KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

阅读报告

汇报人：Fishermanykx, liukunlin123

June 19, 2021

Computer Science Institute

Table of Contents

Table of Contents

Introduction

Table of Contents

What is DSE

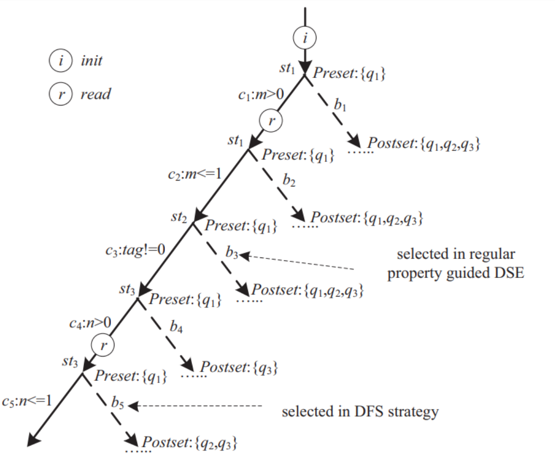

Figure 1: DSE

What is DSE

- Dynamic symbolic execution (DSE) enhances traditional symbolic execution by combing concrete execution and symbolic execution.

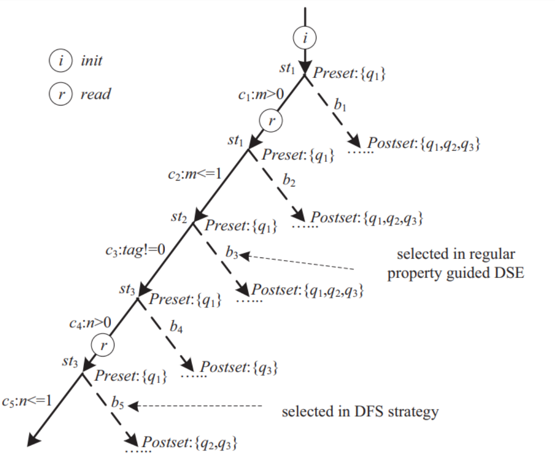

Figure 1: DSE

What is DSE

- Dynamic symbolic execution (DSE) enhances traditional symbolic execution by combing concrete execution and symbolic execution.
- DSE repeatedly runs the program both concretely and symbolically. After each run, all the branches off the execution path, called the off-path-branches, are collected, and then one of them is selected to generate new inputs for the next run to explore a new path

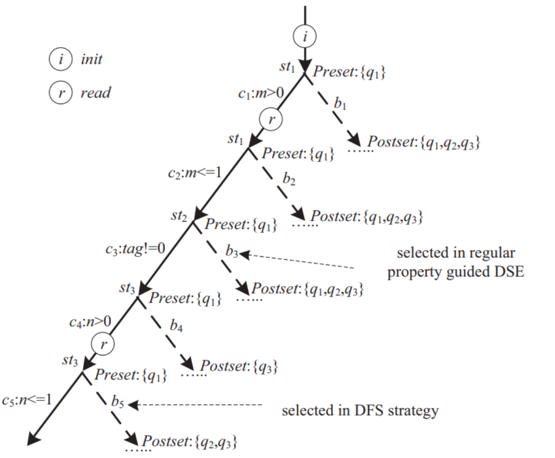

Figure 1: DSE

What is KLEE

What is KLEE?

- KLEE is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure
- KLEE can also be used as a bug finding tool

Table of Contents

Contributions

This paper makes two contributions.

1. It presents a new symbolic execution tool, KLEE, which they designed for robust, deep checking of a broad range of applications, leveraging several years of lessons from their previous tool, EXE
2. It shows that KLEE’s automatically-generated tests get high coverage on a diverse set of real, complicated, and environmentally-intensive programs.

KLEE’ s preformance

KLEE’s preformance

- KLEE gets high coverage on a broad set of complex programs.
- KLEE can get significantly more code coverage than a concentrated, sustained manual effort.
- With one exception, KLEE achieved these highcoverage results on unaltered applications.
- KLEE finds important errors in heavily-tested code.
- The fact that KLEE test cases can be run on the raw version of the code (e.g., compiled with gcc) greatly simplifies debugging and error reporting.
- When used to crosscheck purportedly identical BUSYBOX and GNU COREUTILS tools, it automatically found functional correctness errors and a myriad of inconsistencies.
- KLEE can also be applied to non-application code.

