Automated testing using symbolic analysis

Tudor Cazangiu, Razvan Crainea
Automatic Control and Computers Faculty
Politehnica University of Bucharest
Emails: {razvan.crainea,tudor.cazangiu}@cti.pub.ro

***Abstract*—In this paper we present parallel-klee, a tool based on KLEE symbolic virtual machine, which is capable of generating high coverage tests for software programs written in C language. Symbolic analysis is a very time consuming process, and our main contribution was to use all the power offered by multiprocessor systems, in order to reduce the time needed for the execution process, and to make feasible the testing of bigger programs. Also because symbolic analysis involves a lot of memory operations, we redesigned the memory manager and we got a significant improvement in the time needed for memory allocation.**

***Index Terms*—symbolic analysis, software testing**

I. Introduction

Nowadays, problems caused by software bugs are very often and they can lead to important damages. Companies spend lots of money on insurance, but sometimes software bugs cause problems which cannot be measured in money. Software testing is a very complicated work, and often the results are not so good. This happens because software testing is time consuming, resource hungry, labor intensive and prone to human error and omission. Important investments are made in quality assurance, but despite this, serious code defects are discovered only after software has been released. Fixing this type of bugs often implies important costs, because serious modifications are needed.

In order to solve this kind of problems, important modifications in how software programs are tested are needed. The only way in which the human related limitations of software testing can be overcome is by developing automated software testing techniques. Some automated testing techniques are used mainly to reduce the time consuming and the laborious process of regression testing. This kind of automated testing can be applied to products that have a long maintenance life. Patches fix some minor issues but can also break some working features in the long run. These techniques are based on writing test cases and running them automatically at given time intervals, to ensure that changes do not affect in a negative way the desired behavior of the application.

These issues have led in the last years at the development of automated testing programs, which generates inputs capable of executing a very large number of the lines of code in a program. One of these tools is KLEE[1]. It is based on symbolic execution, and is capable of generating high coverage tests for programs written in C language. Symbolic analysis, is a static analysis and his ultimate purpose is to construct a mathematical model of the software program. From this model, we can know how the program will behave in all situations. In order to achieve this goal, all the input variables are symbolic and also the environment in which the program runs is symbolic. Symbolic variables are programs variables, which can have all the values permitted by their types. An symbolic environment, means that when a library or a system call is made, this will behave also in a symbolic way, taking into consideration the call parameters, which can be symbolic or concrete, and all possible hardware states.

The key for running programs symbolically, is to transform the instructions that operates with concrete variables, in instructions which can also handle symbolic variables. When a branch instruction based on a symbolic value is encountered, the system follows both branches and adds on each path, a set of constrains called path conditions, which must hold on the execution of that path. When a bug is encounter or a path terminates, the current path conditions are transformed in concrete values, and a test case can be generated. This test case can be later used in order to execute the same path of the program or to hit the same bug.

For now, this approach has proved to achieve good coverage, on small programs. Also, series of bugs have been found, in heavily tested open sorce programs, like Coreutils or Busybox, but real world programs have not been yet tested, mainly due to some drawbacks of this approach. These are the path explosion and the symbolic environment. When a program is tested using symbolic analysis the number of paths tends to grow exponentially. This makes the analysis take a lot of time and use a lot of memory. The second problem, symbolic environment, appears because almost all programs interact heavily with the operating system, the user or the network.

The contributions presented in this paper, tries to attack the path explosion problem. Our version of KLEE is able to use all the computation power, provided by, the multiprocessor architecture in order to analyse bigger programs in shorter time.

II. Architecture

Our work was based on the KLEE project. KLEE is a symbolic execution tool, designed for robust, deep checking of a broad range of applications. Is is based on EXE[2] and runs by interpreting LLVM[5] bitcode. KLEE has some very good features, like compact state representation, constraint solving optimization and uses search heuristics in order to get high code coverage. When it needs to deal with the system environment, it uses a simple approach, external calls

beeing executed in the normal environment. KLEE proved his efficiency, when it was used in order to test Coreutils, arguably the single most heavily tested set of open-source programs in existence. It was able to automaticaly generate tests that covers 84.5% of the total lines of code in Coreutils. KLEE needed 89 hours to run on all Coreutils programs, and it was capable of generating tests, that beat the developers test suit, build incrementally in over 15 years, with 16.8%. Another advantage of KLEE is that all the tests generated, can be run on the row version of the code (compiled with `gcc` for example), simplifying the debugging and error reporting.

