18:08 A Guide to KLEE LLVM Execution Engine Internals

by Julien Vanegue

Greetings fellow neighbors!

It is my great pleasure to finally write my first article in PoC||GTFO after so many of you have contributed excellent content in the past dozens of issues that Pastor Laphroig put together for our enjoyment. I have been waiting for this moment for some time, and been harassed a few times, to finally come up with something worthwhile. Given the high standards set upon all of us, I did not feel like rushing it. Instead, I bring to you today what I think will be a useful piece of texts for many fellow hackers to use in the future. Apologies for any errors that may have slipped from my understanding, I am getting older after all, and my memory is not what it used to be. Not like it has ever been infaillible but at least I used to remember where the cool kids hung out. This is my attempt at renewing the tradition of sharing knowledge through some more informal channels.

Today, I would like to talk to you about KLEE, an open source symbolic execution engine originally developed at Stanford University and now maintained at Imperial College in London. Symbolic Execution (SYMEX) stands somewhere between static analysis of programs and [dynamic] fuzz testing. While its theoretical foundations dates back from the late seventies (King's paper), practical application of it waited until the late 2000s (such as SAGE[40] at Microsoft Research) to finally become mainstream with KLEE in 2008. These tools have been used in practice to find thousands of security issues in software, going from simple `NULL` pointer dereferences, to out of bound reads or writes for both the heap and the stack, including use-after-free vulnerabilities and other type-state issues that can be easily defined using "asserts."

In one hand, symbolic execution is able to undergo concrete execution of the analyzed program and maintains a concrete store for variable values as the execution progresses, but it can also track path conditions using constraints. This can be used to verify the feasibility of a specific path. At the same time, a process tree (`PTree`) of nodes (`PTreeNode`) represent the state space as an `ImmutableTree` structure. The `ImmutableTree` implements a copy-on-write mechanism so that parts of the state (mostly variable values) that are shared across the node don't have to be copied from state to state unless they are written to. This allows KLEE to scale better under memory pressure. Such state contains both a list of symbolic constraints that are known to be true in this state, as well as a concrete store for program variables on which constraints may or may not be applied (but that are nonetheless necessary so the program can execute in KLEE).

My goal in this article is not so much to show you how to use KLEE, which is well understood, but bring you a tutorial on hacking KLEE internals. This will be useful if you want to add features or add support for specific analysis scenarios that you care about. I've spent hundreds of hours in KLEE internals and having such notes may have helped me in the beginning. I hope it helps you too.

Now let's get started.

Working with Constraints

Let's look at the simple C program as a motivator.

```
int fct(int a, int b) {
  int c = 0;
  if (a < b)
    c++;
  else
    c--;
  return c;
}

int main(int argc, char **argv) {
  if (argc != 3) return (-1);
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  if (a < b)
    return (0);
  return fct(a, b);
}
```

It is clear that the path starting in `main` and continuing in the first `if (a < b)` is infeasible. This is because any such path will actually have finished with a `return (0)` in the `main` function already. The way KLEE can track this is by listing constraints for the path conditions.

This is how it works: first KLEE executes some bootstrapping code before `main` takes control, then

[40] `unzip pocorgtfo18.pdf automatedwhiteboxfuzzing.pdf`

starts executing the first LLVM instruction of the `main` function. Upon reaching the first `if` statement, KLEE forks the state space (via function `Executor::fork`). The left node has one more constraint (`argc != 3`) while the right node has constraint (`argc == 3`). KLEE eventually comes back to its main routine (`Executor::run`), adds the newly-generated states into the set of active states, and picks up a new state to continue analysis with.

Executor Class

The main class in KLEE is called the `Executor` class. It has many methods such as `Executor::run()`, which is the main method of the class. This is where the set of states: added states and removed states set are manipulated to decide which state to visit next. Bear in mind that nothing guarantees that next state in the `Executor` class will be the next state in the current path.

Figure 26 shows all of the LLVM instructions currently supported by KLEE.

- `Call/Br/Ret`: Control flow instructions. These are cases where the program counter (part of the state) may be modified by more than just the size of the current instruction. In the case of `Call` and `Ret`, a new object `StackFrame` is created where local variables are bound to the called function and destroyed on return. Defining new variables may be achieved through the KLEE API `bindObjectInState()`.

- `Add/Sub/Mul/*S*/U*/*Or*`: The Signed and Unsigned arithmetic instructions. The usual suspects including bit shifting operations as well.