Table of Contents

Basic Architecture

- The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state.
- Storage locations for a state —registers, stack and heap objects —refer to **expressions** (trees) instead of raw data values.
- The **leaves** of an expression are **symbolic variables or constants**, and the **interior nodes** come from **LLVM assembly language operations**.

Basic Architecture

- **Conditional branches** take a boolean expression (branch condition) and alter the instruction pointer of the state based on whether the condition is true or false.
- **Potentially dangerous operations** implicitly generate branches that check if any input value exists that could cause an error.
- **Load and store instructions** generate checks: in this case to check that the address is in-bounds of a valid memory object
- **If a pointer can refer to many objects**, when a dereferenced pointer p can refer to N objects, KLEE clones the current state N times.

Code Review

Table of Contents

This is the general struct of KLEE

```
|-- include //包含公共的头文件
|-- tools //所有KLEE的二进制的main都在这里，有些是python脚本
|-- lib //包含大部分的源码
|  |-- Core //包含解释和执行LLVM字节码和KLEE内存模型。
|  |-- Expr // klee的表达式库
|  |-- Module //包含在执行LLVM字节码前的一些操作代码。比如链接POSIX运行函数等等。
|  |-- Solver//包含所有求解器
|-- runtime//包含各种KLEE的运行时支持。
|-- test//包含一些小的C程序和LLVM的字节码，用来给KLEE做回归测试
```

Figure 2: General Struct

Table of Contents

Main Execution Flow

This is the main flow of execution of KLEE, source code can be seen at lib/Core/Executor.cpp:2954 Searcher->selectState selects a state and returns to the variable *state*, then function *stepInstruction* points pc (program counter) to instruction to be executed. After this, function *executeInstruction* executes this Instruction according to its type.

```
while (!states.empty() && !haltExecution) {
  ExecutionState &state = searcher->
        selectState();
  KInstruction *ki = state.pc;
  stepInstruction(state);

  executeInstruction(state, ki);
  timers.invoke();
  if (::dumpStates) dumpStates();
  if (::dumpPTree) dumpPTree();

  checkMemoryUsage();

  updateStates(&state);
}
```

Table of Contents

Branch (Br)

Br Instruction has to types: unconditional branch (function as goto in C) and conditional branch.
In unconditional case, all we have to do is to transfer BasicBlock from current BasicBlock to its successor.

```
case Instruction::Br: {
  BranchInst *bi = cast<BranchInst>(i);
  if (bi->isUnconditional()) {
    transferToBasicBlock(bi->getSuccessor(0), bi->
          getParent(), state);
  }
```

Branch (Br)

In conditional case, we will

- fork the current state and add corresponding condition to the constraint set.

```
case Instruction::Br: {
  /* ... */
  } else {
    assert(bi->getCondition() == bi->getOperand(0) && "Wrong operand index!");
    ref<Expr> cond = eval(ki, 0, state).value;

    cond = optimizer.optimizeExpr(cond, false);
    Executor::StatePair branches = fork(state, cond, false);

    if (statsTracker && state.stack.back().kf->trackCoverage)
      statsTracker->markBranchVisited(branches.first, branches.second);

    if (branches.first)
      transferToBasicBlock(bi->getSuccessor(0), bi->getParent(), *branches.first);
    if (branches.second)
      transferToBasicBlock(bi->getSuccessor(1), bi->getParent(), *branches.second);
  }
  break;
}
```

Branch (Br)

In conditional case, we will

- send the constraint sets to solver (STP, Z3 etc.) to determine if the branch condition is either provably true (return trueState) or provably false (return falseState).

```
case Instruction::Br: {
  /* ... */
  } else {
    assert(bi->getCondition() == bi->getOperand(0) && "Wrong operand index!");
    ref<Expr> cond = eval(ki, 0, state).value;

    cond = optimizer.optimizeExpr(cond, false);
    Executor::StatePair branches = fork(state, cond, false);

    if (statsTracker && state.stack.back().kf->trackCoverage)
      statsTracker->markBranchVisited(branches.first, branches.second);

    if (branches.first)
      transferToBasicBlock(bi->getSuccessor(0), bi->getParent(), *branches.first);
    if (branches.second)
      transferToBasicBlock(bi->getSuccessor(1), bi->getParent(), *branches.second);
  }
  break;
}
```

