CS554: Software Engineer
Ovidiu Mura
March 12, 2020

Symbolic Execution with KLEE
- term paper -

Introduction. Formal methods are mathematical approaches to software development advocated by the compute scientist researchers. A typical approach is to define a formal model, then the model is formally analyzed for errors and inconsistencies. To prove that the program is inconsistent with the model, it can be applied a series of correctness-preserving transformations to the model to generate a program. Using formal methods can reduce faults in the system. Examples of industrial applications where the formal methods can successfully be applied are train control system, cash card system, and flight control system. Formal methods are used in static verification tools (i.e. Microsoft driver verification system). In software development, the mathematically formal approach idea is that formal specification and the program can be developed independently. Next, a mathematical proof can be developed to show that the program and its specification are consistent.

Advantages of developing a formal specification and using it in a formal development process are the followings: first, the process will develop a deep and detailed understanding of the system requirements; second, the specification is expressed in a language with formally defined semantics, easy to discover inconsistencies and incompleteness; third, symbol execution formal method can be used to transform the formal specification into a program through a sequence of correctness-preserving transformations, guaranteeing to meet its specification; forth, the testing costs are reduced because the verification of the program against its specifications are already completed. These advantages show that using formal methods errors are reduced in the delivery software, the costs and the time needed for software development are reduced.

There are disadvantages using formal methods in developing software: first, product owners and domain experts cannot understand the formal specification, and software engineers who understand the formal specification may not understand the application domain; second, difficult to estimate the possible savings resulted from using formal methods, but easy to quantify the costs creating a formal specification; third, not many software engineers are trained to use formal specification languages; fourth, it is difficult to scale the formal methods to large systems, they are used for specifying critical kernel software rather than the complete system; fifth, tool support for formal methods are limited; sixth, formal methods are not compatible with agile development where programs are developed incrementally.

Current formal methods are criticized by computer scientists about their fundamentally incorrect premises. I believe that the formal methods must be upgraded to be more efficient and simple to be able to be implemented at a lower cost. These changes will require a different type of mathematics as a basis because to reduce the costs significantly needs the implementation, which is build on mathematics, to be changed accordingly.

This paper presents symbolic execution approach to show the program and its specification are consistent. There are two classes of error in software representations that can be discovered with formal methods: first error class is specification, design errors, and omissions, and second error class, inconsistencies between a program and its specification.

A mathematical system model is created which will act as a system specification. The creation process of the mathematical system mode is done by translating the system specification which are natural language, diagrams and tables, into a mathematical language that has formally defined

semantics. Because the formal specification is an unambiguous description of what the system should do, we use mathematical language that has formally defined semantics. Formal method, symbolic execution, is used to build an operational model which analyzes a program and its specification are consistent. Symbolic execution in computer science is used to analyze a program to determine the execution paths and what inputs cause each path to execute. It is a software technique that is used to help the generation of test data and proving the program quality.

Symbolic execution tools automatically generates tests and achieve high coverage on a diverse set of complex and environmentally-intensive programs. It can be used as bug finding tool and check the correctness errors and look for inconsistencies between specification and implementation of the application. These tools automatically generates test inputs to find functional correctness bugs running at a better performance. Symbolic execution tools replace the manual running code and the randomly-constructed input, with symbolic values and replaces the corresponding operations with one that manipulate symbolic values. If the program execution branches based on a symbolic value, the system follow both branches on each path maintaining a set of constraints, path condition, which must hold on that execution path. If a path terminates or find a bug, a test case can be generated by solving the current path condition for concrete values. We assume the deterministic code will take the concrete input and feeds into unmodified version of the checked code which will follow the same path and find the same bug. I am not sure if the approach has any hope of consistently achieving high coverage in real applications. There are two issues for not being able to achieve high coverage, first, the exponential number of paths through code and second, the challenges in handling code that interacts with its surrounding environment, such as the operating system, the network, or the user.