From a high level perspective, KLEE is a mix between an operating system for symbolic processes and an interpreter. Like all normal process, symbolic processes have their own stack, registers files, heap and program counter. In addition, they have path conditions. In order to avoid confusion with Linux processes, the symbolic processes are called states. Because KLEE interprets LLVM instructions and maps them to constraints, all the programs which are run on KLEE must be compiled LLVM.

When a program is tested, KLEE needs to cope with a large number of states. The core of KLEE is a loop in which one state is selected and then the next instruction is symbolically executed, in the context of that state. The execution continues until all the states are finished, or a defined time threshold is reached. An important difference between processes and states is how the memory objects are stored. In an normal process, all memory object are data values, while in an symbolic process they are expresions. This expressions are trees in which the leaves are constants or symbolic variables and the interiour nodes are LLVM operations like arithmetic operations, comparisons or memory accesses.

In our quest of making KLEE a solution for testing real world software programs, we first wanted to use all the power offered by multiprocessor systems, so we tried to parallelize the symbolic execution. To do that, we designed two different approaches.

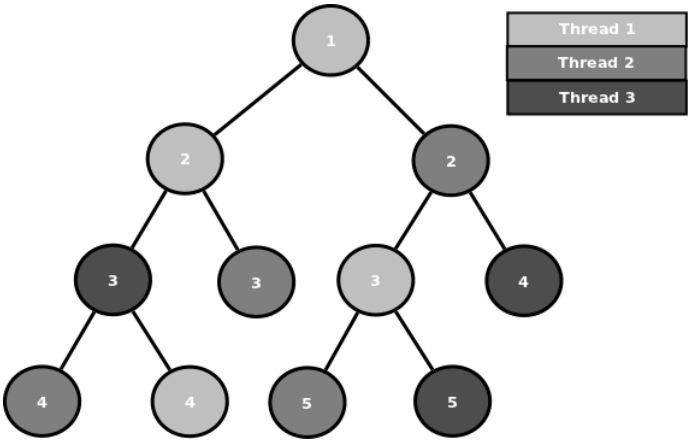

Figure 1. Parallel symbolic execution with one executor and multiple threads

In the first approach we maintain only one executor[1] and we executed multiple symbolic instructions on it. To do so, we shared the executor and we also create a pool in which all the states were put. We designed this solution to work with threads, because we needed to share a lot of the memory space between two different executors. Also, the thread creation overhead, is smaller than the overhead of the by processes. Each thread takes one state from the pool, and executes symbolically the next instruction. When he finishes the execution he puts the state, or states (if there was a branching instruction) back into the pool. The process starts all over again, until there are no states in the pool, or a defined time threshold is reached. An example of how this approach runs is shown in Figure 1. The numbers inside the nodes represent the relative time values when the state is processed.