Branch (Br)

In conditional case, we will

- check if both branches are all provably true, and transfer the BasicBlocks by function transferToBasicBlock

```
case Instruction::Br: {
  /* ... */
  } else {
    assert(bi->getCondition() == bi->getOperand(0) && "Wrong operand index!");
    ref<Expr> cond = eval(ki, 0, state).value;

    cond = optimizer.optimizeExpr(cond, false);
    Executor::StatePair branches = fork(state, cond, false);

    if (statsTracker && state.stack.back().kf->trackCoverage)
      statsTracker->markBranchVisited(branches.first, branches.second);

    if (branches.first)
      transferToBasicBlock(bi->getSuccessor(0), bi->getParent(), *branches.first);
    if (branches.second)
      transferToBasicBlock(bi->getSuccessor(1), bi->getParent(), *branches.second);
  }
  break;
}
```

Zero division check

Instrument function klee_div_zero_check before instructions like sdiv when the divisor of the instruction is 0.

```
void klee_div_zero_check(long long z) {
  if (z == 0)
    klee_report_error(__FILE__, __LINE__, "divide by zero", "div.err");
}
```

```
  LLVMContext &ctx = M.getContext();
  KleeIRMetaData md(ctx);
  auto divZeroCheckFunction = M.getOrInsertFunction("klee_div_zero_check", Type::getVoidTy(ctx), Type::getInt64Ty(ctx) KLEE_LLVM_GOIF_TERMINATOR);

  for (auto &divInst : divInstruction) {
    llvm::IRBuilder<> Builder(divInst /* Inserts before divInst*/);
    auto denominator = Builder.CreateIntCast(divInst->getOperand(1), Type::getInt64Ty(ctx),
                                  false, "int_cast_to_i64");
    Builder.CreateCall(divZeroCheckFunction, denominator);
    md.addAnnotation(*divInst, "klee.check.div", "True");
```

Boundary check

When encountering an array operation instruction such as Store, Load, etc., klee will check the array boundary overflow

```
ref<Expr> getBoundsCheckOffset(ref<Expr> offset,
      unsigned bytes) const {
  if (bytes<=size) {
    return UltExpr::create(offset, ConstantExpr::
          alloc(size - bytes + 1, Context::get().
          getPointerWidth()));
  } else {
    return ConstantExpr::alloc(0, Expr::Bool);
  }
}
```

Table of Contents

Scheduling

```
Searcher *getNewSearcher(Searcher::CoreSearchType type, Executor &executor) {
  Searcher *searcher = NULL;
  switch (type) {
  case Searcher::DFS: searcher = new DFSSearcher(); break;
  case Searcher::BFS: searcher = new BFSSearcher(); break;
  case Searcher::RandomState: searcher = new RandomSearcher(); break;
  case Searcher::RandomPath: searcher = new RandomPathSearcher(executor); break;
  case Searcher::NURS_CovNew: searcher = new WeightedRandomSearcher(
        WeightedRandomSearcher::CoveringNew); break;
  case Searcher::NURS_MD2U: searcher = new WeightedRandomSearcher(
        WeightedRandomSearcher::MinDistToUncovered); break;
  case Searcher::NURS_Depth: searcher = new WeightedRandomSearcher(
        WeightedRandomSearcher::Depth); break;
  case Searcher::NURS_RP: searcher = new WeightedRandomSearcher(WeightedRandomSearcher
        ::RP); break;
  case Searcher::NURS_ICnt: searcher = new WeightedRandomSearcher(
        WeightedRandomSearcher::InstCount); break;
  case Searcher::NURS_CPICnt: searcher = new WeightedRandomSearcher(
        WeightedRandomSearcher::CPInstCount); break;
  case Searcher::NURS_QC: searcher = new WeightedRandomSearcher(WeightedRandomSearcher
        ::QueryCost); break;
  }

  return searcher;
}
```