KLEE is a symbolic execution tool which is designed for robust, deep checking of a broader range of applications. KLEE implement a variety of constraint solving optimizations, which represents program states compactly, uses search heuristics to get high code coverage, and it uses a simple and straightforward approach to dealing with the external environment. KLEE automatically generate tests that get high coverage on a diverse set of real, complicated, and environmentally-intensive programs. KLEE ran on more than 452 programs to show the tool's performance and potential. The experiments highlight the followings: first, KLEE gets high coverage on a broad set of complex programs; next, KLEE gets significantly more code coverage than manual efforts; next, with one exception, KLEE achieved the high coverage results on unaltered application; next, KLEE finds errors in heavily-tested code; next, KLEE test cases can run on the raw version of the code, greatly simplifies debugging and error reporting; next, KLEE is not limited to low level programming errors; next, KLEE can also be applied to non-application code. These finding show that KLEE's automatically generated test cases have a high coverage results, have good performance, and find difficult errors in programs. Also, the experiments show that KLEE is not limited to low level programming errors, it can be used to test applications. KLEE has two goals during the execution: first, hit every line of executable in the program and second, detect at each dangerous operation if any input value exists that could cause an error, running programs symbolically – they generate constraints that exactly describe the set of values possible on a given path. When KLEE detects an error or when a path reaches an exit call, KLEE solves the current path's constraints to produce a test case that will follow the same path when rerun on an unmodified version of the checked program; the paths followed by the unmodified program will always follow the same path KLEE took.

To start checking real programs, KLEE do not require source modifications or manual work. The first step to start KLEE is, the users compile their code to byte-code using the publicly-available LLVM for GNU C, for example to compile program p.c run the command: **llvm-gcc –emit-llvm -c p.c -o p.bc**

Then users run KLEE on the generated byte-code with the options, start the number, size, type of

symbolic inputs to test the code on. For example, to run KLEE on the *p* program, the following command: **klee –ma-time 2 –sym-args 1 10 10 –sym-files 2 2000 –max-fail 1 p.bc** The first option, --max-time, tells KLEE to check p.bc for at most two minutes, the rest describe the symbolic inputs: the option –sym-args 1 10 10 says to use zero to three command line arguments, the first 1 character long, the others 10 characters long. Next, the option –sym-files 2 2000 says to use standard input and one file, each holding 2000 bytes of symbolic data and lastly, the option –max-fail 1 says to fail at most one system call along each program path.

I will present next an symbol execution scenario with KLEE. For example KLEE runs program **p,** then it finds a buffer overflow error, **if (*arg++ != '-') {**, in the code then it produces a concrete test case **(tr [“” “”)** that will execute it. KLEE runs the rest of the program with the options presented above, as follows.

KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination, then create a constraint for the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively, then main function is called with these constraints. KLEE runs on program **p** and it finds buffer overflow error in ling 18, see the appendix – Source Code 1, for sample code from MINIX. Next, KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. Next, KLEE executes the branch argc > 1 at line 33, it uses its constraint solver STP[23] to see which directions can execute given the current path condition. Since KLEE forks execution and follows both paths, KLEE will add the constraint **argc > 1** on the false path and **argc <= 1** on the true path. Next, KLEE chooses which path to execute when there are many paths. Next, KLEE checks for errors using all possible values allowed by the constraints. KLEE detects the error in line 18 described above, then it continues following the current path, adding the constraint that the error does not occur to be able to find new errors. Next, KLEE generates concrete values for argc and argv such as **p [“” “”** then when the test case is executed, the execution will find this bug.

KLEE is designed to function as a hybrid between an operating system for symbolic processes and an interpreter. The symbolic process represents a KLEE state which has a register file, stack, heap, program counter, and path condition. The programs are compiled to the LLVM assembly language, then KLEE interprets this instruction set and maps instructions to constraints without approximation – bit level accuracy. KLEE has as its core a loop which selects a state to run then symbolically executes a single instruction in the context of that state. KLEE terminates the loop when the all the states were executed or a user-defined timeout is reached. The storage locations for a state refers to expressions instead of raw data values, and the leaves of an expression are symbolic variables or constants (concrete), and the interior nodes come from LLVM assembly language operations. Symbolically execution of an LLVM add instruction is **%dst = add i32 %src0, %src1**. KLESS retrieves the values from %src0 and %src1 registers and writes a new expression **Add(%src0, %src1)** to the **%dst** register. KLEE alter the instruction pointer of the state based on whether the condition is true or false for a conditional branches that take a boolean expression. KLEE queries the constraint solver to determine if the branch condition is either provably true or provably false on the current path, then the instruction pointer is updated to the appropriate location or KLEE clones the state to explore both paths updating the instruction pointer and the path condition on each path. When a dangerous operation is found in the source code KLEE generates checks and if an error is detected, KLEE generates a test case to trigger the error and then terminates the state.

