The key steps in the writing of a pintool begin with initialization. I noticed there is a command-line parameter/option for some of the tools that allow you to pass values into the Pintool. In the case of the examples provided, two of them utilized Knobs to pass in the file name as input. We can also use Knobs to provide options as well, thresholds or to enable and disable features.

All of the Pintools provided also had instrumentation routines that specified what behavior we were monitoring or modifying. PIN APIs such as INS_AddInstrumentFunction specify what occurs every time a new instruction is encountered. This is called in each of the tools but the contents of this function are different. There is also a RTN_AddInstrumentFunction which adds routines, an IMG_AddInstrumentFunction which instruments images and a TRACE_AddInstrumentFunction which adds traces.

There are also analysis routines that perform analysis on what we desire, in this case its counting instructions, tracing memory instructions, or capturing function arguments. At the end, we have a finalization stage that is used to process/report any of the results that was gathered during the execution. This is done with the PIN_AddFiniFunction function which registers a Fini function to handle post-processing such as the closing of files, displaying output, etc.

When running the Pintools, there was a noticeable overhead when running some basic commands like gedit and ls. The instruction count counted 1389993325 lines when running gedit but only 718386 lines when running ls. This was expected, as increasing functionality will result in a significant amount of instruction calls. When running the memory reference trace, there was a ton of information printed out onto the screen especially when running more complex applications like Firefox. Having a data analysis tool would be critical for understanding data generated from pin.