Random Path Selection

States are selected by traversing this tree from the root and randomly selecting the path to follow at branch points. Therefore, when a branch point is reached, the set of states in each subtree has equal probability of being selected, regardless of the size of their subtrees.

```
ExecutionState &RandomPathSearcher::selectState() {
  unsigned flips=0, bits=0;
  PTreeNode *n = executor.processTree->root.get();
  while (!n->state) {
    if (!n->left) {
      n = n->right.get();
    } else if (!n->right) {
      n = n->left.get();
    } else {
      if (bits==0) {
        flips = theRNG.getInt32();
        bits = 32;
      }
      --bits;
      n = (flips&(1<<bits)) ? n->left.get() : n->
          right.get();
    }
  }

  return *n->state;
}
```

All kinds of searcher are implemented in lib/Core/Searcher.cpp. Below is the implementation of the Coverage-Optimized Search. States are preserved in a Red-Black tree, and its weight is computed according to the type attribute of the state. After a new state is generated, it's inserted in the RB Tree (namely DiscretePDF). After a state is selected (In Coverage-Optimized Search, it's randomly selected as well), the node is removed from the RB-Tree.

```
ExecutionState &WeightedRandomSearcher::selectState() {
  return *states->choose(theRNG.getDoubleL());
}
```

Coverage-Optimized Search

```
double WeightedRandomSearcher::getWeight(
      ExecutionState *es) {
  switch(type) {
  default:
  case Depth:
    return es->depth;
  case RP:
    return std::pow(0.5, es->depth);
  case InstCount: {
    uint64_t count = theStatisticManager->
          getIndexedValue(stats::
          instructions, es->pc->info->id);
    double inv = 1. / std::max((uint64_t) 1,
          count);
    return inv * inv;
  }
  case CPInstCount: {
    StackFrame &sf = es->stack.back();
    uint64_t count = sf.callPathNode->
          statistics.getValue(stats::
          instructions);
    double inv = 1. / std::max((uint64_t) 1,
          count);
    return inv;
  }
    case QueryCost:
    return (es->queryCost.toSeconds() < .1)
          ? 1. : 1./ es->queryCost.
          toSeconds();
  case CoveringNew:
  case MinDistToUncovered: {
    uint64_t md2u =
          computeMinDistToUncovered(es->pc,
          es->stack.back().
          minDistToUncoveredOnReturn);
    double invMD2U = 1. / (md2u ? md2u :
          10000);
    if (type==CoveringNew) {
      double invCovNew = 0.;
      if (es->instsSinceCovNew)
        invCovNew = 1. / std::max(1, (int)
            es->instsSinceCovNew - 1000);
      return (invCovNew * invCovNew +
          invMD2U * invMD2U);
    } else {
      return invMD2U * invMD2U;
    }
  }
  }
}
```

Table of Contents

Memory Organization

Use Alloca as an example. When instruction type is alloca, in function executeInstruction it turns to the *Alloca* case. In this case, a pointer is generated accroding to the size of the element. Then executeAlloc is called.

```cpp
case Instruction::Alloca: {
  AllocaInst *ai = cast<AllocaInst>(i);
  // KModule: 基于 llvm::module，但添加了其他 klee 相关信息
  unsigned elementSize =
    kmodule->targetData->getTypeStoreSize(ai->getAllocatedType());
  // 根据上面拿到的元素大小创建一个指针
  ref<Expr> size = Expr::createPointer(elementSize);
  if (ai->isArrayAllocation()) {
    ref<Expr> count = eval(ki, 0, state).value;
    count = Expr::createZExtToPointerWidth(count);
    size = MulExpr::create(size, count);
  }
  executeAlloc(state, size, true, ki);
  break;
}
```

Memory Organization

MemoryObject’s represent allocation sites in the program (calls to malloc, stack objects, global variables) and, at least conceptually, can be thought of as the unique name for the object allocated at that site. ObjectState’s are used to store the actual contents of a MemoryObject in a particular ExecutionState (but can be shared).