The load and store instructions generate checks for the address to be in bounds of a valid memory object. The memory representation is a flat byte array. Loads and stores would simply map to array read and write expressions. Every memory object in KLEE is mapped into an distinct array. The arrays which are not referenced by a given expression are ignored. If many operations require object-

specific info, these operations become difficult to perform, KLEE clones the current state to the number of objects the pointer is referring. Each state will constrain the pointer to be withing bounds of its respective objects and then perform the read or write operation. These operations can be very expensive and KLEE need to be optimized for this scenario.

KLEE optimizes the queries before they reach the KLEE's solver STP. The benefits for using the optimizer are simplifying the queries make solving faster, reduce the memory usage, and increase the query cache's hit rate. The number of states can grow very fast since small programs can sometimes generate hundreds of thousands of concurrent states during the interpretation step. KLEE tracks all memory object and implement copy-on-write policy at the object level to reduce per-state memory requirements. KLEE also implement the heap as an immutable map with parts of the heap structure to be shared among multiple states, and this heap can be cloned in constant time. KLEE uses optimization techniques such as expression rewriting, constraint set simplification, implied value concretization, constraint independence, counter example-cache.

KLEE uses two search heuristics to decide what state to run at each instruction. ***Random path selection*** search heuristic maintains a binary tree recording the paths of the program followed for all active states; the leaves are the current states and the nodes on the other depths are where the execution forked. The states are selected by traversing the tree from root and randomly selects the path to follow at branch points. When a branch point is reached, the set of states in each sub-tree has equal probability of being selected, regardless of the size of their sub-trees. These strategy has two properties: first, the states high in the branch tree are more likely to run because they have less constraints on their symbolic inputs. Second, it avoids starvation when some part of the program is rapidly creating new states, this happen when a loop contains a symbolic condition. ***Coverage-Optimized Search*** search heuristic selects states with high probability to cover new code in the immediate future. It uses heuristics to compute a weight for each state and then randomly selects a state according to these weights considering the minimum distance to an uncovered instruction, the call stack of the state, and if the state covered new code recently. These strategies are used in round robin fashion, this way, KLEE ensures that a state with expensive instructions will not dominate execution time, running each state for a “time slice” defined by maximum number of instructions and a maximum amount of time.

KLEE reads values from environment such as command-line arguments, environment variables, file data, and metadata, network packets then redirects the calls to models that understand the semantics of the desired action well enough to generate the required constraints. For example, for file system operation KLEE checks if the action is for an actual concrete file on disk or a symbolic file, then for concrete files KLEE invoke the corresponding system call in the running operating system, and for symbolic files KLEE emulate the operation's effect on a simple symbolic file system, private to each state. KLEE has an optional mode for dealing with the real environment failure from unexpected ways such as write() system call fails because of a full disk. Such failures can often lead to unexpected and hard to debug.

KLEE generated test cases are rerun on the unmodified native binaries by supplying them to a replay driver we provide. The individual test cases describe an instance of the symbolic environment. The driver uses this description to create actual operating system objects such as files, pipes, ttys, directories, and links, using the values from the test case. It then executes the unmodified program using the concrete command-line arguments from the test case. The biggest challenge was making system calls fail outside of KLEE; a utility was developed using ***ptrace*** debugging interface to skip the system calls that were supposed to fail and instead return an error.

KLEE tool was evaluated running against COREUTILS 5.2 and Busybox 5.3 and analyzing **in-depth coverage** and the **errors** found during quick bug-finding runs. KLEE also run to find deep correctness errors. KLEE measure the coverage by running KLEE-generated test cases on a stand-alone

version of each utility and using *gcov* to measure coverage for the code in the tool without the library code. Counting the library code is difficult because it double-count many lines, since the library function can be called multiple times, and it unfairly under-counts coverage because the library function balled by an application is dead code since the library code is general but call sites are not.

The evaluation of KLEE will include library code only when measuring the raw size of the application. KLEE generates paths that only reach new statements or branch in the main utility code. KLEE coverage results for all 89 GNU COREUTILS utilities are the followings – the smallest five have between 2K and 3K executable lines of code, 52 have between 3K and 4K, and ten have over 6K. The tools were tested using the same command, **./run <tool-name> --max-time 60 –sys-args 10 2 2 –sys-files 2 8 [--max-fail 1]**

After the tools were run for about 60 minutes the line coverage results on average cover 90.9% of the lines in each tool, with an overall coverage across all tools of 84.5%. There are 100% line coverage on 16 tools, over 90% on 56 tools, and over 80% on 77 tools. Minimum tool coverage was 62.6%. These results show how powerful the approach is run across the entire tool suite. KLEE generates high coverage. KLEE was evaluated with and without failing system call invocations for all tools and hitting system call failure paths was shown useful because the overall results improved from 79.9% to 84.5%.