- Cast operations (`UItoFP`, `FPtoUI`, `IntToPtr`, `PtrToInt`, `BitCast`, etc.): used to convert variables from one type to a variable of a different type.

- `*Ext*` instructions: these extend a variable to use a larger number of bits, for example 8b to 32b, sometimes carrying the sign bit or the zero bit.

- `F*` instructions: the floating point arithmetic instructions in KLEE. I dont myself do much floating point analysis and I tend not to modify these cases, however this is where to look if you're interested in that.

- `Alloca`: used to allocate memory of a desired size

- `Load/Store`: Memory access operations at a given address

- `GetElementPtr`: perform array or structure read/write at certain index

- `PHI`: This corresponds to the PHI function in the Static Single Assignment form (SSA) as defined in the literature.[41]

There are other instructions I am glossing over but you can refer to the LLVM reference manual for an exhaustive list.

So far the execution in KLEE has gone through `Executor::run() -> Executor::executeInstruction() -> case ...` but we have not looked at what these cases actually do in KLEE. This is handled by a class called the `ExecutionState` that is used to represent the state space.

ExecutionState Class

This class is declared in `include/klee/ExecutionState.h` and contains mostly two objects:

- `AddressSpace`: contains the list of all metadata for the process objects in this state, including global, local, and heap objects. The address space is basically made of an array of objects and routines to resolve concrete addresses to objects (via method `AddressSpace::resolveOne` to resolve one by picking up the first match, or method `AddressSpace::resolve` for resolving to a list of objects that may match). The `AddressSpace` object also contains a concrete store for objects where concrete values can be read and written to. This is useful when you're tracking a symbolic variable but suddently need to concretize it to make an external concrete function call in libc or some other library that you haven't linked into your LLVM module.

[41] `unzip pocorgtfo18.pdf cytron.pdf`

```
$ grep -rni 'case Instruction::' lib/Core/
lib/Core/Executor.cpp:2452:  case Instruction::Ret: {
lib/Core/Executor.cpp:2591:  case Instruction::Br: {
lib/Core/Executor.cpp:2619:  case Instruction::Switch: {
lib/Core/Executor.cpp:2731:  case Instruction::Unreachable:
lib/Core/Executor.cpp:2739:  case Instruction::Invoke:
lib/Core/Executor.cpp:2740:  case Instruction::Call: {
lib/Core/Executor.cpp:2987:  case Instruction::PHI: {
lib/Core/Executor.cpp:2995:  case Instruction::Select: {
lib/Core/Executor.cpp:3006:  case Instruction::VAArg:
lib/Core/Executor.cpp:3012:  case Instruction::Add: {
lib/Core/Executor.cpp:3019:  case Instruction::Sub: {
lib/Core/Executor.cpp:3026:  case Instruction::Mul: {
lib/Core/Executor.cpp:3033:  case Instruction::UDiv: {
lib/Core/Executor.cpp:3041:  case Instruction::SDiv: {
lib/Core/Executor.cpp:3049:  case Instruction::URem: {
lib/Core/Executor.cpp:3057:  case Instruction::SRem: {
lib/Core/Executor.cpp:3065:  case Instruction::And: {
lib/Core/Executor.cpp:3073:  case Instruction::Or: {
lib/Core/Executor.cpp:3081:  case Instruction::Xor: {
lib/Core/Executor.cpp:3089:  case Instruction::Shl: {
lib/Core/Executor.cpp:3097:  case Instruction::LShr: {
lib/Core/Executor.cpp:3105:  case Instruction::AShr: {
lib/Core/Executor.cpp:3115:  case Instruction::ICmp: {
lib/Core/Executor.cpp:3207:  case Instruction::Alloca: {
lib/Core/Executor.cpp:3221:  case Instruction::Load: {
lib/Core/Executor.cpp:3226:  case Instruction::Store: {
lib/Core/Executor.cpp:3234:  case Instruction::GetElementPtr: {
lib/Core/Executor.cpp:3289:  case Instruction::Trunc: {
lib/Core/Executor.cpp:3298:  case Instruction::ZExt: {
lib/Core/Executor.cpp:3306:  case Instruction::SExt: {
lib/Core/Executor.cpp:3315:  case Instruction::IntToPtr: {
lib/Core/Executor.cpp:3324:  case Instruction::PtrToInt: {
lib/Core/Executor.cpp:3334:  case Instruction::BitCast: {
lib/Core/Executor.cpp:3343:  case Instruction::FAdd: {
lib/Core/Executor.cpp:3358:  case Instruction::FSub: {
lib/Core/Executor.cpp:3372:  case Instruction::FMul: {
lib/Core/Executor.cpp:3387:  case Instruction::FDiv: {
lib/Core/Executor.cpp:3402:  case Instruction::FRem: {
lib/Core/Executor.cpp:3417:  case Instruction::FPTrunc: {
lib/Core/Executor.cpp:3434:  case Instruction::FPExt: {
lib/Core/Executor.cpp:3450:  case Instruction::FPToUI: {
lib/Core/Executor.cpp:3467:  case Instruction::FPToSI: {
lib/Core/Executor.cpp:3484:  case Instruction::UIToFP: {
lib/Core/Executor.cpp:3500:  case Instruction::SIToFP: {
lib/Core/Executor.cpp:3516:  case Instruction::FCmp: {
lib/Core/Executor.cpp:3608: case Instruction::InsertValue: {
lib/Core/Executor.cpp:3635: case Instruction::ExtractValue: {
lib/Core/Executor.cpp:3645:  case Instruction::Fence: {
lib/Core/Executor.cpp:3649:  case Instruction::InsertElement: {
lib/Core/Executor.cpp:3691:  case Instruction::ExtractElement: {
lib/Core/Executor.cpp:3724:  case Instruction::ShuffleVector:
```

