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

Cristian Cadar, Daniel Dunbar, Dawson Engler [*]
Stanford University

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We applied KLEE to all 90 programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on almost all Unix systems and, as such, represent some of the most heavily used and tested open-source programs in existence. For 84% of these utilities, KLEE's automatically generated tests covered 80–100% of executable statements and, in aggregate, significantly beat the coverage of the developers' own hand-written test suites. KLEE also found nine serious bugs (including three that had been missed for over 15 years!) and produced concrete inputs that triggered the errors when run on the uninstrumented code. When applied to MINIX's versions of a small selection of the same applications, KLEE achieved similar coverage (along with two bugs). In addition, we also used KLEE to automatically find numerous incorrect differences between several MINIX and COREUTILS tools. Finally, we checked the kernel of the HISTAR operating system, generating tests that achieved 76.4% (without paging enabled) and 67.1% coverage (with paging) and found one important security bug.

1 Introduction

The importance of testing and the poor performance of random and manual approaches has led to much recent work in using *symbolic execution* to automatically generate high-coverage test inputs [9, 19, 10, 28, 18, 20, 6, 15, 14, 8, 16]. At a high-level, these tools use variations on the following idea: Instead of running code on manually or randomly constructed input, they run it on symbolic input initially allowed to be "anything." They substitute program inputs with symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches at once, maintaining on each path a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition to find concrete values. Assuming deterministic code, feeding this concrete input to an raw version of the checked code will cause it to follow the same path and hit the same bug.

Results from these tools and others are promising. However, while researchers have shown such tools can get high coverage and find bugs on a small number of programs, it has been an open question, especially to outsiders, whether the approach has any hope of consistently achieving these goals on real applications. Two common concerns are the exponential number of paths through code and the difficulty of handling the environment ("the environment problem"). Neither concern has been much helped by the fact that most past work, including ours, has usually reported results on a limited set of hand-picked benchmarks and typically has not included any coverage numbers.

This paper makes two contributions: First, we present a new symbolic execution tool, KLEE, which we designed from scratch to be robust and to deeply check a broad range of applications. We leveraged several years of lessons from our previous tool, EXE [10]. KLEE uses novel constraint solving optimizations that improve performance by over an order of magnitude and let it handle many programs that are completely intractable otherwise. Its space-efficient representation of a checked path means it can have tens to hundreds of thousands of such paths active simultaneously. Its search heuristics effectively select from these large sets of paths to get high code coverage. Its simple, straightforward approach to handling the environment let it check a broad range of

[*] Author names are in alphabetical order. Daniel Dunbar is the main author of the KLEE system.

system-intensive programs.

Second, we show that KLEE's automatically generated tests get high coverage on a diverse set of real, complicated, and environmentally-intensive programs. Our main evaluation applies KLEE to all 90 programs in the latest stable version of GNU COREUTILS (version 6.10). In total, COREUTILS consists of about 78,000 lines of library code and 65,000 lines in the actual utilities. The utilities themselves interact aggressively with their environment to provide a variety of functions, including managing the file system (e.g., `ls`, `dd`, `chmod`), displaying and configuring system properties (e.g., `logname`, `printenv`, `hostname`), controlling command invocation (e.g., `nohup`, `nice`, `env`), processing text files (e.g., `sort`, `od`, `patch`), and so on. They form the core user-level environment installed on almost all Unix systems. They are used daily by millions of people, bug fixes are handled promptly, and new releases are pushed regularly. The breadth of functions they perform means that our system cannot be a "one trick pony" special-cased to one application class. Moreover, the heavy use of the environment stress tests our system where symbolic execution has historically been weakest.

Further, the robustness of COREUTILS programs make them very tough tests in terms of bug-finding. They arguably are the single most well-tested suite of open-source applications (e.g., is there a program the reader has used more often than "`ls`"?). The "fuzz" study found that GNU utilities had 2.5 to 7 times less failures than the utilities on seven commercial Unix systems [27]. This difference was in 1995, and there have been over 12 more years of testing since then. The last COREUTILS vulnerability reported on the SecurityFocus or the US National Vulnerability Database was three years ago [2, 4].

Our experiments (§ 5) show the following:

1 KLEE works well on a broad set of complex programs. When applied to the entire 90-program COREUTILS suite it automatically generated tests covering 80–100% of the statements on 84% of the utilities, in aggregate covering 81.9% of the total code. Sixteen tools have 100% coverage and 38 over 90%. These results held true when applied to 14 comparable versions of MINIX utilities, where it achieved over 90% statement coverage (and found two bugs). To the best of our knowledge, COREUTILS contains an order of magnitude more programs than prior symbolic test generation work has attempted to test.

2 KLEE can get significantly more code coverage than a concentrated, sustained manual effort. The 90-hour run used to generate its COREUTILS coverage beat the developer's own test suite built incrementally over fifteen years by over 14%!

3 With one exception, KLEE got these high-coverage results by checking the applications "out of the box" without requiring any special-case source modifications. (We did a one-line modification to `sort` to shrink a large buffer that caused problems for the constraint solver.)

4 KLEE finds important errors in heavily tested code. KLEE found nine fatal errors in the latest version COREUTILS (including three that had escaped detection for 15 years!), which account for more crashing bugs than found in 2006, 2007 and 2008 combined. In addition, KLEE produces concrete inputs that can be run independently to demonstrate the error, greatly simplifying debugging. All outstanding bugs were confirmed and fixed each within two days of our report and versions of the tests KLEE generated were included in the standard regression suite.

5 KLEE also handles operating system code well. We applied it to the core part of the HISTAR kernel, achieving an average statement coverage of 76.4% (with disk) and 67.1% (without disk) and finding a serious security bug.

6 KLEE is not limited to finding low-level programming errors such as memory overflows, but has also been applied to checking functional correctness by finding inconsistencies between several purportedly identical MINIX and COREUTILS tools.

We give an overview of our approach in the next section. Section 3 describes KLEE, focusing on the most important optimizations we do. We then discuss how we model the environment (§ 4) and then our experiments (§ 5). Finally we describe related work (§ 6) and then conclude (§ 7).

2 Overview

This section explains how KLEE works by walking through the testing of MINIX's `tr` tool. Although this version of `tr` is very small – only 169 lines, 83 of which are executable – it illustrates several issues common to the programs we check:

1 *Complexity.* The code's intent is to translate and delete characters from its input. It hides this intent well beneath non-obvious input parsing code, tricky boundary conditions, and hard-to-follow control flow. For example, Figure 1 shows one of the complicated string parsing procedures contained in the utility.

