Anatomy of a Dyninst Program

A Memory Tracing Tool

Dyninst can run in three execution modes

- **Rewriter Mode: Open a binary, instrument and write to a new binary**
- **Attach Mode: Attach to a running process, instrument and continue execution**
- **Create Mode: Create a new process, instrument and continue execution**

Instrumentation is in the form of Abstract Syntax Trees (ASTs)

- **Instrumentation calls an external library function**

Dyninst can parse & analyze binary code

- **Find all the functions in a binary**
- **Generate control flow graph (CFG) for the functions and find all the basic blocks in the CFG**
- **Find all the load and store instructions in the basic block**

Dyninst can insert instrumentation at any point in the binary code

- **Insert instrumentation at every load and store instruction**

Dyninst can monitor & control processes

- **Continue execution of a stopped process**
- **Check if the process has terminated**
- **Wait for the process to change execution status**

```
// include files left out for brevity
int main (int argc, const char *argv[]) {
   BPatch *bpatch = new BPatch; BPatch_addressSpace *addSpace;
   enum {binary_rewriter, attach, create} runmode;
   switch (argv[1][1]) {
      case 'b': runmode = binary_rewriter; break;
      case 'c': runmode = create; break;
      case 'a': runmode = attach; break;
   }
   const char *mutatee = argv[2];

   // Open a binary or create/attach to a process.
   if (runmode == binary_rewriter)
      addSpace = bpatch->openBinary(mutatee);                    // input binary file
   else if (runmode == attach)
      addSpace = bpatch->processAttach(mutatee, atoi(argv[3])); // input process and pid
   else if (runmode == create)
      addSpace = bpatch->processCreate(mutatee, argv+3);         // input executable and arguments

    // Find all the functions in the binary
   BPatch_Vector<BPatch_function *>*funcs = addSpace->getImage()->getProcedures();

   // Load libTraceMemory.so
   // Find "void printLoadMemoryAccess(void *addr)" and "void printStoreMemoryAccess(void *addr)" .
   // These function calls will be instrumented at load and store instructions.
   // Generate argument for the functions  - Effective Address of the instrumented instruction
   addSpace->loadLibrary("libTraceMemory.so");
   BPatch_Vector<BPatch_snippet *> args;
   args.push_back(new BPatch_effectiveAddressExpr());
   BPatch_Vector<BPatch_function *> instCallFuncs;
   addSpace->getImage()->findFunction("printLoadMemoryAccess", instCallFuncs);
   BPatch_function *instLoadCallFunc = instCallFuncs[0];
   BPatch_funcCallExpr callLoadPrint(*instLoadCallFunc, args);
   addSpace->getImage()->findFunction("printStoreMemoryAccess", instCallFuncs);
   BPatch_function *instStoreCallFunc = instCallFuncs[1];
   BPatch_funcCallExpr callStorePrint(*instStoreCallFunc, args);

   // For each function, for each basic block, instrument the load and store instructions in the block.
   for (BPatch_Vector<BPatch_function*>::iterator i = funcs->begin(); i != funcs->end(); i++){
      set<BPatch_basicBlock *> blocks;
      (*i)->getCFG()->getAllBasicBlocks(blocks);
      for (set<BPatch_basicBlock *>::iterator j = blocks.begin(); j != blocks.end(); j++){
         BPatch_Set <BPatch_opCode> loadOpcode; loadOpcode.insert (BPatch_opLoad);
         BPatch_Vector <BPatch_point *> *instLoadPoints = (*j)->findPoint(loadOpcode);
         addSpace->insertSnippet(callLoadPrint, *instLoadPoints);
         BPatch_Set <BPatch_opCode> storeOpcode; storeOpcode.insert (BPatch_opStore);
         BPatch_Vector <BPatch_point *> *instStorePoints = (*j)->findPoint(storeOpcode);
         addSpace->insertSnippet(callStorePrint, *instStorePoints);
      }
   }

   // If rewriter mode, rewrite instrumented binary to a new binary,
   // If dynamic instrumentation, continue execution and  wait for the process to terminate
   if (runmode == binary_rewriter)
      dynamic_cast<BPatch_binaryEdit *>(addSpace)->writeFile(argv[3]);
   else {
      dynamic_cast<BPatch_process *>(addSpace)->continueExecution();
      while (!dynamic_cast<BPatch_process *>(addSpace)->isTerminated())
         bpatch->waitForStatusChange();
   }
   return 0;
}
```

A *mutator* program for printing a trace of all memory accesses in a *mutatee* program

xterm

```
> ./traceMemory -b hello hello-rewritten
> ./hello-rewritten
      Load  Memory Access at 0xbf82c218
      Load  Memory Access at 0xbf82c21c
      Store Memory Access at 0xbf82c21c
      Store Memory Access at 0xbf82c218
      Load  Memory Access at 0xbf82c210
      Load  Memory Access at 0xbf82c21c
      Store Memory Access at 0xbf82c21c
      ….
   Hello World
      …..

> ./traceMemory -c hello
      Load  Memory Access at 0xbf981db8
      Load  Memory Access at 0xbf981dbc
      Store Memory Access at 0xbf981dbc
      Store Memory Access at 0xbf981db8
      …
   Hello World
      …
>
```