Another evaluation was done by comparing KLEE against developer test suites. COREUTILS comes with manually written test suites and KLEE perform better than the developer tests, overall total line coverage is 84.5% for KLEE and 68.4% for developer test suites; KLEE gets 100% coverage on 16 tools and over 90% coverage on 56 while the developer tests get 100% on a single utility and reach over 90% on only 7 utilities. The developer tests get below 60% coverage on 24 tools while KLEE always achieves over 60%. KLEE exceeds the coverage of developer test suite by 16.8%, running 89 hours, compared to the developer test suite built over a period of fifteen years. KLEE only checks for low-level errors and violations of user-level asserts and in contrast developer test suite validate that the application output matches the expected output.

There was ten unique bugs in COREUTILS found by KLEE. The command lines which will trigger these bugs are the followings separated by a comma: *paste -d\\ abcdefghijklmnopqrstuvwxyz, pr -e t2.txt, tac -r t3.txt t3.txt, mkdir -Z a b, mkfifo -Z a b, mknod -Z a b p, md5sum -c t1.txt, ptx -F\\ abcdefghijklmnopqrstuvwxyz, ptx x t4.txt, seq -f %0 1* . All the presented bugs are fixed in the official COREUTILS test suite now. For example, one of the bugs found is "*pr -e t2.txt*" and examining the code from the appendix – Source Code 2, we can see that the path that hits the bug, both *chars_per_input_tab* and *chars_per_c* equal tab width. In line 2665, it computes *width = (tab – input_position mod tab)* using the macro in line 602. The error was in the assumption that *0 <= x mod y < y*, which only holds for positive integers, when *input_position* is positive, *width* will be less than tab since *0 <= input_position mod tab < tab,* and *input_position* become negative when there are backspaces, such as *(-tab < input_position mod tab < tab).* When the code allocates a buffer *clump_buff* of size tab in line 1322 and then writes width characters into this buffer lines 2669-2670 via the pointer *s* the bug is triggered. Memory overflow appear when *width* has a value larger than *2tab -1*. These kind of bugs are very difficult to find manually and using symbolic execution shows KLEE is very effective discovering bugs which are hard to find manually.

Random tests were developed and compared with the manual tests and KLEE symbolic execution tests to evaluate the KLEE tool. To develop the random tests a tool was built which takes the same command line as KLEE and generates random values for the specified type, number and size range inputs, then runs the checked program on these values using the same replay infrastructure as KLEE. The tests were run against 15 benchmarks for 65 minutes using the same command-line as KLEE. The results on running the manual, random, and KLEE tests show that KLEE perform the best,

manual tests get significantly more coverage than random tests.

Tests were run against Busybox, a standard UNIX utilities for embedded systems. The same command lines used as for COREUTILS without the fail system calls. The Busybox “coreutils” sub-directory contains 75 utilities in 72 files which are 14K lines of code, with another 16K of library code. KLEE does better than COREUTILS, over 90.5% total line coverage, on average covering 93.5% per tool. It was 100% coverage on 31 and over 90% on 55 utilities. BUSYBOX compared with COREUTILS has less comprehensive manual test suite, as a result, KLEE beats the BUSYBOX developers tests 90.5% total line coverage versus only 44.8% for the developers' suite. KLEE was used to check for bugs in aprox 360 applications in BUSYBOX and MINIX tools and there were found quickly 21 bugs in BUSYBOX and 21 in MINIX. All these bugs were memory errors and were they were fixed.

KLEE is not only limited to finding generic errors that do not require knowledge of a program expected behavior. KLEE is able to check and verify functional correctness on a finite set of explored paths with no approximations, it uses perfect accuracy constraints which down to the bit level. When KLEE reaches an *assert* and its constraint solver states the false branch cannot be executed with the current path constraints, then this proves that no value exists on the current path that could violate the assertion.