2 *Environmental Dependencies.* Most of the code is controlled by environmental input. Command line arguments determine what procedures execute, input values determine which way if-statements trigger, and the program depends on the ability to read from the file system. These inputs often come from un-

```
void expand(char *arg, unsigned char *buffer) {
  int i, ac;
  while (*arg) {
    if (*arg == '\\') {
      arg++;
      i = ac = 0;
      if (*arg >= '0' && *arg <= '7') {
        do {
          ac = (ac << 3) + *arg++ − '0';
          i++;
        } while (i < 4 && *arg >= '0' && *arg <= '7');
        *buffer++ = ac;
      } else if (*arg != '\0')
        *buffer++ = *arg++;
    } else if (*arg == '[') {
      arg++;
      i = *arg++;
      if (*arg++ != '-') {
        *buffer++ = '[';
        arg −= 2;
        continue;
      }
      ac = *arg++;
      while (i <= ac) *buffer++ = i++;
      arg++;      /* Skip ']' */
    } else
      *buffer++ = *arg++;
  }
}
...
int main(int argc, char* argv[ ]) {
  int index = 1;
  if (argc > 1 && argv[index][0] == '-') {
    ...
  }
  ...
  expand(argv[index++], ...);
  ...
}
```

Figure 1: A representative example of the kind of non-obvious code, taken from MINIX's `tr` which is difficult to verify by inspection or random testing.

constrained external sources (ranging from the user to network packets) and the code must handle arbitrarily invalid or malevolent values gracefully.

The code illustrates two additional common features. First, it has bugs, which KLEE finds and generates test cases for. Second, KLEE quickly achieves good code coverage, generating 40 test cases which cover all executable lines and all branches in the program in under two minutes.

The goal of KLEE is (1) to hit every line of executable code in the program by running it in all possible ways and (2) to check each line against all possible input values to find if some input could trigger errors.

KLEE's basic strategy is to replace a programs inputs with symbolic variables whose values are initially unconstrained. Program values are represented by formulae instead of actual bits and as the program executes the values for all register and memory locations are tracked with complete precision. This allows KLEE to check at each dangerous operation (assertions, memory accesses) if any value exists that could cause an error and to attempt to drive the program down all feasible paths.

2.1 Testing Process

Through careful design, KLEE makes it easy to start checking many real programs in seconds. It requires no source modifications, specifications, or any manual work on the part of the user other than giving command line values indicating the number and size of files, command line strings, or other inputs to test the code on. The user just needs to compile their code using `llvm-gcc` compiler which behaves exactly as `gcc`, except that it emits LLVM bytecode object files. For example, the `tr` tool is compiled using:

```
llvm-gcc --emit-llvm tr.c -o tr.bc
```

KLEE runs directly on the emitted result and dynamically links in LLVM versions of libc and our environmental model, described later (§ 4). The following command was used to test `tr`:

```
klee --max-time 2 --sym-args 10 10
     --sym-files 1 2000 --max-fail 1 tr.bc
```

The `--max-time` option indicates that KLEE should be run for two minutes, while the `--sym-args` option specifies that the program should be run on up to two command line arguments, each up to 10 characters long. The `--sym-files` option directs the environment to make standard input and one additional file symbolic, each of which contain 2000 bytes of data. Finally, the `--max-fail` option indicates that system calls should be allowed to fail at most one time along each program path (e.g. `read()` returning `EIO`).

During testing, KLEE generates concrete test cases for all program errors and for any path through the program which covers a new instruction or branch. After testing is complete, these test cases can be rerun independently of KLEE through a separate replay driver. This driver uses the test case data to construct appropriate inputs to the program (arguments, files, pipes, etc.) and then runs the program natively. Separating test case evaluation from generation in this fashion ensures that test cases have the correct behavior when run using the native compiler, and allows the use of standard tools (e.g., `gdb`, `gcov`) for debugging and evaluating the test results.

2.2 Symbolic Execution

When KLEE runs the program, it tries to explore every possible path. This is done by executing the program *symbolically*, i.e. tracking all constraints on inputs marked *symbolic* as each instruction is run. When a conditional that depends on a symbolic input is encountered, a constraint solver is used to determine which direction the path will follow. In some cases execution is not constrained to follow a single path – the condition can be true or false depending on the input – and the execution conceptually *forks*. When this happens, KLEE clones the current process and follows both paths, adding the appropriate constraint to the path conditions of each process. To clarify this process, we explain how KLEE finds one of the bugs in `expand()`.

The actual error is on lines 16-18 in Figure 1. The code assumes that an argument containing `'['` will be followed by at least two more characters. However, if the argument ends with `'['`, then the increments to `arg` skip the terminating `'\0'` character of the string and the dereference on line 18 is out of bounds.

Recall that we test `tr` using between 0 and 2 arguments each of up to 10 characters. KLEE executes `tr` with an initial path constraint that $1 \leq \text{argc} \leq 3$ (one extra argument is reserved for the program name) and without constraints on the arguments. In Figure 1 when KLEE reaches line 33 in this procedure it needs to determine which direction the process should take through the branch. To do so, the constraint solver is queried to see if the path condition, i.e. $1 \leq \text{argc} \leq 3$, implies $\text{argc} > 1$ or its negation. In this case, the branch condition can be true ($\text{argc} \in \{2, 3\}$) and false ($\text{argc} = 1$) and execution will fork. The path condition will be updated to $\text{argc} = 1$ in the process following the false path and $2 \leq \text{argc} \leq 3$ in the process following the true path.

Once there are multiple concurrent processes at each instruction step, KLEE must choose which process to execute. Details of the scheduling algorithm are given in Section 3, for now we assume that KLEE follows the path that will reach the bug. As execution continues along this path, KLEE will update the variables `index` and `arg` as appropriate and will fork four more times, again at line 33 in `main` and at lines 3, 4, 15, and 18 in `expand`. Figure 2 shows the branch tree at the point when KLEE detects a buffer overflow. The expressions along interior nodes indicate the places where execution forked and circles represent active processes.

When KLEE encounters a bug or a process exits, the path condition records the entire set of constraints on the input that are necessary to drive the program down that path. The constraint solver is used to determine a concrete set of input values which satisfy all of these constraints which are written out as a test case. For the path that exposed the buffer overflow bug on line 18, KLEE generates the input `argc=2` and `argv[1] = ''[''` (the contents of symbolic files are irrelevant here), which can be rerun on a raw version of `tr` to verify the bug independently of KLEE.