Figure 26. LLVM Instructions supported by KLEE

- `ConstraintManager`: contains the list of all symbolic constraints available in this state. By default, KLEE stores all path conditions in the constraint manager for that state, but it can also be used to add more constraints of your choice. Not all objects in the `AddressSpace` may be subject to constraints, which is left to the discretion of the KLEE programmer. Verifying that these constraints are satisfiable can be done by calling `solver->mustBeTrue()` or `solver->MayBeTrue()` methods, which is a solver-independent API provided in KLEE to call SMT or Z3 independently of the low-level solver API. This comes handy when you want to check the feasibility of certain variable values during analysis.

Every time the `::fork()` method is called, one execution state is split into two where possibly more constraints or different values have been inserted in these objects. One may call the `Executor::branch()` method directly to create a new state from the existing state without creating a state pair as fork would do. This is useful when you only want to add a subcase without following the exact fork expectations.

Executor::executeMemoryOperation(), MemoryObject and ObjectState

Two important classes in KLEE are `MemoryObject` and `ObjectState`, both defined in `lib/klee/-Core/Memory.h`.

The `MemoryObject` class is used to represent an object such as a buffer that has a base address and a size. When accessing such an object, typically via the `Executor::executeMemoryOperation()` method, KLEE automatically ensures that accesses are in bound based on known base address, desired offset, and object size information. The `MemoryObject` class provides a few handy methods:

```
(...)
ref<ConstantExpr> getBaseExpr()
ref<ConstantExpr> getSizeExpr()
ref<Expr> getOffsetExpr(ref<Expr> pointer)
ref<Expr> getBoundsCheckPointer(
          ref<Expr> pointer)
ref<Expr> getBoundsCheckPointer(
          ref<Expr> pointer, unsigned bytes)
ref<Expr> getBoundsCheckOffset(
          ref<Expr> offset)
ref<Expr> getBoundsCheckOffset(
          ref<Expr> offset, unsigned bytes)
```

Using these methods, checking for boundary conditions is child's play. It becomes more interesting when symbolics are used as the conditions that must be checked involves more than constants, depending on whether the base address, the offset or the index are symbolic values (or possibly depending on the source data for certain analyses, for example taint analysis).

While the `MemoryObject` somehow takes care of the spatial integrity of the object, the `ObjectState` class is used to access the memory value itself in the state. Its most useful methods are:

```
// return bytes read.
ref<Expr> read(ref<Expr> offset,
               Expr::Width width);
ref<Expr> read(unsigned offset,
               Expr::Width width);
ref<Expr> read8(unsigned offset);

// return bytes written.
void write(unsigned offset,
           ref<Expr> value);
void write(ref<Expr> offset,
           ref<Expr> value);
void write8(unsigned offset,
           uint8_t value);
void write16(unsigned offset,
           uint16_t value);
void write32(unsigned offset,
           uint32_t value);
void write64(unsigned offset,
           uint64_t value);
```

Objects can be either concrete or symbolic, and these methods implement actions to read or write the object depending on this state. One can switch from concrete to symbolic state by using methods:

```
void makeConcrete();
void makeSymbolic();
```

These methods will just flush symbolics if we become concrete, or mark all concrete variables as symbolics from now on if we switch to symbolic mode. Its good to play around with these methods to see what happens when you write the value of a variable, or make a new variable symbolic and so on.

