Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis (Artifact Document)

1 Introduction

This artifact is provided to reproduce the results of all research questions (RQ1 - RQ3) in Section 5 of our companion paper, i.e., the data in:

- Table 1
- Table 2
- Table 3
- Figure 12
- The recall experiment in Section 5.1

To thoroughly evaluate the effectiveness and demonstrate the generality of our CUT-SHORTCUT (`csc` for short) pointer analysis approach, we implemented it on two totally diverse pointer analysis frameworks: the declarative DOOP (written in Datalog) and the imperative TAI-E (written in Java). We conducted two sets of experiments on both frameworks, which may make our evaluation complex. However, for your convenience, we provide 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. In this document, we will first introduce how to setup our artifact and run a basic test in Section 2, and then explain how to use the artifact with different options and reproduce all experimental results in Section 3

2 Getting Start Guide

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 smaller benchmarks (e.g., `hsqldb`, `eclipse`, and `findbugs`) on framework TAI-E (generally, TAI-E consumes less memory than DOOP), and avoid those heavy analyses (like `2obj`). **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 2.1 have been installed).

2.2 Artifact Package

The artifact package contains the following files:

- `csc-artifact.tar.gz`: the compressed Docker image.
- `README.pdf`: this artifact document.
- `LICENCE`: the license file for our artifact.

2.3 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/.

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 csc-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 `csc:pldi23` is the image name and `csc-artifact` is the container name:

   ```
   $ docker run --name csc-artifact -it csc:pldi23
   ```

   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.

```
cosmo@cosmodembp desktop % docker load --input csc-artifact.tar.gz
d543b8cad89e: Loading layer [==================================================>]  75.16MB/75.16MB
147254bec711: Loading layer [==================================================>]  7.348GB/7.348GB
9f8e363fbf43: Loading layer [==================================================>]  370.2MB/370.2MB
Loaded image: csc:pldi23
cosmo@cosmodembp desktop % docker run --name csc-artifact -it csc:pldi23
root@8debd67efca9:/#
```

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 csc-artifact
   $ docker exec -it csc-artifact bash
   ```

2.4 Basic Testing

To test whether the artifact has been successfully set up, please first change your current directory to the artifact folder:

```
$ cd /home/artifact
```

Then you can use the following command to run a pointer analysis using Cut-Shortcut approach (`csc` for short) for our smallest benchmark `hsqldb` on framework TAI-E: (Note that running pointer analysis on DOOP is significantly more time-consuming than TAI-E, and we will introduce how to reproduce the results for DOOP in Section 3).

```
$ python run.py tai-e csc hsqldb
```

This command will start TAI-E to perform `csc` for `hsqldb`, and should be finished within 1 minute. As shown in Figure 2, the precision metrics for pointer analysis (`#fail-cast, #reach-mtd, #poly-call, #call-edge`) will be printed out, and `[Pointer analysis] elapsed time` in the output corresponds to the `Time(s)` column in Table 2 of our companion paper.

```
root@70acf2122a7f:/home/artifact# python run.py tai-e csc hsqldb
Running csc for hsqldb (on Tai-e) ....
Building world of input program(s)...
[Build IR for all methods] elapsed time: 8.77s
5817 classes with 58719 methods in the world
Pointer Analysis starts ...
Using reflection log from /home/artifact/tai-e/java-benchmarks/dacapo-2006/hsqldb-refl.log
[Pointer analysis] elapsed time: 4.16s
------------------------ Pointer analysis statistics: ------------------------
#var pointers:                77,889
#objects:                     6,601
#var points-to:               786,647
#static field points-to:      2,040
#instance field points-to:    139,113
#array points-to:             22,023
#reachable methods:           11,335
#call graph edges:            57,986
#may-fail-cast:               1,169
#poly-call:                   1,618
```

Figure 2: An example for running pointer analysis on TAI-E.

If you get the same results for the 4 metrics and similar result for the analysis time (which varies on different physical machines), then the artifact should have been successfully setup, and the user should have no technical difficulties with the rest of the artifact.

In addition, for the users who are interested in the detailed results of each metric (e.g., which casts may fail or which methods are reachable, etc.), we output them into directory `home/artifact/output`. Files in this directory are organized by the framework used, and are named according to the benchmark, the analysis and the metric. For example, `output/tai-e/hsqldb-csc-fail-cast.txt` stores the detailed results for the metric `fail-cast` under analysis `csc` for the benchmark `hsqldb` on framework TAI-E.

3 Step-by-Step Instructions

3.1 Content of Artifact

Once you launched the container and entered the interactive shell, please first use command `cd` to enter the artifact folder (`/home/artifact/`), and then you can view the content of our artifact using `ls` command.

- `benchmarks/`: This folder contains the 10 Java programs and the Java library used in our evaluation.
- `tai-e/`: This folder contains the source code of TAI-E framework (a state-of-the-art imperative static analysis framework for Java), along with the imperative implementation of our Cut-Shortcut approach. For your convenience, we also provide a pre-built executable JAR file `tai-e-all.jar` in this folder. For those who would like to rebuild TAI-E from source code, simply use the following command:

  ```
  $ cd /home/artifact/tai-e/source
  $ ./gradlew fatJar
  ```

  A new `tai-e-all.jar` will be generated by `gradle` and put into directory `/home/artifact/tai-e` to replace the original one.
- `doop/`: This folder contains the source code of DOOP framework (a state-of-the-art declarative pointer analysis framework for Java), along with the declarative implementation of Cut-Shortcut approach. There is no need for users to build DOOP from source code since DOOP will be auto-compiled when executed.
- `output/`: This folder contains the detailed analysis results for the precision metrics we used in our paper.

- `recall/`: This folder contains all the prepared dynamically-recorded reachable methods and call graph edges for each benchmark. It also contains the scripts for examining the recall rate of these two metrics for different analyses.
- `run.py`: a Python script for driving all the provided analyses.
- `overlap.py`: a Python script for reproducing the results in Table 3.

3.2 Running Experiments

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

```
$ python run.py  doop|tai-e  <ANALYSIS>  <BENCHMARK>
```

The first argument (`doop|tai-e`) specifies to run pointer analysis on DOOP or TAI-E.

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

```
ci, 2obj, 2type, zippere, csc
```

in which `ci` represents context-insensitivity (CI in our paper), `2obj` and `2type` represents two traditional context sensitivity (2-object-sensitive and 2-type-sensitive), `zippere` represents ZIPPER^e guided 2-object-sensitivity (ZIPPER^e is a state-of-the-art selective context sensitive pointer analysis tool which uses 2-object-sensitive by default) and `csc` represents our CUT-SHORTCUT approach (CSC in our paper).

`<BENCHMARK>` can be one of the following Java programs analyzed in our experiments (we will introduce how to analyze a new program in Section 3.6):

```
eclipse, gruntspud, freecol, soot, briss, columba, hsqldb, jython, jedit, findbugs
```

For example, to use `ci` to analyze `eclipse` on DOOP, use command:

```
$ python run.py doop ci eclipse
```

The analysis output will be printed on the console as shown in Figure 3. Note that `analysis execution time (sec)` corresponds to column `Time(s)` in Table 1 and the results in Figure 12 of our companion paper.

For your convenience, the command argument `<ANALYSIS>` and `<BENCHMARK>` can be repeated for multiple times. For example, to run both `zippere` and `csc` for `findbugs`, `jython` and `soot` on TAI-E (2 analyses $\times$ 3 benchmarks = 6 analysis executions in total), use command:

```
$ python run.py tai-e zippere csc findbugs jython soot
```

Then the 6 analysis executions will be performed in sequence.

Also, for your convenience, we provide two options `all-analyses` and `all-benchmarks` to run pointer analysis collectively. Option `all-analyses` represents all 5 given analyses and `all-benchmarks` represents all 10 benchmarks. For example, to run all given analyses for benchmark `eclipse` and `hsqldb` on TAI-E, use command:

```
$ python run.py tai-e all-analyses eclipse hsqldb
```

To run `zippere` and `csc` for all benchmarks on TAI-E, use command:

```
$ python run.py tai-e zippere csc all-benchmarks
```

For saving your time, when option `all-analyses` or `all-benchmarks` are used, the analyses executions that run beyond the time limit in our experiment (i.e., ">2h" in Figure 12, Tables 1 and 2 of our paper) will be skipped. To check whether an analysis execution will run beyond the time limit on your machine, you can run this analysis individually using the command we introduced at the beginning of this Section.