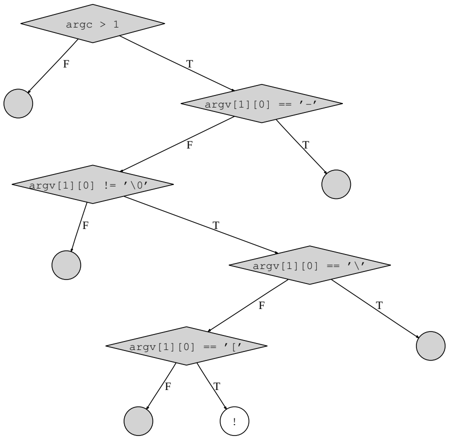

Figure 2: A path to the bug in MINIX's `tr`. Circles represent active processes and the expressions in diamonds indicate places where execution forked.

3 The KLEE Architecture

The KLEE architecture for symbolic execution is a complete redesign of EXE, our previous system [10]. KLEE has been implemented with a focus on precision and scalability. Conceptually, KLEE keeps an explicit model of every possible state that can result from executing the input program symbolicly, including accurate bit-level modeling of the majority of legal C operations[1]

KLEE is implemented as a virtual machine for the Low Level Virtual Machine [25] (LLVM) assembly language. LLVM uses a RISC-like instruction set with an infinite number of registers. Although the instruction set is primarily intended for use as part of the compiler infrastructure, we have found the representation adequate for interpreting directly. Additionally, KLEE provides special *intrinisic* functions which the program can call to create symbolic variables and to communicate with the underlying operating system.

At a high level, KLEE functions as an interesting hybrid between an operating system and an interpreter. Processes are explicitly modeled by their stack, heap, program counter, and path condition. The core of KLEE is a interpreter loop which evaluates instructions until execution is complete. However, unlike a typical interpreter, at each instruction step KLEE selects a process to interpret using a number of search strategies, described in greater detail below. Once a process has been selected, KLEE executes a single instruction in the context of that process:

1 The implementation of most instructions is straightforward. For example, for an add instruction the con-

[1] The current implementation has the following limitations: symbolic floating point and `longjmp` are unsupported and the size of dynamically allocated objects cannot be symbolic.

tents of the argument registers are loaded. If both operands are concrete then the add is performed natively, otherwise an *Add* expression is created from the arguments. In either case the result is written back to the result register.

2. The implementation of most instructions is straightforward. For example, adding two symbolic operands generates the constraint that the result is equal to the sum of the two operands.
3. At a branch instruction, a constraint solver is used to determine if the branch condition must be true or must be false given the current path constraints. If so then execution follows the appropriate path. Otherwise the process is cloned and both paths are followed, with the each child's path condition updated appropriately.
4. At process termination – a return from `main` or an `exit` system call – KLEE queries the constraint solver to determine a set of concrete values that satisfy the process path constraints. These values are used to generate a test case which can be replayed and will follow the same execution path. To avoid generating a large number of uninteresting test cases, by default KLEE only generates test cases for paths which covered new code, either an unexecuted instruction or an untaken branch.
5. At any instruction where an error can occur, for example a memory error or divide by zero, KLEE checks to see if the error is possible along the current path. If so then KLEE creates a test case which will exhibit the error and continues interpreting the current process with the additional constraint that the error does not occur.
6. At a load or store instruction, KLEE determines the set of objects which the target address could point to. If the address could point to multiple objects the process is cloned once for each possible target and each new process adds the constraint that the address is in-bounds of that object. Although this operation is potentially expensive, in practice it does not occur frequently and this implementation greatly simplifies the representation of a memory read or write expression.

3.1 Scalability

The number of possible execution states is exponential in the size of the symbolic input and in practice grows quite quickly. It is not uncommon for KLEE to be simulating tens or even hundreds of thousands of concurrent processes during the first few minutes of interpretation, even for small programs.

To deal with these problems, instead of a flat page-based memory model KLEE uses a memory model where the application can only access memory that is inside an allocated object (i.e. a global variable, stack object, or object obtained via `malloc`). With this representation, KLEE can implement copy-on-write at the level of individual objects which is very effective at minimizing the amount of memory we require per-process. Furthermore, by implementing this structure as a persistent map the heap can be cloned in constant time and portions of the map which are shared among multiple processes do not require additional memory.

Utility	Max. Processes
`echo`	91,912
`pathchk`	51,494
`sort`	29,335
`ls`	15,799

Table 1: Maximum number of KLEEprocesses that fit in 1GB of memory for four of the Coreutils utilities we tested.

Table 1 gives examples for the maximum number of concurrent process which fit in 1GB for a number of the Coreutils applications we tested.

3.2 Process Scheduling

KLEE uses a number of search heuristics to select the process to run at each instruction step. Our basic approach is to interleave two different strategies, each emphasizing a different goal:

1. *Random path selection* maintains a binary tree recording the program path followed for all active processes, i.e. the leaves of the tree are the current processes and the internal nodes are places where execution forked. Processes 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 processes in each subtree will have equal probability of being selected, regardless of their size.

   This strategy has two important properties. First, it favors processes which are high in the branch tree and therefore are relatively unconstrained. It is valuable to select these processes more frequently because they have greater freedom to reach uncovered code. Second, and most importantly, this strategy avoids starvation when some part of the program is rapidly creating new states, i.e. "fork bombing".
2. A strategy which attempts to select states that are likely to cover new code in the immediate future. Heuristics are used to compute a weight for each process and a random process is selected according to these weights. Currently these heuristics use a combination of the minimum distance to an uncovered instruction, taking into account the call stack of the

process, and whether the process has recently covered new code.

These strategies are composed by selecting from each in a round robin fashion. Although this interleaving may increase the time for a particularly effective strategy to achieve high coverage, it protects the system against cases where one individual strategy would become stuck. Furthermore, because the strategies are always selecting processes from the same pool, using interleaving allows the strategies to interact cooperatively.

Finally, once selected each process is run for a "time slice" defined by both a maximum number of instructions and a maximum amount of time. The time to execute an individual instruction can vary widely between simple instructions, like addition, and instructions which may use the constraint solver or fork, like branches or memory accesses. Time-slicing processes helps ensure that a process which is frequently executing expensive instructions will not dominate execution time.

3.3 Query Optimization

Checking with KLEE is almost always dominated by the time it takes to solve the queries made to the underlying constraint solver. Therefore, almost all of our efforts to improve system performance have focused on eliminating or simplifying queries. In particular, KLEE uses two important optimizations which have proven highly effective at reducing query time: *constraint independence* and *counterexample caching*.

The first optimization, constraint independence, takes advantage of the natural decomposition of programs into modular components. This optimization was first implemented for EXE, our previous symbolic execution system [10]. Briefly, constraints can be divided into disjoint *independent* subsets based on the symbolic variables which they reference. By explicitly tracking these subsets, KLEE can frequently eliminate irrelevant constraints in a query prior to passing it to the underlying constraint solver.

