Appendix A

Handling stack operations in Memlock

The following code A.1 is a shortend version of how memlock handles stack operations. In **line 6**, we can see that the `AFLCoverage` class is derived from a `ModulePass`, which runs a function pass over each function in the module. Using this pass, we can run over all instructions of IR. On **line 20** we define the pointer to the performance's array. In **lines 27** and **28**, we focus on each basic block; as a result, Memlock adds the instrumentations in each basic block. After loading the pointers that were defined earlier, on **line 53** Memlock defines **PerfTotalCounter**, for updating the corresponding pointer and in **line 54**, Memlock increments the counter of stack by one, if it is the first time we are visiting the current basic block. To figure out the type of instructions and find the *return call instruction*, Memlock iterates over all instructions of the current basic block and compares the opcode of the instruction, with the opcode defined for return instruction (**line 58**). If the return instruction is found, the stack counter will be decreased by one.
Note that the following code is not the exact code and the source code can be found on Memlock project [22]. Memlock handles the stack and heap operations in seperate

files, doing different instrumentations for each of these two approaches.

```cpp
// required packages are included

using namespace llvm;

namespace {
  class AFLCoverage : public ModulePass {
    public:
      static char ID;
      AFLCoverage() : ModulePass(ID) { }

      bool runOnModule(Module &M) override;
      }
  };
}

bool AFLCoverage::runOnModule(Module &M) {
  LLVMContext &C = M.getContext();
  llvm::IRBuilder<> builder(context);

  GlobalVariable *AFLPerfPtr =
      new GlobalVariable(M, PointerType::get(Int32Ty, 0), false,
                             GlobalValue::ExternalLinkage, 0, "
    __afl_perf_ptr");

  // To convert the range of EdgeID
  ConstantInt* PerfMask = ConstantInt::get(Int32Ty, PERF_SIZE-1);

  for (auto &F : M) {
    for (auto &BB : F) {
      BasicBlock::iterator IP = BB.getFirstInsertionPt();
      IRBuilder<> IRB(&(*IP));

      /* Load SHM pointer */
      LoadInst *MapPtr = IRB.CreateLoad(AFLMapPtr);
      MapPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C
    , None));
      Value *MapPtrIdx =
          IRB.CreateGEP(MapPtr, EdgeId);

      /* Load the Performance Pointer */
      LoadInst *PerfPtr = IRB.CreateLoad(AFLPerfPtr);
      PerfPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(
    C, None));
      Value *PerfBranchPtr =
        IRB.CreateGEP(PerfPtr, IRB.CreateAnd(EdgeId, PerfMask));

      /* Increment performance counter for branch */
      LoadInst *PerfBranchCounter = IRB.CreateLoad(PerfBranchPtr);
      PerfBranchCounter->setMetadata(M.getMDKindID("nosanitize"),
    MDNode::get(C, None));
      Value *PerfBranchIncr = IRB.CreateAdd(PerfBranchCounter,
ConstantInt::get(Int32Ty, 1));
      IRB.CreateStore(PerfBranchIncr, PerfBranchPtr)
          ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));

      /* Increment performance counter for total count  */
      LoadInst *PerfTotalCounter = IRB.CreateLoad(PerfPtr); // Index 0 of the perf map
      PerfTotalCounter->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));
      Value *PerfTotalIncr = IRB.CreateAdd(PerfTotalCounter, ConstantInt::get(Int32Ty, 1));
      IRB.CreateStore(PerfTotalIncr, PerfPtr)
          ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));

      for(BasicBlock::iterator i = BB.begin(), i2 = BB.end(); i!=i2; i++) {
        IRBuilder<> MemFuzzBuilder(&(*i));

        if(Instruction *inst = dyn_cast<Instruction>(i)) {
          if(inst->getOpcode() == Instruction::Ret)
          {
            /* Subtract the memory usage of stack after return */
          }
        }
      }
    }
  }
  return true;
}
```

Listing A.1: Memlock C++ code for stack operations