```
root@70acf2122a7f:/home/artifact# python run.py doop ci eclipse
Running ci for eclipse on Doop....

> Task :run
WARNING: Using custom platforms library in /home/artifact/benchmarks. Unset environment variable DOOP_PLATFORMS_LIB for default
platform discovery.
Running dacapo benchmark: eclipse
Using TAMIFLEX information from ../benchmarks/dacapo-2006/eclipse-refl.log
Main class(es) expanded with 'Harness'
WARNING: Handling of Java reflection via Tamiflex logic!
New context-insensitive analysis
Id       : 222d2bc4a8d6821c60e6a9916e8338d1dc33d9add6eaf073c0acab6993c08475
Inputs   : ../benchmarks/dacapo-2006/eclipse.jar
Libraries: ../benchmarks/dacapo-2006/eclipse-deps.jar
Using a timeout of 120 min.
[Task FACTS...]
[Task COMPILE...]
-- Fact Generation --
Using cached analysis executable /home/artifact/doop/cache/souffle-analyses/context-insensitive/c58291b9138759341a0b6007ded20a6b
269e1d23f268acbdb9d9048eccaf682d
[Task COMPILE Done]
SOOT_FACT_GEN: Logging initialized, using directory: /home/artifact/doop/build/logs
SOOT_FACT_GEN: SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SOOT_FACT_GEN: SLF4J: Defaulting to no-operation (NOP) logger implementation
SOOT_FACT_GEN: SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
SOOT_FACT_GEN: SSA not enabled, generating Jimple instead of Shimple
SOOT_FACT_GEN: Adding archive: ../benchmarks/dacapo-2006/eclipse.jar
SOOT_FACT_GEN: Classes in input (application) jar(s): 413
SOOT_FACT_GEN: Total classes in Scene: 5390
SOOT_FACT_GEN: Retrieved all bodies (time: 30)
SOOT_FACT_GEN: Fact generation cores: 40
Soot fact generation time: 47
Caching facts in /home/artifact/doop/cache/33eff5e10b60b0164757165ba68eb837ce438312521766efaed0614bac6e4f2d
Copying /home/artifact/doop/out/222d2bc4a8d6821c60e6a9916e8338d1dc33d9add6eaf073c0acab6993c08475/database to /home/artifact/doop
/cache/33eff5e10b60b0164757165ba68eb837ce438312521766efaed0614bac6e4f2d...
----
[Task FACTS Done]
Running analysis
Analysis execution time (sec): 219
-- Runtime metrics --
analysis compilation time (sec)                                    0
analysis execution time (sec)                                      219
disk footprint (KB)                                                24,173,246
fact generation time (sec)                                         47
-- Statistics --
reachable casts that may fail                                      5,077
reachable methods                                                  23,549
polymorphic virtual call sites                                     10,744
call graph edges                                                   183,478
var points-to                                                      27,686,613
array index points-to                                              479,323
instance field points-to                                           3,840,933
static field points-to                                             46,314
Making database available at /home/artifact/doop/results/eclipse/context-insensitive/java_6/222d2bc4a8d6821c60e6a9916e8338d1dc33
d9add6eaf073c0acab6993c08475
Making database available at /home/artifact/doop/last-analysis
```

Figure 3: An example for running pointer analysis on DOOP.

Results of Analysis ZIPPER^e Note that for `zippere`, the pointer analysis contains three parts: (1) a context-insensitive pointer analysis, (2) the tool ZIPPER^e itself, and (3) the ZIPPER^e-guided main analysis. **The time cost we recorded in our paper is the sum of execution time of the three parts, and the 4 precision metrics in our paper are the ones produced by the ZIPPER^e-guided main analysis**.

Discrepancy of Framework DOOP's Results DOOP leverages Soot to generate the facts used by pointer analysis. Due to Soot's non-deterministic fact generation, some analyses results may also be non-deterministic, which means that **for each time you run the same analysis, you may get slightly different results for the same metric**. **So some discrepancies may be observed between the artifact results and the results in Table 1**, but the discrepancy is generally very minor and it does not affect the reliability of analysis results.

Columba Issue on TAI-E. This artifact can reproduce all results for precision metrics of TAI-E for all benchmarks in our paper except the case for benchmark `columba`. After investigation, we found that as long as we run TAI-E on physical machine, TAI-E's results for `columba` are consistent with our paper (we ran TAI-E on both Linux and macOS, and obtained the same results); however, the results are slightly different only when we run it in Docker. Hence, we think the difference is caused by Docker, and it does not affect the reliability of analysis results in the paper.

3.3 Reproducing Table 1 and Table 2

Next we will provide instructions on how to reproducing the data of Table 1 and Table 2 in our paper (As for Figure 12, it is directly derived by the `Time(s)` column in Table 1, so it can be easily checked after reproducing Table 1). We provide a simple command to run all the analyses in Table 1 (i.e., all 5 analyses for all 10 benchmarks on framework DOOP):