Furthermore, due to the nature of symbolic execution, queries have a considerable amount of redundancy. Although a straightforward caching mechanism which memoizes queries is effective at eliminating a large number of queries, it does not take advantage of the additional logical structure of a query. We have developed an alternate mechanism, the counterexample cache, to take full advantage of previous query results.

The counterexample cache functions by caching a map of sets of constraints to counterexamples (i.e. variable assignments), with a special sentinel used when a set of constraints has no solution. This mapping is stored in a custom data structure which allows efficiently searching for cache entries for both subsets and supersets of a constraint set. By storing the cache in this fashion, the counterexample cache gains three additional ways to eliminate queries:

1 When a subset of a constraint set has no solution, then neither does the original constraint set. Adding constraints to an unsatisfiable constraint set cannot make it satisfiable.

2 When a superset of a constraint set has a solution, then this is also a solution for the original constraint set. Dropping constraints from a constraint set does not invalidate a solution to that set.

3 When a subset of a constraint set has a solution, it is likely that this is also a solution for the original set. This is because the extra constraints often do not invalidate the solution to the subset. Since checking a potential solution is cheap, KLEE tries substituting in all solutions for subsets of the constraint set and returns a satisfying solution, if found.

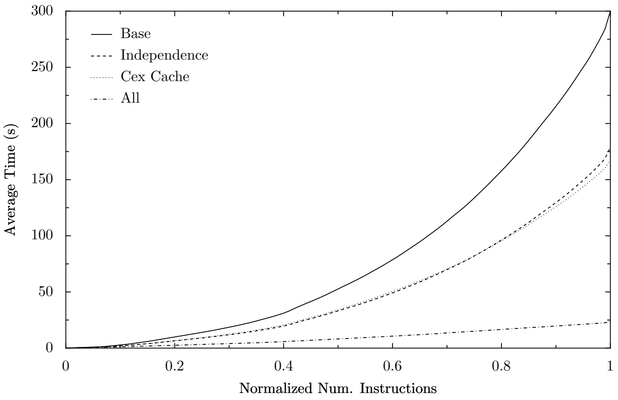

Figure 3: Performance comparison of KLEE's solver optimizations on COREUTILS. Each tool is run for the same number of instructions and results are then normalized and average across all applications.

As an example of the effectiveness of these optimizations, we performed an experiment where all 90 COREUTILS applications were first run for 5 minutes with both of these optimizations turned off. We then reran with constraint independence and the counterexample cache enabled separately and together for the same number of instructions. The results in Figure 3 show an order of magnitude improvement in execution time and indicate that the optimizations scale very well, with each becoming more effective as more instructions are executed.

4 Environment Modeling

Systems code interacts with the environment (e.g. the operating system, the user) in many ways: by reading command-line arguments or environment variables, reading and writing files, checking file metadata such as file permissions and size, sending and receiving packets over the network, and so on. To effectively test such code, we want to explore all legal values that could come from the environment, rather than just a single set of concrete values. For example, checking the permissions of a file should be able to return all possible legal permissions the file could have. Roughly speaking, in KLEE we accomplish this by interposing at each place the user can read environmental data and instead return symbolic data, constrained to obey any required invariants. For example, the bytes of a command-line argument (on Unix: a C string) are entirely unconstrained, except for the last null terminating byte. The code that does this interposition is traditionally called a "model." A key feature of KLEE's models is that they are written in normal C code. As a result, the user can readily customize, extend, or even replace them without having to understand the internals of KLEE. The current models are around 2,500 lines of code. We now describe how KLEE makes the file system symbolic.

4.1 A symbolic file system

Applications read a significant amount of information from the file system: file data itself, metadata information such as file sizes and permissions, directory names, etc. When they attempt to read such information from concrete files and directories, we want things to "just work" as they would when the code is running natively. When they read this information from places that could contain arbitrary data (such as a file provided on the command line), we want the returned values to be symbolic, but constrained to respect any necessary invariants. In this way, we can explore all potential actions, and still have no false positives.

KLEE meets these requirements by providing a simple symbolic file system implementation, and checking on each operation, whether the action is for a concrete file or a symbolic one. In the former case, it calls the corresponding system call in the running operating system, while in the latter case it returns symbolic data, being careful to return the same values for multiple observations of the same object.

Figure 4 gives a rough sketch of the implementation we use for `read()` calls, eliding details needed to make linking work, to handle calls on standard input, and to deal with failures. The code maintains a set of file descriptors, created at file `open()`, and records for each whether the associated file is symbolic or concrete. If the file descriptor `fd` is concrete, our implementation of `read()` accesses the actual disk file by calling the underlying operating system using `pread()` (lines 7-11). We use `pread` because, unlike `read`, it does not affect the position of the file descriptor it is given: Since KLEE's internal processes execute within a single Unix process (the one used to run KLEE), then unless we duplicated file descriptors for each of them (which seemed expensive), a `read` by one would affect all the others.

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

Figure 4: Sketch of KLEE's model for `read()`.

If the file descriptor is symbolic, `read()` just copies out the symbolic file data into the supplied buffer (lines 13-19). Any subsequent constraints on this data will be preserved in the case that `read()` is called again on the same file descriptor and range with no intervening write.

We provide similar symbolic models for the most common system calls, including `open`, `close`, `read`, `write`, `lseek` and `stat`.

Unsurprisingly, the choice of what interface to model has a big impact on model complexity. Rather than having our models at the system call level, we could have instead built them at the C standard library level (`fopen`, `fread`, etc.). Doing so has the potential performance advantage that, for concrete code, we could run these operations natively. The major downside, however, is that the standard library contains a huge number of functions, which would make modeling tedious and error-prone. By only modeling the much simpler, low-level system call API, we can get the richer functionality by just compiling one of the many implementations of the C standard library (we use uClibc [3]) and let it worry about correctness. As a side-effect, we simultaneously check the library for errors as well.

The actual symbolic file system itself is fairly crude, containing only a single directory with N symbolic files in it. KLEE users specify both N and the maximum file size. This symbolic file system coexists with the real file system, so that applications can open both symbolic and concrete files.

The current rule for deciding which `open` calls bind to a symbolic file is that if the program calls `open` with a concrete name, we (attempt to) open the actual file, while if it calls it with a symbolic name, we treat the file as symbolic. Thus, the call:

```
int fd = fopen("/etc/fstab", O_RDNLY);
```

will set `fd` to point to the actual configuration file `/etc/fstab`, while doing the same call with a symbolic command-line argument `argv[1]`:

```
int fd = fopen(argv[1], O_RDNLY);
```

will set `fd` to point to a symbolic file and `argv[1]` constrained to equal this symbolic file's name.