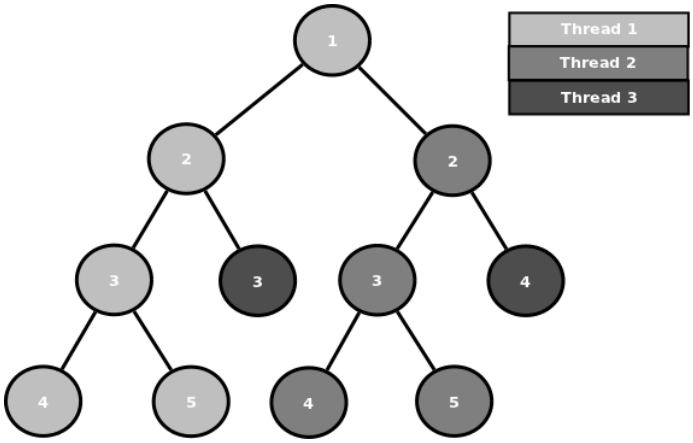

Figure 2. Parallel symbolic execution with one executor per thread

In the second approach we decided to create one executor for each thread. When the symbolic analysis begins, one thread starts executing the instruction of the first state, while the other threads wait until other states are created. When a branching instruction is encounter, and a new state is created, this state is given to a waiting thread. The algorithm goes on, until all the threads have some work to do. From this moment on, each thread keeps all the states generated on his branch. So in this approach states are not passed so frequently between the threads. Only when a thread finishes all his the jobs, he signals this to the other working threads. If they do have more jobs to do, they can pass one of their states to the idle worker. An example of how this approach runs is shown in Figure 2. The numbers inside nodes represent relative time values.

Symbolic execution execution requires a lot of memory. Therefore a lot of memory operations are done, so we decided to change the main program memory manager, in order to reduce the amount of time spent allocating memory. Our profiling analysis reflected a series of memory spaces with specific dimensions that are alocated very often. That is why we decided that instead of allocationg several times small amount of memory, we should create a different memory

[1]operating system for symbolic processes

allocation system that allocates a big chunk of memory and then splits it into small fragments with specific dimensions. When a memory request is issued, instead of calling the `malloc` function each time, we simply return a reference to one of the memory fragments we already have allocated that fits into the desired dimension. The `free` operation is done by simply marking the fragment as unused.

III. Implementation

A. Parallelization

The first solution we tried for KLEE parallelization, in which we use one executor (operating system for symbolic processes) and multiple threads proves to lead to an unexpected problems. In order to use the executor from different threads without problems, we had to use a lot of synchronization mechanisms. After we succeeded synchronizing the access to the LLVM engine, the STP[4] constraint solver and the memory manager, the results were very disappointing. On most of the programs we tested, we could not exceed the normal execution time. The threads were most of the time blocked waiting for synchronization. And this was happening mainly because the symbolic instructions are actually LLVM instructions, and therefore they are executed in the LLVM engine. Also the path constrains, which are executed in the STP constraint solver were consuming a lot of processing time. There were only small regions of code where the threads actually could run in parallel, so the overall speed-up was not so satisfying.

That is why we had to design a different approach, where less synchronization was required. As already presented in section II, we had to implement a different version with a rougher and more complex threads scheduler for states.

So the second approach, where we used one executor for each thread was far more difficult to implement. In order to lower the complexity of the solution, we decided to continue our implementation using the threads paradigm. This approach was proven useful when we started to change states between threads. Even if some parts of the executed programs are still shared between thread, this does not affect us very much since there is a lot of read-only data. Actually sharing this kind of data is useful, because less memory is required. And they are indeed big memory structures like the whole code section representation, the program read-only data, global variables table and so on. But the other information like the states representation and their variables will no longer be shared, and therefore less synchronization mechanisms are required.

KLEE is written in C++, which offered us a big variety of multi threaded programming implementations to choose from, like POSIX threads, TBB, OpenMP and others. Based on our previous experience, we choosed in favor of the POSIX threads[2]. Also the synchronization mechanism offered us the flexibility we needed.

[2]Unix pthreads

B. Memory Allocation Management

According to our profiling analysis, KLEE uses intensively the memory allocation system. And consequently a lot of memory system calls are issued in order to allocate small chunks of data. This obviously drops the performance of KLEE. Therefore we decided that reducing the number of system calls issued, we would improve the global execution time.

Our approach was to create a wrapper over the `malloc` function. The difference between the Standard Template Library memory allocator and our implementation is that instead of executing system calls for each memory request, we allocate once a big chunk at the beginning, and then, for all the sequential requests, we divide this chunk equally in order to fulfill all demands.

In terms of memory, this is definitely not a good solution since we always allocate more memory then needed. But nowadays, the memory consumption is no longer so valuable, in contrast to the execution speed, so we decided to proceed with this approach.

Our implementation consists in developing a C++ class named `MemoryPool` that exports two methods: `klee_alloc` and `klee_free`. The first one is a wrapper over the STL's `malloc` function and it's purpose is to return a chunk of memory and the other one is used to free memory. More about their implementation is detailed in the next paragraphs.