```
$ python run.py table1
```

Similarly, to get all data of Table 2, simply run:

```
$ python run.py table2
```

Also, these two commands will skip those analyses executions that run beyond the time limit to save your time.

3.4 Reproducing Table 3

Table 3 in our paper presents a detailed comparison between ZIPPER^e and CUT-SHORTCUT on DOOP (left half) and TAI-E (right half). Below we explain how to obtain the results of each column in Table 3.

Elapsed Time In Table 3, the column "Total Time" of ZIPPER^e and column "Time" of CSC conform to the column "Time" of ZIPPER^e and CSC analyses respectively in Tables 1 and 2. The column "Pre-analysis" of ZIPPER^e gives the sum of elapsed time of the first two parts of ZIPPER^e analysis, while the column "Main analysis" gives the elapsed time for main analysis of ZIPPER^e (i.e., ZIPPER^e-guided context-sensitive analysis). These results can all be derived directly from the console outputs of the corresponding analyses.

Selected Methods, Involved Methods, and Overlapped Methods To reproduce the numbers in columns "Selected methods" (for ZIPPER^e), "Involved methods" and "Overlapped methods" (for CSC), we provide a script `overlap.py` under `/home/artifact/`. Note that the action conducted by `overlap.py` is based on the analysis results of `zippere` and `csc`, so please ensure that these two analyses have already been executed for the benchmark, and then run the following command under `/home/artifact/`:

```
$ python overlap.py doop|tai-e <BENCHMARK>
```

Figure 4 shows a usage example of `overlap.py`.

```
root@2e957b628788:/home/artifact# python overlap.py doop hsqldb
for hsqldb (framework doop)
Number of selected methods in zippere: 1846
Number of involved methods in csc: 1914
Number of overlapped methods: 784 (41.0%)
```

Figure 4: Reproducing "Selected methods", "Involved methods", and "Overlapped methods" for `hsqldb` on DOOP.

3.5 The Recall Experiment

With the options introduced in Sections 3.2– 3.4, you can already reproduce all the tables and figures in the evaluation (Section 5) of our paper. Additionally, we provided instructions for generating the detailed results of the recall experiment in Section 5.1 (*Soundness (Recall)*) of our companion paper. **Note that in our companion paper, only the summarized recall conclusion is given, but for the users who want to thoroughly examine the recall experiment, we offer the functionalities in our artifact to further investigate the detailed recall results, as introduced in this Section**. As we mentioned in the paper, we ran a recall experiment to validate the soundness of `csc` by executing the benchmark programs with their default tests (if available) or running them manually (e.g., click to interact with GUI programs), recording their reachable methods and call graph edges during dynamic execution, and then examining how many of them can be recalled by `csc` and other analyses. Since it is too troublesome for users to executing the benchmarks manually to get the same dynamic results in our experiment, we provided the dynamically recorded reachable methods and call graph edges we got for each benchmark in this artifact (in directory `/home/artifact/recall/dynamic`) and we will explain how to use them to compute the recall rate for different analyses.

Examining Recall Rate for Different Analyses First, you need to run the analysis you want to examine as introduced in Section 3.2, and then change your working directory to `recall` folder:

```
$ cd /home/artifact/recall
```

In this directory we provided a Python script `recall.py` which uses the dynamic results in folder `dynamic` and the analyses output in `/home/artifact/output/tai-e` (or `/home/artifact/output/doop`) to compute the recall rate of different analyses for different benchmarks. To run this script, use command:

```
$ python recall.py doop|tai-e <BENCHMARK>|all
```

For example, if you have run `csc`, `ci` and `2type` for `hsqldb` on DOOP with this artifact (so that the metrics of these analyses are output into directory `/home/artifact/output/doop`), to get the recall rate of these analyses, type:

```
$ python recall.py doop hsqldb
```

then the recall rate of reachable methods and call graph edges for these three analyses on `hsqldb` will be printed on the screen as shown in Figure 5. In addition, the missed reachable methods (and call edges) for different analyses compared with the dynamically recorded results are output into directory `home/artifact/recall/results/doop` (For TAI-E, the usage is the same, and just replace `doop` by `tai-e`).

```
root@15bd0b888be8:/home/artifact/recall# python recall.py doop hsqldb
Recall on hsqldb (framework doop)
Compare ci with dynamic on hsqldb-reach-mtd.txt
Recall: 2485/2733 (90.93%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-ci-dynamic-missed-reach-mtd.txt
Compare 2type with dynamic on hsqldb-reach-mtd.txt
Recall: 2485/2733 (90.93%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-2type-dynamic-missed-reach-mtd.txt
Compare zippere with dynamic on hsqldb-reach-mtd.txt
Recall: 2485/2733 (90.93%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-zippere-dynamic-missed-reach-mtd.txt
Compare csc with dynamic on hsqldb-reach-mtd.txt
Recall: 2484/2733 (90.89%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-csc-dynamic-missed-reach-mtd.txt
Compare ci with dynamic on hsqldb-call-edge.txt
Recall: 5572/6295 (88.51%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-ci-dynamic-missed-call-edge.txt
Compare 2type with dynamic on hsqldb-call-edge.txt
Recall: 5564/6295 (88.39%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-2type-dynamic-missed-call-edge.txt
Compare zippere with dynamic on hsqldb-call-edge.txt
Recall: 5566/6295 (88.42%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-zippere-dynamic-missed-call-edge.txt
Compare csc with dynamic on hsqldb-call-edge.txt
Recall: 5564/6295 (88.39%)
Output missed items to /home/artifact/recall/results/doop/hsqldb-csc-dynamic-missed-call-edge.txt
```

Figure 5: Recall rate of different analyses for `hsqldb` on DOOP.

Examining Detailed Recall Results To examine the soundness of `csc`, for each benchmark, we compute the recall rate of `csc` and other scalable analyses. The detailed recall results are shown in Table 3.5.1 (**Note that this table is not given in our companion paper; the recall results in the table are only summarized in Section 5.1 (*Soundness (Recall)*) of our companion paper**).

As shown in Table 3.5.1, the recall rate of `csc` is no lower than that of `2obj`, `2type` or `zippere` for most benchmark programs. For the other cases (e.g., call-edge of `jython`), the recall rate of `csc` is slightly lower than all the compared analyses. For them, we further provide a script `diff.py` under directory `/home/artifact/recall/results` to compute the reachable methods and call graph edges that are missed by `csc` (compared with the dynamically recorded results) but are not missed by the compared analyses. (Note that those reachable methods and call edges which are missed by both `csc` and any of the compared analyses are out of interest since it means that the unsoundness are not introduced by `csc`). To use `diff.py`, first change to `results` directory and then run the script:

```
$ cd /home/artifact/recall/results
$ python diff.py doop|tai-e <BENCHMARK>|all
```

For example, Figure 6 shows the result for benchmark `jython` on DOOP. Also, the argument `<BENCHMARK>` can be repeated multiple times and the user can simply use `all` to examining all benchmarks. In addition, By running `diff.py`, the de-duplicated (since different benchmarks may miss the same items) missed reachable methods and call graph edges for `csc` are output into directory `/home/artifact/recall/results/csc-missed/`.