In the case of symbolic files, a call to `open` with an unconstrained symbolic name will match each of the N symbolic files in turn, and will also fail once. Thus, we can regard a call to `open()` as a branch point with $N+1$ possible outcomes, N of which return a file descriptor to one of the symbolic files, and one which fails. For example, given $N = 1$, the second call to `open()` shown in the code above will generate two paths: one in which `fd` points to the single symbolic file in the environment, and one in which `fd` is set to `-1` indicating error.

4.2 Failing system calls

In addition to the kind of failures expected during the normal execution of an application (e.g., file not found, end of file), there are certain failures which are rarely expected (e.g., `write()` fails because the disk is full). We extended the KLEE environment with a *failing mode* in which the system simulates such failures. The motivation for including such failures is twofold: First, not handling such failing situations can lead to unexpected and hard to diagnose bugs. Second, even when applications do include code for dealing with failures, this code is almost never exercised by the regression suite. We made this mode optional since whether such failures are interesting is application-specific — a simple application may not care about disk crashes, while a mail server expends a lot of code to handle such cases. As Section 5 shows, failing system calls does not give large aggregate coverage improvements, but is required to reach the last (tricky) bit of code in many applications with already high coverage.

4.3 Rerunning test cases

A core principle of KLEE is that the test cases it generates can be run on the raw application, independently of KLEE. This completely eliminates any potential problems with the system, makes it easy to confirm and report bugs, and to generate test suites.

Thus, when an application interacts with the symbolic environment, a test case generated by KLEE includes a concrete instantiation of the symbolic environment for the path explored. That is, it contains concrete command-line arguments, and the set of all the files and their associated data and metadata that were accessed on the path explored by KLEE.

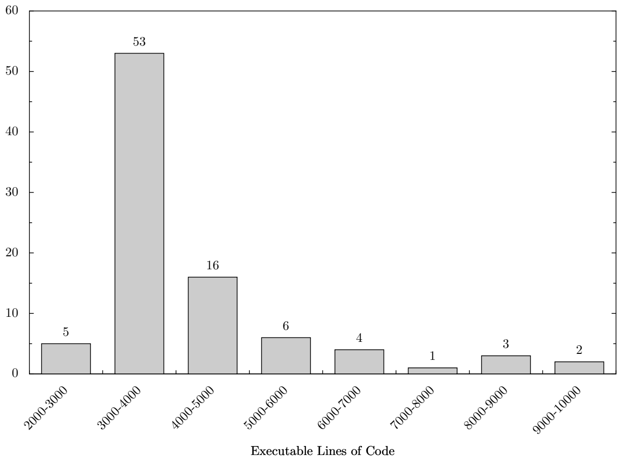

Figure 5: Histogram showing the number of COREUTILS tools that have a given number of executable lines of code (ELOC), including library code. Most tools (53) have between 3K and 4K ELOC.

Running a test case then simply means creating these files on the running file system. Since our symbolic file system consist of N symbolic files in the current directory, the test case will consist of the description of N files (names, data, metadata) that we can easily create in the current directory. We can then run the application on the generated command-line arguments. The only challenge in this case is running a test case where certain system calls fail. In order to run these test cases outside of KLEE, we constructed a simple utility that *ptraces* the given application in the manner of a debugger, and skips the system calls that were supposed to fail, returning instead an error to the ptraced application.

5 Evaluation

This section gives our coverage results and bugs found for COREUTILS (§ 5.1), MINIX (§ 5.1.5), and HiStar (§5.3). We also give preliminary measurements of the effectiveness of KLEE at finding deep correctness errors (§ 5.2).

5.1 GNU Coreutils

This section reports the results of using KLEE to check all 90 tools that are part of the GNU COREUTILS suite of utilities. Previous work, ours included, has evaluated constraint-based execution on a small number of hand-selected benchmarks. To the best of our knowledge, COREUTILS contains an order of magnitude more programs than prior work has attempted to test.

Figure 5 breaks down the tools by executable lines of code (ELOC), including library code the tool calls. For COREUTILS, ELOC are usually a factor of 3 smaller than actual lines of code. It's clear that the tools are not toys — the smallest have over 2K ELOC, over half (53) have more than 3K, and ten have over 6K.

5.1.1 Methodology

With a single exception, we ran all of GNU COREUTILS with no modifications. The exception was `sort`, which required a one-line change to shrink an overlay large buffer that made process size unmanageable.

Almost all tools were tested using the same command:

```
./run <tool-name> --sym-args 10 2 2
                  --sym-files 2 8
                  [--max-fail 1]
                  --max-time=60
```

which tells KLEE to run the given tool with up to three arguments, the first one (if present) being of length at most 10, and the next two (if present) of length at most 2. The option `--sym-files` specifies a symbolic environment with two symbolic files, one of which is `stdin`, each containing 8 bytes. The `--max-fail` option specifies that the system should fail at most one system call on each path; we show this option inside brackets because we run both with and without this option. Finally, the `--max-time` option specifies that each tool should be run for at most 60 minutes.

For eight tools where the coverage results were unsatisfactory, we consulted the `man` page and increased the number and size of files and argument strings.

After KLEE produced test cases, we conservatively measured how comprehensive they were by recording statement coverage. We chose statement coverage because it is widely-understood and uncontroversial. Note, however, that it dramatically underestimates KLEE's capability of exploring each statement on many different paths (potentially all of them) with all possible values.

We do a hard end-to-end check of coverage by running the generated test cases on a stand-alone version of the tool that has been compiled using for instrumentation with `gcov`. Doing this measurement independently of our system completely eliminates the effect of bugs in KLEE and verifies that the produced test case does, in fact, run the code it claims.

Similarly, concrete test cases also allow bug confirmation independently of KLEE, by running the program on the test case for a given error. As a result, a version of the test cases for all previously unknown bugs we reported have now been included in the official GNU COREUTILS test suite.

Coverage (w/o lib code)	**Number of tools**	**Avg. # ELOC (w/ called lib code)**
100%	16	3307
90-100%	38	3958
80-90%	22	5013
70-80%	8	4199
60-70%	6	5217

Table 2: Number of COREUTILS tools which achieve statement coverage in the given ranges. Note, as we discuss (§ 5.1.2), to avoid double-counting our coverage, numbers here and in the other figures exclude library code (which gets shared by many applications). No tool gets less than 60% coverage. The rightmost column shows the average ELOC for tools within each range, including called library code (again, see text).

We made sure to report results for the entire COREUTILS suite, the worst along with the best. We made the decision from the beginning to do so, preventing us from (even unintentionally) cheating through the use of fragile optimizations that would blow up on some (or even many) applications.