When `Instruction::Load` and `::Store` are encountered, the `Executor::executeMemory-Operation()` method is called where symbolic array bounds checking is implemented. This implementation uses a mix of `MemoryObject`, `ObjectState`, `AddressSpace::resolveOne()` and

`MemoryObject::getBoundsCheckOffset()` to figure out whether any overflow condition can happen. If so, it calls KLEE's internal API `Executor::terminateStateOnError()` to signal the memory safety issue and terminate the current state. Symbolic execution will then resume on other states so that KLEE does not stop after the first bug it finds. As it finds more errors, KLEE saves the error locations so it won't report the same bugs over and over.

Special Function Handlers

A bunch of special functions are defined in KLEE that have special handlers and are not treated as normal functions. See `lib/Core/SpecialFunctionHandler.cpp`.

Some of these special functions are called from the `Executor::executeInstruction()` method in the case of the `Instruction::Call` instruction.

All the `klee_*` functions are internal KLEE functions which may have been produced by annotations given by the KLEE analyst. (For example, you can add a `klee_assume(p)` somewhere in the analyzed program's code to say that `p` is assumed to be true, thereby some constraints will be pushed into the `ConstraintManager` of the currenet state without checking them.) Other functions such as malloc, free, etc. are not treated as normal function in KLEE. Because the malloc size could be symbolic, KLEE needs to concretize the size according to a few simplistic criteria (like `size = 0`, `size` = 2^8, `size` = 2^{16}, etc.) to continue making progress. Suffice to say this is quite approximate.

This logic is implemented in the `Executor::executeAlloc()` and `::executeFree()` methods. I have hacked around some modifications to track the heap more precisely in KLEE, however bear in mind that KLEE's heap as well as the target program's heap are both maintained within the same address space, which is extremely intrusive. This makes KLEE a bad framework for layout sensitive analysis, which many exploit generation problems require nowadays. Other special functions include stubs for Address Sanitizer (ASan), which is now included in LLVM and can be enabled while creating LLVM code with clang. ASan is mostly useful for fuzzing so normally invisible corruptions turn into visible assertions. KLEE does not make much use of these stubs and mostly generate a warning if you reach one of the ASan-defined stubs.

Other recent additions were `klee_open_merge()` and `klee_close_merge()` that are an annotation mechanism to perform selected merging in KLEE. Merging happens when you come back from a conditional contruct (e.g., switch, or when you must define whether to continue or break from a loop) as you must select which constraints and values will hold in the state immediately following the merge. KLEE has some interesting merging logic implemented in `lib/Core/MergeHandler.cpp` that are worth taking a look at.

Experiment with KLEE for yourself!

I did not go much into details of how to install KLEE as good instructions are available onine.[42] Try it for yourself!

I personally use LLVM 3.4 mostly but KLEE also supports LLVM 3.5 reliably, although as far as I know 3.4 is still recommended.

My setup is an amd64 machine on Ubuntu 16.04 that has most of what you will need in packages. I recommend building LLVM and KLEE from sources as well as all dependencies (e.g., Z3[43] and/or STP[44]) that will help you avoid weird symbol errors in your experiments.

A good first target to try KLEE on is `coreutils`, which is what prettty much everybody uses in their research papers evaluation nowadays. Coreutils is well tested so new bugs in it are scarce, but its good to confirm everything works okay for you. A tutorial on how to run KLEE on coreutils is available as part of the project website.[45]

I personally used KLEE on various targets: coreutils, busybox, as well as other standard network tools that take input from untrusted data. These will require a standalone research paper explaining how KLEE can be used to tackle these targets.

