Paper title: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
Paper authors: Cristian Cadar, Daniel Dunbar, Dawson Engler
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
The paper addresses the issues related to testing - the need of executing code, the difficulty and the bad performance of manual testing. Previous solutions for automated test generation were prone to two main issues - a high number of possible paths in the code & the difficulty in handling code that interacts with the environment including the OS, the network and the user.

Several previous systems for symbolic execution were either static to avoid any interaction with the running environment or restricted the interaction by forcing the use of concrete procedure call arguments. Unlike this, the paper presents an unrestrictive solution that can evaluate the interaction with the environment. In terms of dealing with the path explosion problem, several works have presented heuristics such as **Best First Search**, **Generational Search** & **Hybrid Concolic Testing.** A number of previous systems also included optimizations like **simple syntactic transformations** and the **constraint subsumption optimization** on the query before it is processed by its corresponding constraint solver.

What are the key contributions of the paper?
The paper presents a tool for symbolic execution, called KLEE**,** which can automatically generate high-coverage tests and help in reporting bugs. KLEE operates with the aim of hitting every possible executable line in the code, detecting if any particular input may generate an error for that operation and ensuring no false positives without any required alteration to the source code.

KLEE represents a symbolic process as a tree where the leaves are symbolic variables or constants and the remaining nodes are assembly language operations. Whenever KLEE hits a conditional branch, it checks with the constraint solver to figure out if the condition is an obvious true or false, and then proceeds accordingly. In case of ambiguity, it creates a copy of the state and explores both the possible paths. To improve performance, KLEE maps memory objects to an STP array. Further, in the cases where a pointer points to multiple objects, multiple copies of the state are generated where each copy restricts the pointer to one object. To deal with state explosion, KLEE uses copy-on-write at the object level which reduces the memory required for each state.

To reduce the cost of solving constraints, the queries to the STP constraint solver are optimized by - rewriting expressions, dynamically simplifying constraint sets, evaluating constant values where it is implied, dividing constraint sets into independent subsets, maintaining a cache which maps a set of constraints to counter-examples. For scheduling states, KLEE combines two heuristics - Random Path Selection which uses a binary tree with current states as leaves and other nodes as the instructions where execution forked and Coverage-Optimized Search which computes weights for each state based on its likelihood to reach new code. Both of them are executed in a round robin manner to prevent getting stuck and improve overall efficacy. Further, by using time slicing, KLEE avoids starvation. To deal with the execution environment related bugs, KLEE redirects the environment related system calls to customizable models that can generate the required constraints. To deal with environment failures, KLEE has the option of failing the related system calls in a controlled fashion. It also provides a replay driver to rerun the generated test cases on native binaries.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
KLEE was run on 452 programs having 430K lines of code, and was able to cover 84.5% of COREUTILS and 90.5% of BUSYBOX. It covered the COREUTILS in 89 hours while the previous test suite was a result of 15 years. It was able to discover 56 serious bugs, out of which 3 were fatal bugs in COREUTILS that were detected for the past 15 years. As a result of its ability to test raw unmodified code, KLEE greatly simplified debugging and resulted in bug fixes across COREUTILS in 2 days. It can also be applied to non-application code, like the core of the HISTAR kernel where another serious bug was caught. KLEE performs several optimizations before passing the query to the constraint solver which directly impact its performance. The constraint independence optimization reduced the overall running time by 45%, while the counter-example cache reduced running time and the STP queries 40%. When combined, the average STP query reduction was 5% while the average runtime reduction was more than an order of magnitude. This demonstrates the KLEE is an effective tool which can generate a high coverage and detect bugs much more accurately than previous methods.

Write down one question you may want to ask the authors.
What kind of optimizations can allow the STP to solve constraints for load and store instructions?