In order to integrate it in KLEE's code, we overloaded the operators `new` and `delete` of the `Expr` class. This class contains the whole structure of the source being analyzed, and therefore most of the memory operations are handled here. Consider for example that for a simple analysis of the classical `HelloWorld.c` test file, there are over 200 memory requests of 8 bits size. The `regexp.c` test (detailed in Section IV) has 192104 memory requests. Also, the whole heap memory used by the analysed program will also be managed by our allocator.

Before proceeding with the implementation, we will have to detail the following terms:

- `MemoryPool` - is the class that represents the memory allocator. It contains the list of buckets.
- `klee_bucket` - a structure that holds the information about the memory allocated. It also holds a pointer to the memory allocated in the first request, and it is split into multiple fragments.
- `klee_frag` - this is a fragment from the actual memory allocated at the request.

The following paragraphs describe the internal implementation of the allocator and the memory organization.

MemoryPool. The `MemoryPool` object consists of a vector of buckets. Each bucket represents a set of objects with similar sizes. For example if a size of 4 bytes is requested, the allocated memory is rounded to the size of the bus (8 in our tests). The same memory size will be allocated if the request would be of 5, 6 or even 8 bytes. But when the requested memory overlaps, a bigger chunk will be returned.

For example for a request of 9 bytes, a memory space of 16 bytes will be returned.

Each bucket has associated a size for it's fragments, which is multiple of the bus size. The first bucket has the size of 8 bytes, the second one 16 bytes, the third 32 and so on. Each bucket has a double size of its predecessor, until the limit of the page size is reached (4KB). For each request of memory larger then 4KB, the allocator will issue a `malloc` call, therefore the allocation is no longer optimized.

Figure 3 illustrates how the memory is organized in a `MemoryPool` object.

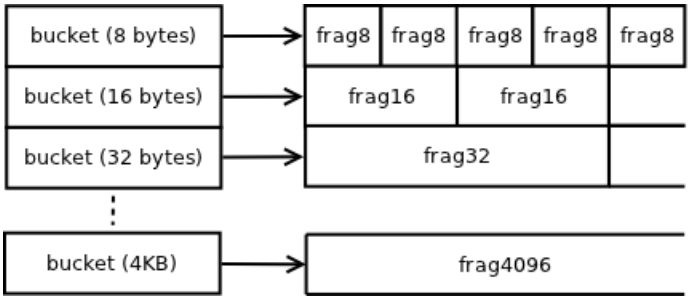

Figure 3. `MemoryPool` object representation

Bucket. `klee_bucket` is a structure that describes the allocated memory zone using `malloc`. It contains information about the size of the fragments it contains, how many fragments it manages, a mask of the free fragments and other meta data. It also contains two pointers: one to the allocated memory zone, and the other to the next `klee_bucket` with the same size. We will see in the next paragraph the reason we have this pointer.

Figure 4 shows the bucket representation.

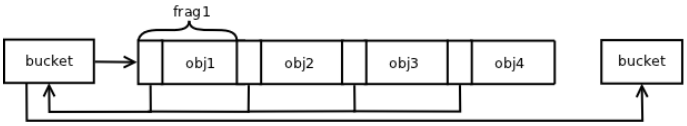

Figure 4. `klee_bucket` structure

Fragment. The fragment is the simplest structure in our allocator. It is also represented in figure 4. It consists only of two fields:

- the object - the actual memory space returned to the request
- a pointer back to the bucket it belongs.

With the terms above being properly explained, we can now describe the behavior of our allocation manager. As stated before, it consists of two actions:

a) `klee_alloc:` it is the method exported by the `MemoryPool` class that returns the needed memory.

The first step done when requesting memory is to compute the rounded size in order to find the needed bucket. As described earlier, the object will be framed in a fragment size greater or equal to the dimension requested. Note that finding the bucket is a very fast process, only some bit operations being required, so the complexity is *O(1)*.

The next step is to check if there was previously allocated a memory for this bucket. If it was not, then the `malloc` function is used to allocate a memory space multiple of the page size[3]. Next, we have to find a free fragment within that bucket. In order to do that, we need to check the bit mask. The first free fragment found is returned to the requested. If there is no free fragment, and the bucket is full, then a new bucket of the same size is allocated and linked to the previous one. And the algorithm goes on, until a free fragment is found. The bucket's mask is updated and then the memory zone is returned.

b) `klee_free:` is called when an object is destroyed.