KLEE is able to do proofs for any condition the programmer expresses as C code – from a simple non-null pointer check to verify the correctness of a program's output. KLEE uses the proof property to verify functional equivalence on a per-path basis. Assume there are two procedures f and f' that take a single argument and they implement the same interface, then KLEE calls both procedures with the same symbolic argument and asserting they return the same value, i.e. *assert(f(x) == f'(x))*. KLEE checks if any value violates the assertion on the path which reaches this assertion, if there is no such value, then it has proven functional equivalence on that path. If one function is correct along the path, then equivalence proves the other one is correct too. Differently, if the function compute different values along the path and there are values which violate the assert, then KLEE will produce a test case demonstrating this difference. This testing using proof path equivalence it's very powerful property which moves beyond the traditional testing.

KLEE is able to transform a path through a C program code into a problem of theorem prover domain, using proving path equivalence with few lines C code and the theorem proving method uses tedious and long manual exercise. The proving path equivalence results only works on the finite set of paths that KLEE explores. Similar to traditional testing, proving path equivalence method cannot make statements about paths it misses; if KLEE is able to exhaust all paths, it shows the total equivalence of the functions.

An code example which shows how the proving path equivalence works, is presented in appendix – Source Code 3. This example checks a trivial module implementation against one that optimizes for modulo by powers of two. It makes the input x and y symbolic, then uses the assert in line 14 to check for differences. Two code paths are created which reach this assert depending on if the test for power of two in line 2 succeeds or fails. The true path uses the solver to check that the constraint *(y&-y) == y* implies *(x&(y-1)) == x%y* holds for all values, which succeeds. The false path checks the constraint *(y&-y) != y* implies that *x%y == x%y* also succeeds, then KLEE checking process terminates, which means KLEE has proved equivalence for all non-zero values. KLEE is very powerful and efficient because it proves the equivalence using only a few lines of code. The proof path equivalence method is very useful because software such as libraries, servers, compilers have multiple implementations can be verified efficiently. There are other scenarios where this method can be applied such as: f is a reference implementation and f' is a real world optimized version, f' is patched version of f that is used to remove bugs or refactor code without changing functionality, and f has an inverse – the

equivalence check can be changed to verify $f^{-1}(f(x)) \equiv x$ such that *assert(uncompress(compress(x))==x)*.

KLEE can be also used to check kernel code. HiStar is a new operation system designed to minimize the amount of trusted code in a system, it provides strict information flow control which allows users to specify precise data security policies without unduly limiting the structure of application. To set the environment, a driver was created based on user-mode HiStar kernel, then the driver create the core kernel data structures and initializes one single process with access to a single page of user memory. The driver then calls a test function which makes the user memory symbolic and executes a predefined number of system calls using entirely symbolic arguments. The system call number is encoded and passed as an argument to this test function, then the driver tests all system calls in the kernel. The coverage obtained for the core kernel for runs with and without a disk show that KLEE's tests achieve significantly more coverage than random testing both for runs with disk +17.0% and without a disk +28.4%. KLEE finds a bug in HiStar which in computing overflow allows a malicious process to gain access to memory regions outside its control, this is a critical security bug in the 32-bit version of the HiStar. This function is called safe_addrptr and its code is shown in the appendix – Source Code 4. It sets *of to true if the addition overflows, and because the inputs of the function are 64 bits long the test used is invalid for overflow for large values of b; the correct test should be (r < a) || (r < b).

This function validates user memory addresses prior to copying data to or from user space. This function uses the overflow to prevent access when a computation could overflow, in the kernel routine which takes a user address and a size to compute if the user is allowed to access its memory for the given range.

The KLEE tools is very powerful and generates high coverage tests but there is still room for improvements to get to +90% code coverage as presented above. This results show KLEE automatically generated tests with average covered over 90% of the lines in 160 complex systems, which exceeded their corresponding hand-written test suites. KLEE checked 452 applications with over 430K lines of code and found 56 serious bugs from which 10 in COREUTILS. COREUTILS is one of the most tested software on the open-source applications which shows that KLEE approach to testing works very well. Because KLEE don't approximate constraints it allows to prove properties of paths and this was used to prove path equivalence for real applications and to find functional correctness error in the applications. KLEE is also used to check non-application code such as HiStar kernel where KLEE found a critical security bug in the 32-bit version of HiStar. This bug allows a malicious process to gain access to memory regions outside its control.

KLEE can be very helpful when the main test structures need to be generated but the effectiveness highly depends on the application structure. KLEE also can be extended to work with other languages other than C because it is build on top of LLVM framework. KLEE has the capability to interact with code outside of program code using the interpreter to simulate all the possible variables such and these interaction can be added to the flow. KLEE uses models that imitate the behavior of various calls and constraints.

