AFL collabrates with PIN

-Urvashi Sharma

AIM :

To study about how the American Fuzzy Logic works with the instrumentation tool PIN

Motivation :

Since I have worked on AFL and PIN tool this semester, and both tools are used for testing purposes but in different ways, therefore I chose to work on this project to see how efficient the testing would be if both the tools work together.

Contribution:

I have pulled down a project from github project AFLPIN (https://github.com/mothran/aflpin), I have installed AFL and PIN together at the same place with this project. The project allows AFL fuzzer to fuzz the non-instrumented binaries using PIN tool. It is done by inserting the same type of branch detection and shared memory mappings just like AFL adds to instrumented binaries. “Aflpin.cpp” is the pintool which is build to “aflpin.so” file. “sleep_test.c” and “crash_test.c” are the test programs. I will be studying how the PIN is invoked from AFL to run these test programs and which part of the code invokes.

Play with the tool :

Built the aflpin.cpp file adding the filename aflpin.cpp in the MakeFile and the run make. "aflpin.so" file will be created in the obj-intel64 folder.

After I have built aflpin.so successfully, run the following command to invoke PIN tool with a target and afl-fuzz is :

```
$ AFL_NO_FORKSRV=1 afl-fuzz -m 500 -i .. -o .. -f .. -- /path/to/pin_app -t /path/to/obj-intel64|obj-ia32/aflpin.so -- TARGETAPP @@.
```

What Invokes PIN from AFL:

As per studying the AFL, we know that AFL injects instrumentation into compiled programs captures branch coverage. The code injected at branch point is essentially equivalent to:

```
cur_location=<COMPILE_TIME_RANDOM>;
shared_mem[cur_location^prev_location]++;
prev_location = cur_location >> 1;
```

We also know that shared_mem[] is a 64Kb SHM region passed to the instrumented binary by the caller.

This design of instrumentation is done by the PIN tool. Therefore the project develops it's own PIN tool known as aflpin.cpp, which performs the similar instrumentation as that by afl-fuzz.c in qemu mode.

Aflpin.cpp is as such :

As the other PIN tool, it also starts with the void main method as following :

```
int main(int argc, char *argv[])

{
```

```
if(PIN_Init(argc, argv)){
        return Usage();
    }
    setup_shm();
    PIN_SetSyntaxIntel();
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddApplicationStartFunction(entry_point, 0);
    PIN_StartProgram();
    // AFL_NO_FORKSRV=1
    // We could use this main function to talk to the fork server's fd and then enable
  the fork server with this tool...
  }
```

In the main method, following methods are being called :

1. **PIN_Init() and Usage() :** PIN_Init, initializes the PIN tool “aflpin” which enables blackbox binaries to be fuzzed with AFL on Linux and returns the Usage(). The Usage method prints the cerr messages and returns -1,
2. **setup_shm() :** This method returns the bool value.

```
  bool setup_shm() {
    if (char *shm_id_str = getenv("__AFL_SHM_ID")) {
        int shm_id;
        shm_id = std::stoi(shm_id_str);

        std::cout << "shm_id: " << shm_id << std::endl;

        bitmap_shm = reinterpret_cast<uint8_t*>(shmat(shm_id, 0, 0));

        if (bitmap_shm == reinterpret_cast<void *>(-1)) {

            std::cout << red << "failed to get shm addr from shmmat()" << cend <<
    std::endl;

            return false;

        }

      }

      else {

        std::cout << red << "failed to get shm_id envvar" << cend << std::endl;

        return false;

      }

      return true;

    }
```

It gets the environment variable value of AFL_SHM_ID storing the shm_id using getenv() function. Get the shared memory stored at this shm_id which is stored in bitmap_shm. “Bitmap_shm” is a global unsigned int pointer initially set to 0 i.e. uint8_t *bitmap_shm = 0;

If everything is retrieved as expected then, this method returns true, otherwise false printing the error messages as shown in the above method.

3. **PIN_SetSyntaxIntel()** : Sets the disassembly syntax to Intel format.

4. **TRACE_AddInstrumentFunction(Trace, 0)** : The function calls the Trace method where instrumentation is performed. The trace method is the basic trace method of PIN tool which traces the each block and instruction and instruments in it.

```
VOID Trace(TRACE trace, VOID *v)

{

  for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))

  {

    for (INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins))

    {

      if (valid_addr(INS_Address(ins)))

      {

        if (INS_IsBranch(ins)) {

            if (INS_HasFallThrough(ins) || INS_IsCall(ins))

          {

            if (Knob_debug) {



              std::cout << "BRANCH: 0x" << std::hex << INS_Address(ins) <<
":\t" << INS_Disassemble(ins) << std::endl;

            }
```

```
                    // Instrument the code.

                         INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)TrackBranch,

                           IARG_INST_PTR,

                           IARG_END);

                       }

                     }

                   }

              }

               }}
```

Now as per **afl-as.c** we only care about instrumenting the conditional branches, therefore no JMP instructions.