```
root@15bd0b888be8:/home/artifact/recall/results# python diff.py doop jython
For benchmark jython (framework doop), compared with ci zippere
csc missed reachable methods:
none
csc missed call graph edges:
<java.io.BufferedInputStream: int read(byte[],int,int)>/java.io.InputStream.available/0 <java.io.FilterInputStream: int available()>
<java.io.FilterInputStream: int available()>/java.io.InputStream.available/0    <java.util.zip.ZipFile$1: int available()>
<java.io.FilterInputStream: int available()>/java.io.InputStream.available/0    <java.util.zip.ZipFile$ZipFileInputStream: int available()>
<java.io.FilterInputStream: int read(byte[],int,int)>/java.io.InputStream.read/0        <java.util.zip.InflaterInputStream: int read(byte[],int,int)>
<java.io.FilterInputStream: int read(byte[],int,int)>/java.io.InputStream.read/0        <java.util.zip.ZipFile$ZipFileInputStream: int read(byte[],int,int)>
<java.io.FilterInputStream: void close()>/java.io.InputStream.close/0   <java.util.zip.ZipFile$1: void close()>
<java.io.FilterInputStream: void close()>/java.io.InputStream.close/0   <java.util.zip.ZipFile$ZipFileInputStream: void close()>

the de-duplicated missing reachable methods (0 in total) for doop are output to ./csc-missed/doop-csc-missed-reach-mtd.txt
the de-duplicated missing call graph edges (7 in total) for doop are output to ./csc-missed/doop-csc-missed-call-edge.txt
```

Figure 6: Reachable methods and call graph edges missed by `csc` compared with other analyses for `jython` on DOOP.

For your convenience, we have prepared all missing items for all benchmarks in the same directory (`/home/artifact/recall/results/csc-missed/`) in advance. The resulting files are named by `<FRAMEWORK>-csc-missed-<METRIC>-provided.txt` (e.g., `doop-csc-missed-call-edge-provided.txt` for all call graph edges missed by `csc` in all benchmarks on DOOP). To reproduce the same resulting files, you should first run all analyses for all benchmarks on DOOP (using script `run.py`) and output the detailed recall results for them (using script `recall.py`), then simply use the following command (For TAI-E, the usage is the same, and just replace `doop` by `tai-e`):

```
$ python diff.py doop all
```

In total, compared with other analyses, `csc` missed 20 call graph edges on DOOP, and 11 call graph edges on TAI-E. We manually inspected all these missing items, and found that they are not true and are incorrectly recorded by the instrumentation tool. (The number is a bit different from what we mentioned in our paper (10 edges for DOOP and 12 edges for TAI-E) as when preparing this artifact, we found that we made a counting mistake in our submission paper, which will be corrected in the final version. But this does not change our conclusion that `csc` is sound because all the missing items, including the miscounted ones, are found not true.)

3.6 Analyzing A New Program

Our artifact also supports users to analyze a new program, beyond the benchmarks analyzed in the evaluation of our paper. To analyze a new program, please first pack the program you want to analyze into JAR file(s), including the application itself and its dependent libraries. Suppose that your new program is packed into 2 application JARs `newapp1.jar` and `newapp2.jar`, with 2 library JARs `newapplib1.jar` and `newapplib2.jar`. Then please organize the JAR files into a folder following the structure shown in Figure 7 (if your program does not depend on third-party library, then just ignore the `lib` directory):

Then, please copy this folder into the `benchmarks/` directory in our Docker container using command:

```
$ docker cp newapp csc-artifact:/home/artifact/benchmarks/newapp
```

Next, please add the information of the new program in `/home/artifact/benchmarks/app-info.yml` from which our scripts read program information. For the example described above, just add the content in Figure 8 to the end of `app-info.yml`.

Now, you can analyze the new program with our `run.py` script. Please first enter the Docker container, change working directory to `/home/artifact/`, and then simply run the following command:

```
$ python run.py doop|tai-e <ANALYSIS> <PROGRAM-ID>
```

where `<PROGRAM-ID>` is the program ID configured by the users in `app-info.yml`, e.g., `newapp` in the above example.

Table 3.5.1: Detailed Recall Results.

Program	Dynamic		DOOP					TAI-E				
	#reach	#call	Analysis	#reach-mtd		#call-edge		Analysis	#reach-mtd		#call-edge	
	-mtd	-edge		Recall	Rate	Recall	Rate		Recall	Rate	Recall	Rate
eclipse	8,093	20,916	CI	6,935	85.69%	17,473	83.54%	CI	7,080	87.48%	18,050	86.30%
			2obj	–	–	–	–	2obj	7,078	87.46%	18,037	86.24%
			2type	6,920	85.51%	17,425	83.31%	2type	7,078	87.46%	18,038	86.24%
			ZIPPER^e	6,933	85.67%	17,465	83.50%	ZIPPER^e	7,078	87.46%	18,040	86.25%
			CSC	6,933	85.67%	17,460	83.48%	CSC	7,078	87.46%	18,038	86.24%
freecol	19,387	57,455	CI	18,585	95.86%	54,566	94.97%	CI	19,119	98.62%	56,593	98.50%
			2obj	–	–	–	–	2obj	–	–	–	–
			2type	–	–	–	–	2type	–	–	–	–
			ZIPPER^e	18,573	95.80%	54,513	94.88%	ZIPPER^e	19,117	98.61%	56,586	98.49%
			CSC	18,579	95.83%	54,550	94.94%	CSC	19,119	98.62%	56,585	98.49%
briss	14,244	39,575	CI	14,016	98.40%	38,782	98.00%	CI	14,010	98.36%	38,743	97.90%
			2obj	–	–	–	–	2obj	–	–	–	–
			2type	–	–	–	–	2type	–	–	–	–
			ZIPPER^e	14,005	98.32%	38,733	97.87%	ZIPPER^e	14,002	98.30%	38,699	97.79%
			CSC	14,008	98.34%	38,735	97.88%	CSC	14,002	98.30%	38,696	97.78%
hsqldb	2,733	6,295	CI	2,485	90.93%	5,572	88.51%	CI	2,608	95.43%	5,856	93.03%
			2obj	–	–	–	–	2obj	–	–	–	–
			2type	2,485	90.93%	5,564	88.39%	2type	2,607	95.39%	5,846	92.87%
			ZIPPER^e	2,485	90.93%	5,566	88.42%	ZIPPER^e	2,607	95.39%	5,846	92.87%
			CSC	2,484	90.89%	5,564	88.39%	CSC	2,607	95.39%	5,850	92.93%
jedit	6,028	13,203	CI	5,866	97.31%	12,732	96.43%	CI	5,859	97.20%	12,707	96.24%
			2obj	–	–	–	–	2obj	5,853	97.10%	12,688	96.10%
			2type	5,855	97.13%	12,704	96.22%	2type	5,853	97.10%	12,688	96.10%
			ZIPPER^e	5,856	97.15%	12,708	96.25%	ZIPPER^e	5,853	97.10%	12,688	96.10%
			CSC	5,859	97.20%	12,712	96.28%	CSC	5,853	97.10%	12,688	96.10%
gruntspud	14,543	42,099	CI	14,340	98.60%	41,419	98.38%	CI	14,360	98.74%	41,426	98.40%
			2obj	–	–	–	–	2obj	–	–	–	–
			2type	–	–	–	–	2type	–	–	–	–
			ZIPPER^e	14,337	98.58%	41,403	98.35%	ZIPPER^e	14,359	98.73%	41,414	98.37%
			CSC	14,339	98.60%	41,410	98.36%	CSC	14,359	98.73%	41,414	98.37%
soot	4,372	16,452	CI	4,285	98.01%	16,222	98.60%	CI	4,288	98.08%	16,233	98.67%
			2obj	–	–	–	–	2obj	–	–	–	–
			2type	–	–	–	–	2type	–	–	–	–
			ZIPPER^e	–	–	–	–	ZIPPER^e	4,287	98.06%	16,230	98.65%
			CSC	4,284	97.99%	16,219	98.58%	CSC	4,287	98.06%	16,230	98.65%
columba	6,757	14,689	CI	6,681	98.88%	14,401	98.04%	CI	6,673	98.76%	14,368	97.81%
			2obj	–	–	–	–	2obj	–	–	–	–
			2type	–	–	–	–	2type	–	–	–	–
			ZIPPER^e	–	–	–	–	ZIPPER^e	6,673	98.76%	14,368	97.81%
			CSC	6,681	98.88%	14,393	97.98%	CSC	6,673	98.76%	14,368	97.81%
jython	4,835	35,970	CI	4,055	83.87%	10,857	30.18%	CI	4,242	87.74%	11,639	32.36%
			2obj	–	–	–	–	2obj	–	–	–	–
			2type	–	–	–	–	2type	–	–	–	–
			ZIPPER^e	4,053	83.83%	10,851	30.17%	ZIPPER^e	4,240	87.69%	11,633	32.34%
			CSC	4,055	83.87%	10,850	30.16%	CSC	4,240	87.69%	11,626	32.32%
findbugs	5,857	14,075	CI	5,662	96.67%	13,319	96.63%	CI	5,808	99.16%	13,899	98.75%
			2obj	–	–	–	–	2obj	5,808	99.16%	13,898	98.74%
			2type	5,662	96.67%	13,318	96.62%	2type	5,808	99.16%	13,898	98.74%
			ZIPPER^e	5,662	96.67%	13,319	96.63%	ZIPPER^e	5,808	99.16%	13,899	98.74%
			CSC	5,662	96.67%	13,318	96.62%	CSC	5,808	99.16%	11,898	98.74%