5.1.2 Coverage Results

Table 2 gives aggregate statement coverage results: KLEE gets 100% statement coverage on 16 tools, over 90% on 54 tools, and over 80% statement coverage on 76 tools (84.4% of all tools). The minimum coverage achieved on any tool is 62.0%, and the average coverage across all tools is 81.9%.

We see such high coverage on a broad swath of applications "out of the box" as a convincing demonstration in the power of the approach, especially since it is across the entire tool suite rather than from just cherry-picking (say) the best 54 performers.

Note that we do not count coverage of library code in our measurements since it makes them harder to interpret:

1 Including library code in the coverage percentages we report would double-count many lines, since often the same library function is called by many applications.

2 Doing so would also unfairly under-count coverage. For a given application, often much of a library function is dead code for the reason that library code is general but the call sites are not. For example, `printf` is exceptionally complex, but the call `printf(‘‘hello’’)` can only hit a small a fraction (missing the code to print integers, floating point, formatting, etc.).

However, in terms of the raw size of the application, the

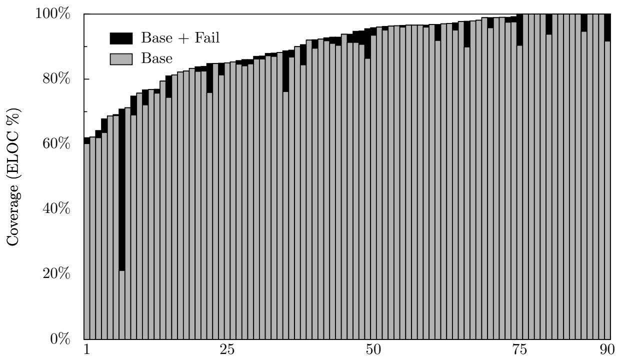

Figure 6: Statement coverage for each application with and without failures.

total executable lines of code (including called library code) is interesting: KLEE must be able to handle this library code (and gets no credit for doing so in terms of coverage) in order to exercise the code in the tool itself.

Figure 6 further shows the coverage achieved on each of the 90 COREUTILS tools, with and without triggering failing system calls (§4.2). Exploring the failure path of system calls is mostly useful for hitting the last few lines of high-coverage tools, rather than significantly improving the results overall (which it only improves from 79.4% to 81.9%). The one exception is `pwd` which requires system call failures to improve from 21.2% to 70.8%. The next largest coverage improvement for a single tool is a more modest (but still notable) 12.5% extra coverage.

5.1.3 Comparison against developer test suites

Each utility in COREUTILS comes with an extensive manually-written test suite, extended each time a new bug fix or extra feature is added. An obvious experiment is to see how well KLEE does in comparison. Overall, the developers get 67.5%, while KLEE gets 81.9%. Thus, a 90 hour run of KLEE (1 hour per application) exceeds the coverage of test suites built over a period of fifteen years by over 14%!

Figure 7 gives a relative view of KLEE versus developer tests by subtracting the lines hit by manual testing from those hit by KLEE and dividing this by the total possible. A bar above zero indicates that KLEE beat the manual test (and by how much). A bar below measures the opposite. KLEE beats manual testing (sometimes significantly) on the vast majority of the applications.

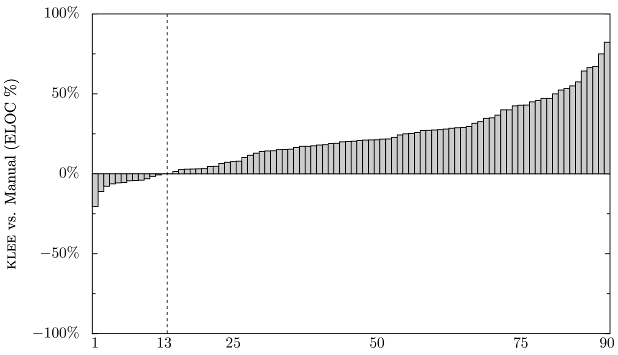

Figure 7: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 13 applications, often significantly.

```
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
seq -f %0 1
```

```
t1.txt: "\t \tMD5("
t2.txt: "\b\b\b\b\b\b\b\t"
t3.txt: "\n"
```

Figure 8: Command lines and inputs which trigger the bugs found by KLEE in COREUTILS version 6.10. All bugs cause program crashes on our Intel Pentium D machine running Fedora Core 7 with SELinux.

5.1.4 Bugs found

We found nine bugs in the latest version of COREUTILS (version 6.10), in `md5sum`, `mkdir`, `mkfifo`, `mknod`, `paste`, `pr`, `ptx`, `seq` and `tac`. All of these were crash bugs, most caused by inadvertent memory overflows. The bug in `seq` had been already fixed in the developers' version, but all the other bugs received immediate attention from the developers, and were each confirmed and fixed within two days of our report.

A list of test inputs which trigger bugs on our systems is shown in Figure 8. The first three were around since 1992 so should theoretically crash any COREUTILS distribution. The next three (which use the `-Z` option) require SELinux. The others are more recent, and do not crash older COREUTILS distributions.

As an illustrative example, we discuss the bug that we found in `pr`, a tool used for paginating files before printing. Figure 8 shows a simple test case that KLEE generates to segfault `pr`, invoking it with flag `-e`, which tells it to expand tabs to spaces, on a file containing a series

```
 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: }
```

Figure 9: Code snippet from `pr` where a memory overflow of `clump_buff` via pointer `s` is possible if `chars_per_input_tab == chars_per_c` and `input_position < 0`.

of backspace characters followed by a tab.

Figure 9 shows the portion of the code containing the bug. On the path where the bug occurs, both `chars_per_input_tab` and `chars_per_c` are equal to the tab width (let's call it T). Line 2665 (via the macro on line 602) computes `width` as $(T-$ `input_position` $\bmod T)$. The root cause of the bug is the incorrect assumption that $0 \leq x \bmod y < y$, which only holds for positive integers. When `input_position` is positive, `width` will be indeed less than T since $(0 \leq$ `input_position` $\bmod T < T)$. However, in the presence of backspaces, `input_position` can become negative, so $(-T <$ `input_position` $\bmod T < T)$. Consequently, `width` can be as large as $(2 \times T - 1)$.

The bug arises when the code allocates a buffer `clump_buff` of size T (line 1322) and then writes `width` characters into this buffer (lines 2669–2670) via the pointer `s` (initially set to `clump_buff`). Because `width` can be as large as $(2\times T-1)$, a memory overflow is possible. Note that the tab width T can be specified from the command line, and thus this is an unbounded buffer overflow.