The implementation of this function is trivial. When given the object, we determine the pointer to the bucket (it is locate right before the object) and update it's mask by setting the fragment as freed. This is also a executed very fast with a *O(1)* complexity.

C. Thread safety

As we already presented in the previous section III-B, memory operations happen very often. Sharing a `MemoryPool` object among all the threads will require a lot of synchronization. Therefore our program performance will decrease considerably. That is why we designed a different approach, where each thread has his own memory manager.

This solution spares us of expensive synchronization between threads and increases the overall performance of KLEE. The implementation was also simple, each thread having to keep it's own `MemoryPool` object in his Thread Local Storage. Each time a thread issues a memory request, the corresponding manager object is determined and the request is satisfied. This method guarantees that two threads can not share the same memory object.

One of the biggest problems with our approach is the memory fragmentation that may appear for intensive testing of big programs. Even if we have not met any problems with the memory during our tests, we would have to consider this in the final deployment of the tool. But once again, our initial settlement was to give memory size a lower priority in contrast with speed.

IV. Experimental Setup

The testing environment we used was an 8 core virtual machine, with QEMU Virtual CPUs clocked at 2GHz and 8 GB of RAM memory. This allowed us to test the program without concerning of the memory limitations. But once again, on a regular computer, this should be taken into account.

As any other system analysis tool, KLEE needs the source code of the program in order to perform the testing and verify it's functional correctness. The program used in our tests was a regular expression matching, presented in the following source code snippet.

[3]One or more pages can be allocated, depending on the configuration of the memory allocator

```
 1. static int matchstar(int c, char *re, char *text) {
 2.   do {
 3.     if (matchhere(re, text))
 4.       return 1;
 5.   } while (*text != '\0' && (*text++ == c || c== '.'));
 6.   return 0;
 7. }

 8. static int matchhere(char *re, char *text) {
 9.   if (re[0] == '\0')
10.     return 0;
11.   if (re[1] == '*')
12.     return matchstar(re[0], re+2, text);
13.   if (re[0] == '$' && re[1]=='\0')
14.     return *text == '\0';
15.   if (*text!='\0' && (re[0]=='.' || re[0]==*text))
16.     return matchhere(re+1, text+1);
17.   return 0;
18. }

19. int match(char *re, char *text) {
20.   if (re[0] == '^')
21.     return matchhere(re+1, text);
22.   do {
23.     if (matchhere(re, text))
24.       return 1;
25.   } while (*text++ != '\0');
26.   return 0;
27. }
```

Even if this short example is not very well structured and does not have a real life appliance, we used it because it emphasises the limitations of a symbolic analysis tool.

- computing intensive - a normal execution of this program takes over 10 seconds.
- highly recursive - all these three functions recurently call each other (lines 3, 12, 16, 21, 23).
- intensive memory access - it iterates over two strings in order to find if they match. Also switching through the stack frames affects a lot the memory operations.
- high number of branches - this program generates a huge amount of states. This is mainly caused by the do-while loops with if nested (see lines 2-5 and 22-25).

Having a well formed image of the testing environment and the testing example, we can now proceed to the next section where we will present our achievements.

V. Scenarios and Results

As we previously stated, our research focuses in two directions: optimizing the memory operations and parallelizing the computing intensive parts.

According to our initial profiling of KLEE running on the regular expression testing program presented in the previous section IV, the `malloc` function call takes over 21% of the total running time. Even the `free` function takes about 11%, counting over 32% of the total amount being wasted on memory operations. This information is presented in Table I.

Our tests consisted in four different scenarios. For each one we reserved different memory chunks for their buckets. The first one allocates only 1 page of 4KB for each bucket, the second one 2 pages, the third 4 and the last 8 pages. The more allocated pages, the better performance we obtain, as we can see in Table I, also graphically represented in Figure 5.