Memory Organization

```
size = toUnique(state, size);
if (ConstantExpr *CE = dyn_cast<ConstantExpr>(size)) {
  const llvm::Value *allocSite = state.prevPC->inst;
  if (allocationAlignment == 0) {
    allocationAlignment = getAllocationAlignment(allocSite);
  }
  MemoryObject *mo =
      memory->allocate(CE->getZExtValue(), isLocal, /*isGlobal=*/false, allocSite,
             allocationAlignment);
  if (!mo) {
    bindLocal(target, state,
              ConstantExpr::alloc(0, Context::get().getPointerWidth()));
  } else {
    ObjectState *os = bindObjectInState(state, mo, isLocal);
    if (zeroMemory) os->initializeToZero();
    else os->initializeToRandom();
    bindLocal(target, state, mo->getBaseExpr());

    if (reallocFrom) {
      unsigned count = std::min(reallocFrom->size, os->size);
      for (unsigned i=0; i<count; i++)
        os->write(i, reallocFrom->read8(i));
      state.addressSpace.unbindObject(reallocFrom->getObject());
    }
  }
}
```

Memory Organization

Each ExecutionState stores a mapping of MemoryObjects -> ObjectState using the AddressSpace data structure (implemented as an immutable tree so that copying is cheap and the shared structure is exploited). Each AddressSpace may ”own” some subset of the ObjectStates in the mapping. When an AddressSpace is duplicated it loses ownership of the ObjectState in the map. Any subsequent write to an ObjectState will create a copy of the object (AddressSpace::getWriteable). This is the COW mechanism (which gets used for all objects, not just globals).

```
ObjectState *Executor::bindObjectInState(ExecutionState &state, const MemoryObject *mo, bool isLocal, const Array *array) {
  ObjectState *os = array ? new ObjectState(mo, array) : new ObjectState(mo);
  state.addressSpace.bindObject(mo, os);

  if (isLocal)
    state.stack.back().allocas.push_back(mo);

  return os;
}
```

Memory Organization

From the point of view of the state and this mapping there is no distinction between stack, heap, and global objects. The only special handling for stack objects is that the MemoryObject is marked as isLocal and the MemoryObject is stored in the StackFrame alloca list. When the StackFrame is popped these objects are then unbound so that the state can no longer access the memory directly (references to the memory object may still remain in ReadExprs, but conceptually the actual memory is no longer addressable).

Table of Contents

Constraint independence

Constraint independence divides constraint sets into disjoint independent subsets. By explicitly tracking these subsets, KLEE can frequently eliminate irrelevant constraints

For example, given the constraint set $i < j, j < 20, k > 0$, a query of whether $i = 20$ just requires the first two constraints.

Constraint independence

```
IndependentElementSet getIndependentConstraints(const Query& query,
                      std::vector< ref<Expr> > &result) {
  IndependentElementSet eltsClosure(query.expr);
  std::vector< std::pair<ref<Expr>, IndependentElementSet> > worklist;
  for (const auto &constraint : query.constraints)
  worklist.push_back(std::make_pair(constraint, IndependentElementSet(constraint)));
  bool done = false;
  do {
    done = true;
    std::vector< std::pair<ref<Expr>, IndependentElementSet> > newWorklist;
    for (std::vector< std::pair<ref<Expr>, IndependentElementSet> >::iterator
    it = worklist.begin(), ie = worklist.end(); it != ie; ++it) {
        if (it->second.intersects(eltsClosure)) {
          if (eltsClosure.add(it->second))
          done = false;
          result.push_back(it->first);
        } else {
          newWorklist.push_back(*it);
        }
      }
    worklist.swap(newWorklist);
  } while (!done);
  return eltsClosure;
}
```

Constraint Set Simplification

Constraint Set Simplification : klee simplifies the constraint set by rewriting previous constraints when new equality constraints are added to the constraint set. The constraint $\mathrm{x} < 10$. followed by $\mathrm{x} = 5$, substituting the value for x into the first constraint simplifies it to true, which KLEE eliminates.