This bug is representative of the bugs found by KLEE in COREUTILS: complex, non-obvious code which is hard to reason about manually. As a consequence, this bug has been present in `pr` for more than 15 years, since at least 1992 when COREUTILS was first added to a CVS repository.

5.1.5 MINIX Utilities

MINIX has its own utility suite with many of the same programs as in GNU COREUTILS. As a quick check to ensure our results were not somehow COREUTILS specific, we ran KLEE on 14 simple MINIX utilities. We found two buffer overflows and got 90.6% overall coverage.

5.2 Checking tool equivalence

When KLEE reaches an `assert` or a similar error checking `if` statement, it tries to drive execution down both branches. Thus, if KLEE can hit the error on a certain path, then it will. Conversely, if the condition leading to the error is not satisfiable on a path, then KLEE can prove full correctness along that path.

Assume we have two procedures `int p(int x)` and `int p'(int x)` that purport to implement the same interface. For example, `p` and `p'` could be two different implementations of the same library function, or perhaps `p` is a simple reference implementation and `p'` a heavily optimized version. Then, running roughly the following code with KLEE will check `p` and `p'` for equivalence:

```
int x;
make_symbolic(&x);
assert(p(x) == p'(x));
```

When a path reaches the `assert`, if any possible value of the constraints on that path could violate the `assert` (and the constraint solver can reason about all constraints), then KLEE will generate a test case that does so. If at least one implementation is correct on that path, then such a mismatch is a correctness violation in the other.

Conversely, if the constraint solver shows such a value does not exist, then we have *proved* that the two implementations are equivalent for all values on the checked path. These are both powerful results, completely beyond the reach of traditional testing. One way to look at KLEE is that it automatically translates a path through a C program into a form that a theorem prover can reason about. As a result, proving path equivalence just takes a few lines of C code (the assertion above), rather than an enormous manual exercise in theorem proving.

There are many applications of this basic approach. For example, `p'` could be a patched version of `p` that purports to only remove bugs, and so should have strictly fewer crashes. Or we may have a function and its inverse (such as `compress` and `uncompress`) and so can check that `assert(uncompress(compress(x)) == x)`.

We checked the equivalence of the MINIX tools discussed in §5.1.5 against the COREUTILS implementations. For example, given the same input, the MINIX and COREUTILS versions of `wc` should output the same number of lines, words and bytes, regardless of how the tool is implemented internally. In fact, all Unix utilities should conform to IEEE Standard 1003.1 [1], and both MINIX and COREUTILS suites intend to do so.

Tool	Input	MINIX	COREUTILS
wc	`0:0`	`0 0 3`	`0 1 3`
wc	`0\t0`	`0 1 3`	`0 2 3`
wc	`0\n0`	`1 1 3`	`1 2 3`
wc	`0\f\r`	`1 1 3`	`0 1 3`
basename	`""`	`/`	`""`
basename	`-- PP`	`--`	`PP`
printf	`\\`	`""`	`\`
printf	`%%*`	`%`	`%*`
printf	`%i`	`""`	`0`
printf	`%x -2`	fffffffe	fffffffffffffffe
printf	`%%r ""`	`%%s`	`%r`
printf	`%is ""`	-1077520586s	0s
fold -w 2	`\t`	`\t`	`\n\t`
fold -w 2	`\t\t\t`	`\n\t\n\t\n\t`	`\t\n\t\n\t`

Table 3: Mismatches automatically detected by KLEE

We have not yet added automatic cross-checking to KLEE. We currently crosscheck by manually including the MINIX tool into the COREUTILS program and rename any conflicting identifiers. Because of the required manual work, we only crosschecked four applications. However, we hope that this will convey the general idea behind this KLEE capability. We plan to do a full study demonstrating this technique in future work.

The input to a Unix tool consists of the command-line options and the input files. For the tools we checked, the output is written on `stdout`. Thus, our system runs both the MINIX and the COREUTILS implementation of a tool on identical inputs, and compares the characters written on `stdout`. When a mismatch is detected, the system generates a test case, which is subsequently run on a GCC-compiled version of each tool, to confirm the mismatch. Table 3 shows several mismatches automatically detected by KLEE between the MINIX and COREUTILS versions of `wc`, `basename`, `printf` and `fold`. These mismatches reveal several bugs in the MINIX versions of the tools. For example, `wc` incorrectly counts the number of words and the number of lines in certain cases [2] and `fold` incorrectly adds an additional newline when it encounters a tab at the end of a line. The `printf` tool reveals a large number of mismatches between the two versions (we presented only a small sample in Table 3), while the two `basename` versions disagree on only two inputs.

5.3 The HiStar OS Kernel

To demonstrate the applicability of our system to other forms of system code we applied KLEE to testing a user-mode version of the HiStar [29] operating system kernel.

```
static void test(void *upage, unsigned num_calls) {
    klee_make_symbolic(upage, PGSIZE, "upage");
    for (int i=0; i<num_calls; i++) {
        uint64_t args[8];
        for (int j=0; j<8; j++)
            klee_make_symbolic(&args[j], sizeof(args[j]), "arg");
        kern_syscall(args[0], args[1], args[2], args[3],
                             args[4], args[5], args[6], args[7]);
    }
    sys_self_halt();
}
```

Figure 10: Our test driver for the HiStar kernel. The test makes a single page of user memory symbolic and executes a user-specified number of system calls (which may refer to the given page) with entirely symbolic arguments.

Test	Random	KLEE	ELOC
With Disk	50.1%	67.1%	4617
No Disk	48.0%	76.4%	2662

Table 4: Coverage on the HiStar kernel for runs with up to three system calls, configured with and without a RAM disk. For comparison, we implemented a test driver which calls a random system call and uses random values for all other inputs. This driver was run one million times, with and without a disk.

To do so we use a user-mode version of the kernel which uses an optional RAM disk and a small amount of core memory.

This kernel uses a simplified bootstrap procedure which creates the core kernel data structures and initializes a single thread with access to a single page of user memory. Once loaded, this thread executes the test procedure shown in Figure 10, which makes the user memory symbolic and executes a user-specified number of system calls using entirely symbolic arguments.

Although this environment may seem very restrictive, in practice we have found that this approach is able to quickly generate test cases — sequences of system call vectors and memory contents — which cover a large portion of the HiStar kernel and uncover interesting behaviors. Table 4 shows the coverage obtained for the core kernel for runs with and without a disk. When configured with a disk, a majority of the uncovered code can only be triggered when there are a large number of kernel objects. This currently does not happen in our testing environment; we are investigating ways to exercise this code adequately during testing.

We also tested HiStar using a version of our driver which select a random system call number and uses random values for all other inputs. The results from running this driver one million times are also shown in Table 4.