As you can see, we managed to improve the total amount of time spent with memory operations from 32.74% to 22.45%

Operation	Initial	1 page	2 pages	4 pages	8 pages
`malloc`	21.57%	15.83%	15.25%	15.03%	14.91%
`free`	11.17%	8.03%	7.87%	7.76%	7.54%
Total	32.74%	23.86%	23.12%	22.79%	22.45%

Table I
MEMORY OPERATIONS PERCENTAGE

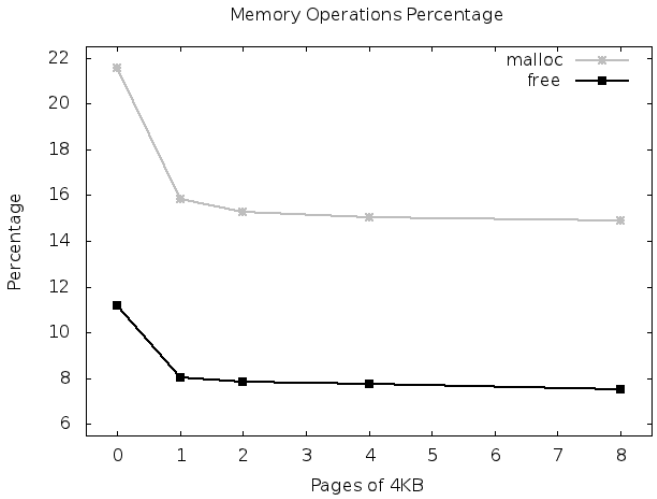

Figure 5. Memory operations percentage

from the total amount of time. Table II presents the total execution time for each case. As you can see, we managed to improve the total amount of time with over a second for this scenario.

	Initial	1 page	2 pages	4 pages	8 pages
Time	13.54s	12.72s	12.59s	12.41s	12.27s

Table II
EXECUTION TIME

The second optimization we performed is related to the parallelization of the high computing area. The approach implemented as described in section III improved the total execution time as described in Table III. Note that this are our final results, including the memory allocator manager.

	1 thread	2 threads	4 threads
Time	13.54s	9.29s	6.21s
Speedup	-	1.45	2.18

Table III
PARALLEL EXECUTION TIME

Table III reflects the final results of our research. Starting from the initial execution time of 13.54 seconds, after applying our optimizations we managed to obtain an execution time of 9.29 seconds for two threads. Executing it with 4 threads, the results are even promising, obtaining an execution time of 6.21 seconds and a speedup of 2.18.

VI. Related Work

The idea of parallel symbolic execution is not new. It was first described in[3]. Cloud9, the result of those ideas, is a distributed solution, which can run symbolic execution on clusters. Cloud9 is based on workers, and uses a load balancer to equal the tasks done by each of those workers.

JPF, Java Pathfinder[6], also has an extension that parallelizes symbolic execution by using parallel random searches on a static partition of the execution tree. JPF is also based on workers and pre-computes a set of disjoint constraints that, when used as preconditions on a worker's exploration, steers each worker to explore a subset of paths disjoint from all other workers. In this approach, using constraints as preconditions, imposes at every branch in the program, a solving overhead relative to exploration without preconditions. The complexity of these preconditions increases with the number of workers, as the preconditions need to be more elective.

VII. Conclusion and Further Work

The results presented in section V illustrate our achievements. We have successfully accomplished or goal by improving KLEE, parallelizing the intensive computational zone. We have also proposed a new memory allocator manager that offers a better memory management, but with the cost of internal fragmentation.

There is still a lot of optimizations that can be done for KLEE and generally in the symbolic analysis field. Researches like optimizing the path merging or the constraint solver, I/O buffering features, would improve this domain, and therefore will create more powerful software testing tools that will be able to verify any application.

Acknowledgment

The authors would like to thank Emil Slusanschi and Mircea Bardac for the suggestions, ideas and feedback they gave throughout the course of the project.

References

[1] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. 2008.
[2] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. Exe: Automatically generating inputs of death. 2006.
[3] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. 2009.
[4] V. Ganesh and D. Dill. A decision procedure for bit-vectors and array. 2007.
[5] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation, 2004.
[6] M. Staats and C. Pasareanu. Parallel symbolic execution for structural test generation. 2010.