Therefore PIN finds the branch statement using INS_isBranch() and if it is, it checks the conditional branch instruction with the PIN API INS_HasFallThrough(ins). If yes, then it instruments the code, and calls the TrackBranch method by passing the instruction pointer which points to the address of the instruction.

5. **<u>TrackBranch()</u>** : TrackBranch method is the method which performs the AFL logic of :

```
cur_location=<COMPILE_TIME_RANDOM>;
shared_mem[cur_location^prev_location]++;
prev_location = cur_location >> 1;
```

The method is as follows :

```
VOID TrackBranch(ADDRINT cur_addr)
{
  ADDRINT cur_id = cur_addr - min_addr;
  if (bitmap_shm != 0){
     bitmap_shm[((cur_id ^ last_id) % MAP_SIZE)]++;
  }
  else {
     bitmap[((cur_id ^ last_id) % MAP_SIZE)]++;
  }
  last_id = cur_id;
}
```

MAP_SIZE and bitmap_shm are the global variables defined in the program. The method performs the same action as afl-as.c.

6. **PIN_AddApplicationStartFunction(entry_point, 0) :**

This method registers a notification which is called after pin initialization is finished. The notification is availabe when pin launches the application and when pin attaches to a running process.

The method calls the “entry_point()” method which is much like the original instrumentaion from AFL, so that we only instrument segments of code of the actual application only , and not the link and pin itself.

```
VOID entry_point(VOID *ptr)

{

   IMG img = APP_ImgHead();

   for(SEC sec= IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec))

   {

          if (SEC_IsExecutable(sec) && SEC_Name(sec) == ".text")

       {

           ADDRINT sec_addr = SEC_Address(sec);

           UINT64  sec_size = SEC_Size(sec);

            if (Knob_debug)

           {

               std::cout << "Name: " << SEC_Name(sec) << std::endl;

               std::cout << "Addr: 0x" << std::hex << sec_addr << std::endl;

               std::cout << "Size: " << sec_size << std::endl << std::endl;

           }



           if (sec_addr != 0)

           {
               ADDRINT high_addr = sec_addr + sec_size;

               if (sec_addr > min_addr || min_addr == 0)
                   min_addr = sec_addr;

               // Now check and set the max_addr.
               if (sec_addr > max_addr || max_addr == 0)
                   max_addr = sec_addr;

               if (high_addr > max_addr)
                   max_addr = high_addr;
           }
       }
   }
   if (Knob_debug)
   {
       std::cout << "min_addr:\t0x" << std::hex << min_addr << std::endl;
            std::cout << "max_addr:\t0x" << std::hex << max_addr << std::endl <<
std::endl;
   }
```

```
}
```

Limitations/Advantages:

Advantages:

- In the regular AFL, the instrumentation is performed at the compiled time of the target program, I.e. AFL fuzz the instrumented binary file, whereas

  In AFLPIN, dynamic instrumentation is performed at the run time on the compiled binary files. Therefore recompilation of source code is not required like regular AFL does, and it can also support instrumenting programs that dynamically generate code.Hence AFLPIN enables fuzzer to fuzz non instrumented binaries.
- With the AFLPIN, if the AFL reports about the test case that resulted in the crash, then check the pin.log file where you can see PIN specific errors.

Limitations :

- Instrumentation done by PIN add cost to performance of AFL, therefore exec time is slow.
- To run AFLPIN, -m 500 command is also given because pin will be needing a large chunk of memory and might be needed to tune this for a given target.

Conclusion:

AFL is a fuzzer which fuzzes the instrumented binaries. PIN is a tool which is used for dynamic instrumentation of a compiled programs. When AFL performs the instrumentation at compile time and fuzz that particular binary, PIN instruments the compiled binary at run time. With this, this report discuss what it would be if AFL fuzzes the non instrumented binaries and instead the instrumentation is performed by PIN rather AFL.

So we saw that by inserting the same type of branch detection and shared memory mappings that AFL adds to instrumented binaries we can achieve the same instrumentation by PIN tool. Only difference here is the instrumentation is now performed at run time rather than compile time on the targeted program, this also eliminated the re compilation of the program.

Later we also saw how PIN is invoked by AFL and how the instrumentation is performed by PIN just like AFL.

By running the crash_test.c, I found the exec times is slower while using AFLPIN but on the other hand, you can check for PIN specific errors of the test case resulted during crash while fuzzing.