KLEE should not be used on path explosions because symbolically executing all feasible program paths does not scale to large programs and the number of feasible paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations. KLEE also should not be used in program dependent efficiency because symbolic execution is used to reason about program path-by-path which is an advantage over reasoning about a program input-by-input as other testing paradigms use such as dynamic program analysis; if few inputs take the same path through the program, there is little savings over testing each of the inputs separately. KLEE also should not be used in environment interactions unless implement a driver

because programs interact with their environment by performing system calls, receiving signals; consistency problems may arise when execution reaches components that are not under control of the symbolic execution tool.

References

[1] Kneuper R. (December 1990). Symbolic execution: a schematic approach. March 10, 2020, from https://core.ac.uk/download/pdf/82302097.pdf

[2] Sommerville, I. (2015). SOFTWARE ENGINEERING, 10th Ed., Pearson.

[3] Baldoni, R. (2018). A Survey of Symbolic Execution Techniques. March 10, 2020, from https://arxiv.org/pdf/1610.00502.pdf

[4] Cadar, C. (2008). KLEE: Unassisted and Automatic Generation of high-Coverage Tests for Complex Systems Programs. March 10, 2020, from https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar.pdf

[5] Zmyslowski, M. (2018). Feeding the Fuzzers with KLEE. March, 10, 2020, from https://srg.doc.ic.ac.uk/klee18/talks/Zmyslowski-Feeding-the-Fuzzers-with-KLEE.pdf

APENDIX

Source Code 1: Program *p*, it is used as an example to describe the symbol execution tool, KLEE. The following code is an short sample from MINIX to show the tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are shown on the right side of the code snippets.

```
1 : void expand(char *arg, unsigned char *buffer) { //8
2 : int i, ac; //9
3 : while (*arg) { //10*
4 :         if (*arg == '\\') { //11*
5 :                 arg++;
6 :                 i = ac = 0;
7 :         if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :                 ac = (ac << 3) + *arg++ - '0';
10:                 i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:         } else if (*arg != '\0')
14:                 *buffer++ = *arg++;
15:         } else if (*arg == '[') { //12*
16:         arg++; //13
17:         i = *arg++; //14
18:         if (*arg++ != '-') { //15!
19:                 *buffer++ = '[';
20:                 arg --= 2;
21:                 continue;
22:         }
23:         ac = *arg++;
24:         while (i <= ac) *buffer++ = i++;
25:         arg++; /* Skip ']' */
26:         } else
27:                 *buffer++ = *arg++;
28:         }
29: }
30: . . .
31: int main(int argc, char* argv[ ]) { //1
32:         int index = 1; //2
33:         if (argc > 1 && argv[index][0] == '-') { //3*
34: . . . //4
35:         } //5
36: . . . //6
37:         expand(argv[index++], index); //7
38: . . .
39: }
```

Source Code 2: The program pr where memory overflow of clump_buff via pointer s is possible if chars_per_input_tab == chars_per_c and input_position < 0.

```
602: #define TAB WIDTH(c , h ) ((c ) - ((h ) % (c )))
. . .
1322: clump buff = xmalloc(MAX(8,chars per input tab));
. . . // (set s to clump buff)
2665: width = TAB WIDTH(chars per c, input position);
2666:
2667: if (untabify input)
2668: {
2669: for (i = width; i; --i)
2670: *s++ = ' ';
2671: chars = width;
2672: }
```

Source Code 3: Example for equivalence checking, KLEE proves total equivalence when y != 0.

```
1 : unsigned mod opt(unsigned x, unsigned y) {
2 : if((y & −y) == y) // power of two?
3 : return x & (y−1);
4 : else
5 : return x % y;
6 : }
7 : unsigned mod(unsigned x, unsigned y) {
8 : return x % y;
9 : }
10: int main() {
11: unsigned x,y;
12: make symbolic(&x, sizeof(x));
13: make symbolic(&y, sizeof(y));
14: assert(mod(x,y) == mod opt(x,y));
15: return 0;
16: }
```

Source Code 4: HiStar function containing an important security vulnerability. This function should set *of to true if the addition overflows but can fail to do so in 32-bit version for very large values of b.

```
1 : uintptr t safe addptr(int *of, uint64 t a, uint64 t b) {
2 : uintptr t r = a + b;
3 : if (r < a)
4 : *of = 1;
5 : return r;
6 : }
```