Debug KLEE with GDB on Ubuntu16.04

Weikun Han

1. Install GDB:

KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure, and available under the UIUC open source license. And, GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed [1].

```
$ sudo apt-get install gdb
```

2. Load KLEE into GDB:

First, I need to locate the path where is KLEE program. For me, it can do like that:

```
$ cd ~/Projects/klee/Debug+Asserts/bin
$ gdb klee
```

And you will see like that:

```
weikun@ubuntu:~/Projects/klee/Debug+Asserts/bin$ gdb klee
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from klee...done.
```

Note that the GDB operates on executable files which are binary files produced by the compilation process. Here, KLEE is already a binary file.

3. Understand KLEE running process:

Assignment 1: In the part, I need to know how KLEE processes for certain file. I chose the example file from KLEE tutorial, get_sign.

First, I need set running arguments in order to check how exactly KLEE run. For me, it can do like that *(Debugging with GDB: The GNU Source-Level Debugger*, P28):

```
(gdb) set args ~/Desktop/examples/get_sign/get_sign.bc
```

Next, I can check arguments.

```
(gdb) show args
```

Now, I can use start in GDB to check the KLEE running process *(Debugging with GDB: The GNU Source-Level Debugger*, P26):

```
(gdb) start
```

```
(gdb) start
Temporary breakpoint 1 at 0xb05a2c: file main.cpp, line 1163.
Starting program: /home/weikun/Projects/klee/Debug+Asserts/bin/klee ~/Desktop/ex
amples/get_sign/get_sign.bc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, main (argc=2, argv=0x7fffffffded8, envp=0x7fffffffdef0)
    at main.cpp:1163
1163    int main(int argc, char **argv, char **envp) {
```

Here, I can realize that KLEE program is start from line 1163 in main.cpp.

```
1158   klee_message("NOTE: Using klee-uclibc : %s", uclibcBCA.c_str());
1159   return mainModule;
1160 }
1161 #endif
1162
1163 int main(int argc, char **argv, char **envp) {
1164   atexit(llvm_shutdown);  // Call llvm_shutdown() on exit.
1165
1166   llvm::InitializeNativeTarget();
1167
1168   parseArguments(argc, argv);
1169   sys::PrintStackTraceOnErrorSignal();
1170
```

And I can imagine that next execute instruction should be line 1164. Therefore, I can use step in GDB to see one step by one step how KLEE execute instruction *(Debugging with GDB: The GNU Source-Level Debugger*, P61).

```
(gdb) step
```

```
(gdb) step
1164        atexit(llvm_shutdown);  // Call llvm_shutdown() on exit.
(gdb)
```

From the result, I can see how step in GDB run. I just need to press “ENTER” to see next execute instruction. Also, I can use next in GDB to see KLEE next execute instruction *(Debugging with GDB: The GNU Source-Level Debugger*, P61).

```
(gdb) next
```

Here, I want to see the next source line in the current (innermost) stack frame. Therefore, I use next in GDB.

```
1163    int main(int argc, char **argv, char **envp) {
(gdb) next
1164      atexit(llvm_shutdown);  // Call llvm_shutdown() on exit.
(gdb)
1166      llvm::InitializeNativeTarget();
(gdb)
1168      parseArguments(argc, argv);
(gdb)
1169      sys::PrintStackTraceOnErrorSignal();
(gdb)
1171      if (Watchdog) {
(gdb)
1237      sys::SetInterruptFunction(interrupt_handle);
(gdb)
1240      std::string ErrorMsg;
(gdb)
1241      LLVMContext ctx;
(gdb)
1242      Module *mainModule = 0;
(gdb)
1244      OwningPtr<MemoryBuffer> BufferPtr;
(gdb)
1245      error_code ec=MemoryBuffer::getFileOrSTDIN(InputFile.c_str(), BufferPt
r);
(gdb)
1246      if (ec) {
(gdb)
1251      mainModule = getLazyBitcodeModule(BufferPtr.get(), ctx, &ErrorMsg);
```

I keep move on, and I find this line of code can locate and output path and folder.

```
390       IOpts.MakeConcreteSymbolic = MakeConcreteSymbolic;
gdb)
391       KleeHandler *handler = new KleeHandler(pArgc, pArgv);
gdb)
LEE: output directory is "/home/weikun/Desktop/examples/get_sign/klee-out-4"
393         theInterpreter = Interpreter::create(ctx, IOpts, handler);
gdb)
```

Next, I find KLEE generates 3 path to check function (the original function have 3 conditions).

```
(gdb)
KLEE: Using STP solver backend
1394      handler->setInterpreter(interpreter);
(gdb)
1396      for (int i=0; i<argc; i++) {
(gdb)
1397        handler->getInfoStream() << argv[i] << (i+1<argc ? " ":"\n");
(gdb)
1396      for (int i=0; i<argc; i++) {
(gdb)
1397        handler->getInfoStream() << argv[i] << (i+1<argc ? " ":"\n");
(gdb)
1396      for (int i=0; i<argc; i++) {
(gdb)
1399      handler->getInfoStream() << "PID: " << getpid() << "\n";
(gdb)
1402        interpreter->setModule(mainModule, Opts);
(gdb)
```

Next execute instruction will create the assembly.ll and store it into output folder.

```
(gdb)
1403       externalsAndGlobalsCheck(finalModule);
(gdb)
```

Base on the assembly.ll. KLEE will generate 3 test cases for each path.

```
504         interpreter->runFunctionAsMain(mainFn, pArgc, pArgv, pEnvp);
gdb)
506         while (!seeds.empty()) {
```

n. References:

[1] Richard Stallman, Roland Pesch, Stan Shebs, *Debugging with GDB: the GNU source-level debugger,*