Constraint Set Simplification

```
ref<Expr> ConstraintManager::simplifyExpr(const ConstraintSet &constraints,
                    const ref<Expr> &e) {
  if (isa<ConstantExpr>(e))
    return e;
  std::map< ref<Expr>, ref<Expr> > equalities;
  for (auto &constraint : constraints) {
      if (const EqExpr *ee = dyn_cast<EqExpr>(constraint)) {
          if (isa<ConstantExpr>(ee->left)) {
              equalities.insert(std::make_pair(ee->right,ee->left));
              } else {
                equalities.insert(
                std::make_pair(constraint, ConstantExpr::alloc(1, Expr::Bool)));
              }
      } else {
          equalities.insert(
          std::make_pair(constraint, ConstantExpr::alloc(1, Expr::Bool)));
      }
  }

  return ExprReplaceVisitor2(equalities).visit(e);
}
```

Expression Rewriting

Expression Rewriting: simple arithmetic simplifications $(x + 0 = x)$, strength reduction $(x * 2^n = x << n)$, linear simplification $(2 * x - x = x)$.

```
bool ConstraintManager::rewriteConstraints(ExprVisitor &visitor) {
  ConstraintSet old;
  bool changed = false;

  std::swap(constraints, old);
  for (auto &ce : old) {
      ref<Expr> e = visitor.visit(ce);

      if (e!=ce) {
        addConstraintInternal(e); // enable further reductions
        changed = true;
      } else {
        constraints.push_back(ce);
      }
    }

  return changed;
}
```

Counter-example Cache

Redundant queries are frequent, and a simple cache is effective at eliminating a large number of them.The counter-example cache maps sets of constraints to counter-examples (i.e., variable assignments), along with a special sentinel used when a set of constraints has no solution.

- When a subset of a constraint set has no solution, then neither does the original constraint set.
- When a superset of a constraint set has a solution, that solution also satisfies the original constraint set.
- When a subset of a constraint set has a solution, it is likely that this is also a solution for the original set.

Counter-example Cache

```
bool CexCachingSolver::searchForAssignment(
      KeyType &key, Assignment *&result) {
   Assignment * const *lookup = cache.
        lookup(key);
   if (lookup) {
      result = *lookup;
      return true;
    }
   if (CexCacheTryAll) {
      // Look for a satisfying assignment
           for a superset, which is
           trivially an
      // assignment for any subset.
      Assignment **lookup = 0;
      if (CexCacheSuperSet)
      lookup = cache.findSuperset(key,
           NonNullAssignment());
      if (!lookup)
      lookup = cache.findSubset(key,
           NullAssignment());
      // If either lookup succeeded, then
           we have a cached solution.
      if (lookup) {
         result = *lookup;
         return true;
       }
// Otherwise, iterate through the set of
      current assignments to see if one
// of them satisfies the query.
      for (assignmentsTable_ty::iterator
           it = assignmentsTable.begin(),
      ie = assignmentsTable.end(); it !=
           ie; ++it) {
         Assignment *a = *it;
         if (a->satisfies(key.begin(),
              key.end())) {
            result = a;
            return true;
          }
        }
     }
   return false;
  }
```

Table of Contents

Solver

When encountering conditional Br instructions or supposed errors, a solver (STP, Z3, etc.) is called.

```
bool CexCachingSolver::getAssignment(const Query& query,
      Assignment *&result) {
  KeyType key;
  if (lookupAssignment(query, key, result))
    return true;

  std::vector<const Array*> objects;
  findSymbolicObjects(key.begin(), key.end(), objects);

  std::vector< std::vector<unsigned char> > values;
  bool hasSolution;
  if (!solver->impl->computeInitialValues(query, objects,
        values, hasSolution))
    return false;

  /* ...*/

  result = binding;
  cache.insert(key, binding);

  return true;
}
```