```
newapp/
    |--app/
    |   |--newapp1.jar
    |   |--newapp2.jar
    |--lib/
        |--newapplib1.jar
        |--newapplib2.jar
```

Figure 7: Structure of the `newapp` folder.

```
newapp:  # id of the program
  jdk: 6 # analyzed JDK version
  main: my.program.Main # main class of the program
  apps:
    - newapp/app # path of the application JAR directory
                 # relative to /home/artifact/benchmarks/
  libs:
    - newapp/lib # path of the library JAR directory
                 # relative to /home/artifact/benchmarks/
```

Figure 8: Configuration in `app-info.yml` for `newapp`.

3.7 Potential Issue

When running DOOP, if it exits with an exception as shown in Figure 9, it means that the memory of your machine is too small to execute the analysis:

```
Generic exception java.lang.RuntimeException: Command exited with non-zero status:
 /home/artifact/doop/cache/souffle-analyses/2-object-sensitive+heap/f6b953fb2ae434a9b1c188b79f189ce1a149169016cecb6d5b421734b2c8076d -j4 -F/home/artifact/doop/out/b5598c2c08f42a
4e42bf61d17f0ecefa607a25106a3a2a423cbd6b9f0ceb9b1e/database -D/home/artifact/doop/out/b5598c2c08f42a4e42bf61d17f0ecefa607a25106a3a2a423cbd6b9f0ceb9b1e/database
Command exited with non-zero status:
 /home/artifact/doop/cache/souffle-analyses/2-object-sensitive+heap/f6b953fb2ae434a9b1c188b79f189ce1a149169016cecb6d5b421734b2c8076d -j4 -F/home/artifact/doop/out/b5598c2c08f42a
4e42bf61d17f0ecefa607a25106a3a2a423cbd6b9f0ceb9b1e/database -D/home/artifact/doop/out/b5598c2c08f42a4e42bf61d17f0ecefa607a25106a3a2a423cbd6b9f0ceb9b1e/database
java.lang.RuntimeException: Command exited with non-zero status:
 /home/artifact/doop/cache/souffle-analyses/2-object-sensitive+heap/f6b953fb2ae434a9b1c188b79f189ce1a149169016cecb6d5b421734b2c8076d -j4 -F/home/artifact/doop/out/b5598c2c08f42a
4e42bf61d17f0ecefa607a25106a3a2a423cbd6b9f0ceb9b1e/database -D/home/artifact/doop/out/b5598c2c08f42a4e42bf61d17f0ecefa607a25106a3a2a423cbd6b9f0ceb9b1e/database
        at org.clyze.utils.Executor.invoke(Executor.groovy:83)
        at org.clyze.utils.Executor.execute(Executor.groovy:32)
        at org.clyze.utils.Executor.execute(Executor.groovy)
        at org.clyze.doop.utils.SouffleScript$_run_closure2.doCall(SouffleScript.groovy:220)
        at org.clyze.doop.utils.SouffleScript$_run_closure2.call(SouffleScript.groovy)
        at org.clyze.utils.Helper.timing(Helper.groovy:74)
        at org.clyze.doop.utils.SouffleScript.run(SouffleScript.groovy:219)
        at org.clyze.doop.core.SouffleAnalysis.run(SouffleAnalysis.groovy:117)
        at org.clyze.doop.Main$1.run(Main.groovy:119)
```

Figure 9: Exception on DOOP if memory is too small to scale the analysis.