[42] `http://klee.github.io/build-llvm34/`
[43] `unzip pocorgtfo18.pdf z3.pdf`
[44] `unzip pocorgtfo18.pdf stp.pdf`
[45] `http://klee.github.io/docs/coreutils-experiments/`

```
$ grep -in add\( lib/Core/SpecialFunctionHandler.cpp
66:#define add(name, handler, ret) { name, \
81:  add("calloc", handleCalloc, true),
82:  add("free", handleFree, false),
83:  add("klee_assume", handleAssume, false),
84:  add("klee_check_memory_access", handleCheckMemoryAccess, false),
85:  add("klee_get_valuef", handleGetValue, true),
86:  add("klee_get_valued", handleGetValue, true),
87:  add("klee_get_valuel", handleGetValue, true),
88:  add("klee_get_valuell", handleGetValue, true),
89:  add("klee_get_value_i32", handleGetValue, true),
90:  add("klee_get_value_i64", handleGetValue, true),
91:  add("klee_define_fixed_object", handleDefineFixedObject, false),
92:  add("klee_get_obj_size", handleGetObjSize, true),
93:  add("klee_get_errno", handleGetErrno, true),
94:  add("klee_is_symbolic", handleIsSymbolic, true),
95:  add("klee_make_symbolic", handleMakeSymbolic, false),
96:  add("klee_mark_global", handleMarkGlobal, false),
97:  add("klee_open_merge", handleOpenMerge, false),
98:  add("klee_close_merge", handleCloseMerge, false),
99:  add("klee_prefer_cex", handlePreferCex, false),
100:  add("klee_posix_prefer_cex", handlePosixPreferCex, false),
101:  add("klee_print_expr", handlePrintExpr, false),
102:  add("klee_print_range", handlePrintRange, false),
103:  add("klee_set_forking", handleSetForking, false),
104:  add("klee_stack_trace", handleStackTrace, false),
105:  add("klee_warning", handleWarning, false),
106:  add("klee_warning_once", handleWarningOnce, false),
107:  add("klee_alias_function", handleAliasFunction, false),
108:  add("malloc", handleMalloc, true),
109:  add("realloc", handleRealloc, true),
112:  add("xmalloc", handleMalloc, true),
113:  add("xrealloc", handleRealloc, true),
116:  add("_ZdaPv", handleDeleteArray, false),
118:  add("_ZdlPv", handleDelete, false),
121:  add("_Znaj", handleNewArray, true),
123:  add("_Znwj", handleNew, true),
128:  add("_Znam", handleNewArray, true),
130:  add("_Znwm", handleNew, true),
134:  add("__ubsan_handle_add_overflow", handleAddOverflow, false),
135:  add("__ubsan_handle_sub_overflow", handleSubOverflow, false),
136:  add("__ubsan_handle_mul_overflow", handleMulOverflow, false),
137:  add("__ubsan_handle_divrem_overflow", handleDivRemOverflow, false),
jvanegue@llvmlab1:~/hklee$
```

Figure 27. KLEE Special Function Handlers

Symbolic Heap Execution in KLEE

For heap analysis, it appears that KLEE has a strong limitation of where heap chunks for KLEE as well as for the target program are maintained in the same address space. One would need to introduce an allocator proxy[46] if we wanted to track any kind of heap layout fidelity for heap prediction purpose. There are spatial issues to consider there as symbolic heap size may lead to heap state space explosion, so more refined heap management may be required. It may be that other tools relying on selective symbolic execution (S2E)[47] may be more suitable for some of these problems.

Analyzing Distributed Applications.

These are more complex use-cases where KLEE must be modified to track state across distributed component.[48] Several industrially-sized programs use databases and key-value stores and it is interesting to see what symbolic execution model can be defined for those. This approach has been applied to distributed sensor networks and could also be experimented on distributed software in the cloud.

You can either obtain LLVM code by compiling with the clang compiler (3.4 for KLEE) or use a decompiler like McSema[49] and its ReMill library.

There are enough success stories to validate symbolic execution as a practical technology; I encourage you to come up with your own experiments, to figure out what is missing in KLEE to make it work for you. Getting familiar with every corner cases of KLEE can be very time consuming, so an approach of "least modification" is typically what I follow.

Beware of restricting yourself to artificial test suites as, beyond their likeness to real world code, they do not take into account all the environmental dependencies that a real project might have. A typical example is that KLEE does not support inline assembly. Another is the heap intrusiveness previously mentioned. These limitations might turn a golden technique like symbolic execution into a vacuous technology if applied to a bad target.

I leave you to that. Have fun and enjoy!

—Julien

[46] `unzip pocorgtfo18.pdf nextgendebuggers.pdf`
[47] `unzip pocorgtfo18.pdf s2e.pdf`
[48] `unzip pocorgtfo18.pdf kleenet.pdf`
[49] `git clone https://github.com/trailofbits/mcsema`