Call stack of solver

```
#0  CexCachingSolver::getAssignment (this=this@entry=0x2ae8d00, query=...,
    result=@0x7fffffffd530: 0x7fffffffd587)
    at /mnt/klee/klee/lib/Solver/CexCachingSolver.cpp:223
#1  0x000000000183b05b in CexCachingSolver::computeValidity (this=0x2ae8d00,
    query=..., result=@0x7fffffffd86c: klee::Solver::Unknown)
    at /mnt/klee/llvm-project-llvmorg-9.0.1/llvm/include/llvm/ADT/APInt.h:33
#2  0x0000000001839653 in CachingSolver::computeValidity (this=0x2c245d0,
    query=..., result=@0x7fffffffd86c: klee::Solver::Unknown)
    at /mnt/klee/klee/lib/Solver/CachingSolver.cpp:194
#3  0x000000000182c6a4 in IndependentSolver::computeValidity (this=0x2ad8190
    query=..., result=@0x7fffffffd86c: klee::Solver::Unknown)
    at /mnt/klee/klee/lib/Solver/IndependentSolver.cpp:416
#4  0x00000000004bcf47 in klee::TimingSolver::evaluate (this=0x2ad8170,
    constraints=..., expr=..., result=@0x7fffffffd86c: klee::Solver::Unknown
    metaData=...) at /mnt/klee/klee/lib/Core/TimingSolver.cpp:40
#5  0x0000000000486682 in klee::Executor::fork (this=this@entry=0x2cb8000,
    current=..., condition=..., isInternal=isInternal@entry=false)
    at /mnt/klee/klee/lib/Core/Executor.cpp:1045
#6  0x000000000048e580 in klee::Executor::executeInstruction (this=0x2cb8000
    state=..., ki=0x2cbf9f0) at /mnt/klee/klee/lib/Core/Executor.cpp:2161
#7  0x000000000049612a in klee::Executor::run (this=this@entry=0x2cb8000,
    initialState=...) at /mnt/klee/klee/lib/Core/Executor.cpp:3539
```

Table of Contents

Environment Modeling

Handle the environment by redirecting calls that access it to models that understand the semantics of the desired action well enough to generate the required constraints, these models are written in normal C code, 2,500 lines of code to define simple models for roughly 40 system calls

ALL shared libraries are initialized by __uClibc_main

Environment Modeling

```
int main(int argc, char **argv, char **envp) {
  switch (Libc) {
    case LibcType::UcLibc:
    linkWithUclibc(LibraryDir, opt_suffix, loadedModules);
    break;
}
linkWithUclibc(StringRef libDir, std::string opt_suffix,
    std::vector<std::unique_ptr<llvm::Module>> &modules) {
  for (auto i = newModules, j = modules.size(); i < j; ++i) {
    replaceOrRenameFunction(modules[i].get(), "__libc_open", "open");
    replaceOrRenameFunction(modules[i].get(), "__libc_fcntl", "fcntl");
  }
  createLibCWrapper(modules, EntryPoint, "__uClibc_main");
    } /* ... */
```

```
createLibCWrapper(std::vector<std::unique_ptr<llvm::Module>> &modules, llvm::StringRef intendedFunction, llvm::StringRef libcMainFunction) {
  llvm::Function *libcMainFn = nullptr;
  for (auto &module : modules) {
    if ((libcMainFn = module->getFunction(libcMainFunction)))
      break;
  }
  /* ... */
  BasicBlock *bb = BasicBlock::Create(ctx, "entry", stub);
  llvm::IRBuilder<> Builder(bb);
  /* ... */
  Builder.CreateCall(libcMainFn, args);
  Builder.CreateUnreachable();
}
```

Environment Modeling

For each file system operation we check if the action is for an actual concrete file on disk or a symbolic file. For concrete files, we simply invoke the corresponding system call in the running operating system. For symbolic files we emulate the operation's effect on a simple symbolic file system, private to each state.

```
ssize_t read(int fd, void *buf, size_t count) {
  if (is_invalid(fd)) {
    errno = EBADF;
    return -1;
  }
  struct klee_fd *f = &fds[fd];
  if (is_concrete_file(f)) {
    int r = pread(f->real_fd, buf, count, f->off);
    if (r != -1)
      f->off += r;
    return r;
  } else {
    /* sym files are fixed size: don't read beyond
        the end. */
    if (f->off >= f->size)
      return 0;
    count = min(count, f->size - f->off);
    memcpy(buf, f->file data + f->off, count);
    f->off += count;
    return count;
  }
}
```

KLEE Usage

Table of Contents

Command line options

1. Compling to LLVM bitcode: clang -I ../../include -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone pragram.c
2. General usage of KLEE:
   klee [klee-options] <program.bc> [program-options]
3. Set main search heuristics:
   klee --search=random-state --search=nurs:md2u program.o

Other options can be seen at
https://klee.github.io/docs/options/

The End

Thank you

Thank you for listening!

Questions?