysis: ci) for findbugs:
Recall-#reach: 91.8%
Recall-#edges: 89.2%
#Time (s): 34.0
#reach: 15425
#edges: 116642
root@55e828176909:/home/pointer#
```

Figure 2: Summarized pointer analysis results for `findbugs`.

```
Averages of soot for findbugs:
#Classes: 1332
#Methods: 9865
#Time (s): 0.15
#Methods/s: 65766

Averages of tai-e for findbugs:
#Classes: 1332
#Methods: 9865
#Time (s): 0.35
#Methods/s: 28185

Averages of wala for findbugs:
#Classes: 400
#Methods: 2201
#Time (s): 0.30
#Methods/s: 7336

Averages of spotbugs for findbugs:
#Classes: 2274
#Methods: 16611
#Time (s): 5.17
#Methods/s: 3212
root@55e828176909:/home/dataflow#
```

Figure 3: Summarized data flow analysis results for `findbugs`.

3.1 Running Pointer Analysis

To run the experiments, please run the Python script `pointer.py` under the directory `/home/pointer/` by using the following command (note that | means "or", and `[...]` means "optional"):

```
$ ./pointer.py  tai-e|qilin|doop|soot  [-pta=<ANALYSIS>]  <BENCHMARK>
```

The first argument (`tai-e|qilin|doop|soot`) specifies to run pointer analysis on which analysis tool.

`<ANALYSIS>` can be one of the following pointer analyses evaluated in our experiments:

`ci`, `2-obj`, `2-call`
(`ci` stands for context-insensitive pointer analysis)

`<BENCHMARK>` can be one of the following Java programs analyzed in our experiments:

`findbugs`, `soot`, `gruntspud`, `columba`, `antlr`, `bloat`, `xalan`,
`eclipse`, `hsqldb`, `jython`, `luindex`, `lusearch`, `pmd`, `chart`

For example, to use `2-obj` to analyze `antlr` by `Tai-e`, use command:

```
$ ./pointer.py tai-e -pta=2-obj antlr
```

Note that the argument `-pta` is optional, and when it is not specified, the script `pointer.py` runs `ci` analysis.

As mentioned in Section 2, different analysis tools have different output formats, and to ease the result check, our script will output analysis results in a uniform and prettified format at the end of each execution, so that you could conveniently compare them with Table 1.0.1.

For your convenience, the command argument `<BENCHMARK>` can be repeated for multiple times. For example, to run `2-obj` for `findbugs`, `antlr` and `lusearch` on `Doop`, use command:

```
$ ./pointer.py doop -pta=2-obj findbugs antlr lusearch
```

Then the three analysis executions will be performed in sequence.

Also, for your convenience, we provide `-all` to run pointer analysis collectively. When specifying `-all` for `<ANALYSIS>`, it represents all four analysis tools, and when using `-all` for `<BENCHMARK>`, it represents all 14 benchmarks. For example, to run `2-obj` analysis for benchmark `antlr` on four analysis tools, use command:

```
$ ./pointer.py -all -pta=2-obj antlr
```

To run `ci` analysis on `Tai-e` for all benchmarks, use command:

```
$ ./pointer.py tai-e -all
```

For saving your time, when option `-all` is used, the analyses executions that run beyond the time limit or run out of memory in our experiment will be skipped. To check whether an analysis execution will run out of time budget or memory on your machine, you can run this analysis individually using the command we introduced at the beginning of this Section.

When you specify to analyze more benchmarks (either by given multiple benchmark names or use `-all` option), the `pointer.py` script will output the average numbers for the results of the specified benchmarks at the end.

Running `2-call` Analysis on `Doop` For `2-call` analysis, as the latest version of `Doop` either runs out of memory (and killed by operating system) or exceeds time limit for all evaluated benchmarks, we instead adopt the results of running an old version of `Doop` for `2-call` in pointer analysis.

By default, our artifact uses Java 17 as default JDK to run virtually all analysis tools. However, old `Doop` requires Java 8, so please switch Java version before running old `Doop`. We have prepared a script `java-switcher.sh` in folder `/home` for switching Java version, and this can be done by command:

```
$ source /home/java-switcher.sh 8
```

Next, change current directory to `/home/pointer/doop-old`. Then you could run old `Doop` by using the following command:

```
$ ./doop-old.py <ANALYSIS> <BENCHMARK>
```

The arguments `<ANALYSIS>` and `<BENCHMARK>` are the same as above. For example, to run `2-call` for benchmark `luindex` on old `Doop`, use command (note that this could be time and space consuming):

```
$ ./doop-old.py 2-call luindex
```

If you want to run other analysis (after running old `Doop`), remember to switch Java version back to 17:

```
$ source /home/java-switcher.sh 17
```

3.2 Running Data Flow Analysis

The command line usage of data flow analysis in the artifact is similar with that of pointer analysis.

To run the experiments, please run the Python script `dataflow.py` under the directory `/home/dataflow/` by using the following command (note that `|` means "or", and `[...]` means "optional"):

```
$ ./dataflow.py  soot|tai-e|wala|spotbugs  <BENCHMARK>
```

The first argument (`soot|tai-e|wala|spotbugs`) specifies to run data flow analysis (specifically, live variable analysis) on which analysis tool.

Similar with pointer analysis, in data flow analysis, our script outputs uniform results at the end of each execution, so that you could easily compare them with Table 1.0.2. Note that the data for column "#Methods/s" you got may be different from the table, as their computation depends on analysis time.

Again, for your convenience, the command argument `<BENCHMARK>` can be repeated for multiple times. For example, to analyze `findbugs`, `antlr` and `lusearch` with `Soot`, use command:

```
$ ./dataflow.py soot findbugs antlr lusearch
```

Also, for your convenience, we provide `-all` to run data flow analysis collectively. When specifying `-all` for `<ANALYSIS>`, it represents all four analysis tools, and when using `-all` for `<BENCHMARK>`, it represents all 14 benchmarks. For example, to run live variable analysis for benchmark `eclipse` on four analysis tools, use command:

```
$ ./dataflow.py -all eclipse
```

To analyze all benchmarks with `SpotBugs`, use command:

```
$ ./dataflow.py spotbugs -all
```

Similar with pointer analysis, when you specify to analyze more benchmarks (either by given multiple benchmark names or use `-all` option), the `dataflow.py` script will output the average numbers for the results of the specified benchmarks at the end.

3.3 Reproducing Tables 1 and 2

As mentioned in Section 1, in Tables 1 and 2 of RQ4 in the accompanying paper, we give the summarized average numbers of each analysis tools for all the benchmarks (excluding unavailable cases, e.g., the analysis runs out of time budget). These results (except the ones concerning analysis time which vary on different runtime environments) can be easily reproduced with `-all` options, which output the average results at the end of each execution.

Although the results concerning analysis time may not be precisely reproduced, you should be able to observe the same performance trends among different analysis tools as in the paper, e.g., `Tai-e` generally runs faster than all other tools for context-insensitive pointer analysis.

In Table 1, for reproducing data under columns "Recall" and "Context Insensitivity", use command:

```
$ ./pointer.py -all -all
```

For column “2-obj”, use command:

```
$ ./pointer.py -all -pta=2-obj -all
```

For column “2-call”, use command:

```
$ ./pointer.py -all -pta=2-call -all
```

Note that the commands for “2-obj” and “2-call” could be very time consuming.
To reproduce Table 2, use command:

```
$ ./dataflow.py -all -all
```

The data for column “#Methods/s” you got may also be different from Table 2 of the paper, as their computation depends on analysis time.