[2] On the upside, the MINIX version always gets the number of bytes correct!

```
uintptr_t safe_addptr(int *of, uint64_t a, uint64_t b) {
   uintptr_t r = a + b;
   if (r < a)
      *of = 1;
   return r;
}
```

Figure 11: HiStar function containing an important security vulnerability. The function is supposed to set `*of` to true if the addition overflows but can fail to do so in the 32-bit version for very large values of `b`.

KLEE's tests achieve significantly more coverage than random testing both for runs with (+17.0%) and without (+28.4%) a disk device.

In addition to generating tests which cover a substantial portion of the kernel, our testing found a critical security bug in the 32-bit version of HiStar. The `safe_addptr` function containing the bug is shown in Figure 11. The function is supposed to set `*of` to true if the addition overflows. However, because the inputs are 64 bits the test used is insufficient (it should be `(r < a) || (r < b)`) and the function can fail to indicate overflow for large values of `b`.

The `safe_addptr` function is used to implement HiStar's validation of user memory addresses prior to copying data to or from user space. A kernel routine takes a user address and a size and computes if the user is allowed to access the memory in that range; this routine uses the overflow to prevent access when a computation could overflow. This bug in computing overflow therefore allows a malicious process to gain access to memory regions outside its control for system calls where the user can pass in an arbitrarily large size.

6 Related Work

Researchers have recently designed a variety of tools based on symbolic execution [9, 19, 10, 28, 18, 20, 6, 15, 14, 8, 16]. We discuss how our work compares in the way it addresses (1) the environment problem and (2) the path explosion problem.

To the best of our knowledge, traditional symbolic execution systems [11, 12, 24] are static in a strict sense and do not call out into the live running environment at all. They either cannot handle programs that make use of the environment or require a complete working model. More recent work in test generation [10, 19, 28] does allow external interactions but forces them to use entirely concrete procedure call arguments. This approach allows these tools to check this kind of code, but prevents them from exploring more behaviors: a concrete external call will do exactly what it did, rather than all things it could potentially do. In a sense, KLEE combines the best of both worlds: when calls already have concrete arguments, it can call the external environment (allowing it to handle a broad range of programs), but it also provides a facility to judiciously make symbolic those parts of the environment that are interesting in terms of generating behaviors.

The path explosion problem has instead received more attention [6, 18, 20, 26, 16]. Similarly to the search heuristics presented in Section 3, search strategies proposed in the past include Best First Search [10], Generational Search [20], and Hybrid Concolic Testing [26]. Orthogonal to search heuristics, researchers have addressed the path explosion problem by testing paths compositionally [18, 5], and by tracking the values read and written by the program [6].

Like KLEE, other symbolic execution systems implement their own optimizations before sending the queries to the underlying constraint solver, such as the simple syntactic transformations presented in [28], and the *constraint subsumption* optimization discussed in [20].

Similar to symbolic execution systems, model checkers have been used to find bugs in both the design and the implementation of software [21, 22, 7, 13, 17]. These approaches often require a lot of manual effort to build test harnesses. However, to some degree, the approaches are complementary to KLEE: the tests KLEE generates could be used to drive the model checked code, similar to the approach embraced by the Java PathFinder (JPF) project [23].

7 Conclusion

The long-term goal of our work is to be able to take an arbitrary program and routinely get 90%+ code coverage, crushing it under test cases that explore all interesting paths with all interesting values. While there is still a long way to go to reach this goal, the results in this paper show that the approach can get high code coverage over a broad range of real applications, coverage that exceeded that of a high-quality, manual test suites constructed incrementally over a period of 15 years, as well as finding bugs that had been around over a decade. The techniques we describe should work well with other tools and give similar help in handling a broad class of applications.

8 Acknowledgements

We would like to thank the GNU COREUTILS developers, and in particular Jim Meyering, the maintainer of COREUTILS, for promptly confirming our bug reports and fixing the bugs, as well as providing us with a variety of useful information about COREUTILS.

We would also like to thank Nickolai Zeldovich, the designer of HISTAR, for his great help in checking HISTAR and in particular for providing us with a user-level driver.

We also thank Philip Guo for his careful proofreading and valuable comments on the text.

References

[1] IEEE Std 1003.1, 2004 edition. `http://www.unix.org/version3/ieee_std.html`, May 2008.

[2] Security focus website, `http://www.securityfocus.com`, March 2008.

[3] uCLibc website. `http://www.uclibc.org/`, May 2008.

[4] United States National Vulnerability Database website, `http://nvd.nist.gov`, March 2008.

[5] ANAND, S., GODEFROID, P., AND TILLMANN, N. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)* (2008).

[6] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)* (2008).

[7] BRAT, G., HAVELUND, K., PARK, S., AND VISSER, W. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)* (2000).

[8] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[9] CADAR, C., AND ENGLER, D. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software* (August 2005).

[10] CADAR, C., GANESH, V., PAWLOWSKI, P., DILL, D., AND ENGLER, D. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (October-November 2006).

[11] CLARKE, E., AND KROENING, D. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003* (January 2003), IEEE Computer Society Press, pp. 308–311.

[12] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 168–176.

[13] CORBETT, J., DWYER, M., HATCLIFF, J., LAUBACH, S., PASAREANU, C., ROBBY, AND ZHENG, H. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000* (2000).

[14] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing software by blocking bad input. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP)* (October 2007).

[15] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: end-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)* (October 2005).

[16] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis (ISSTA'07)* (2007), ACM.

[17] GODEFROID, P. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages* (1997).

[18] GODEFROID, P. Compositional dynamic test generation. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL'07)* (Jan. 2007).

[19] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (Chicago, IL USA, June 2005), ACM Press.

[20] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of Network and Distributed Systems Security* (2008), pp. 151–166.

[21] HOLZMANN, G. J. The model checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.

[22] HOLZMANN, G. J. From code to models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design* (Newcastle upon Tyne, U.K., 2001), pp. 3–10.

[23] KHURSHID, S., PASAREANU, C. S., AND VISSER, W. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2003).

[24] KROENING, D., CLARKE, E., AND YORAV, K. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003* (2003), ACM Press, pp. 368–371.

[25] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization* (Washington, DC, USA, 2004), IEEE Computer Society, p. 75.

[26] MAJUMDAR, R., AND SEN, K. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)* (May 2007).

[27] MILLER, B., KOSKI, D., LEE, C. P., MAGANTY, V., MURTHY, R., NATARAJAN, A., AND STEIDL, J. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Tech. rep., University of Wisconsin - Madison, 1995.

[28] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A concolic unit testing engine for C. In *In 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)* (Sept. 2005).

[29] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 19–19.