# Program Analysis and Verification
# Final Project: Recording Java Traces

Students: Saar Scheinkman, ID: 203853171, saarsch@post.bgu.ac.il
and Nevo Mashiach, ID: 308435924, nevoma@post.bgu.ac.il

August 5, 2018

**Introduction**

The goal of this project is to be able to record the traces of a Java program, in a way that could later be used to synthesize Java code back from the given traces. In order to achieve this goal, we used Soot, adding a **BodyTransformer** we implemented. This transformer is applied on every method of the code before the Jimple/Class file is finally written. We instrument only one method at a time, so the transformer is only relevant for the chosen method (and the 'main' method which has one line added to it).

**Implementation**

The **CodeImplant** class inherits from **BodyTransformer**, allowing it to be added to the Soot transformation sequence. This transformer is responsible for implanting lines of code that call our **Logger** methods which are responsible for tracking and logging the environment's state at every step.

The **Logger** class implements the methods that are responsible for logging the state of the environment at each step. The implanted lines in the modified class file (after every Java command) call a logging method for each one of the local variables (including the modified method's arguments), which in it's turn tracks the current state of every variable and writes it's value (if -deltas is set, the value is printed only if it has been changed since the last time it was printed). The logger is also responsible for printing the commands themselves (if -nologcmd is not set) and finally writing the result specification into a file (Test_Results.spec for example).

The **Project** class includes the main, which handles the program's arguments and calls Soot with the **CodeImplant** transformer added. When it finished running, the result is a modified class file of the tested program (Benchmarks in our case) which is ready to be ran (writing the specification to the .spec file after it is ran).

**Running instructions**

1. Import the project into IntelliJ. Note this is an IntelliJ project for ease of use - the project can be run with any IDE or without it.
2. Run the main method in **Project** using the following run arguments:
   **-cp . [-f jimple](a) -p jb use-original-names -keep-line-number -print-tags -pp Benchmarks factorial [-deltas](b) [-nologcmd](c)**.
   In this example, **Benchmarks** is the instrumented class name and **factorial** is the instrumented method name.

   (a) Optional for jimple output. To get a runnable class file this must be removed.

   (b) Optional for deltas-stores mode: Only the delta between each state will be stored.

   (c) Optional for disabling command logging: Only the stores will be printed, omitting the commands.
3. Use windows CMD to run the following Java command:
   **java -cp C:\Users\saars\IdeaProjects\Project\sootOutput Benchmarks**.
   When the path points the the absolute location of the sootOutput directory inside the project. This is where the Benchmarks.class file (which is the modified .class file that uses our Logger) is located after step 2.

Notes:

1. The Logger.class file must be placed inside sootOutput.
2. The attached JAR files must be placed in the Project's root directory as in the submitted directory.

In other words, all the dependencies must be in the same directory as the tested file (Benchmarks in our case).

**Benchmarks**

1. **simpleNumericTest**: Demonstrates the ability to handle simple assignments.
2. **gcd**: Greatest common divisor calculation, demonstrates the ability to handle loops and conditions.
3. **factorial**: Factorial calculation, demonstrates the ability to handle loops and conditions.
4. **simpleSLL**: Simple assignments of SLLNodes, demonstrates the handling of objects in addition to integers.
5. **findMax**: Demonstrates more complex iterations over a SLL.
6. **reverse**: Demonstrates more complex iterations over a SLL.

All our benchmarks .spec, Java and .dfy files that were generated by Pexyn are in the 'Test Results' directory.

**Limitations**

1. Since every new method examined by our transformer causes a new initialization of the static logger, tested methods shouldn't call other tested methods, to prevent the clearing of variable-maps and the printed strings.
2. Due to an issue with Soot, a tested method may not instantiate new objects.

Being cognizant that academic dishonesty is an offense against the regulations of the university, the faculty of natural sciences, the department of computer science, and the program analysis and verification course, we hereby assert that the work we submit for this assignment is entirely our own. We have made no use of solutions and/or of code from other students in the class, from students who took the class in previous years, or from any other source outside the class (e.g., a friend, the internet, etc). We realize that if we are suspected of academic dishonesty, we shall be called before the disciplinary committee (Va'adat Mishma'at), and that the minimal penalty for cheating on an assignment is a failing grade in the course. We proudly and steadfastly maintain that this work is entirely and only our own.