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 used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat the coverage of the developers' own hand-written test suite. When we did the same for 75 equivalent tools in the BUSYBOX embedded system suite, results were even better, including 100% coverage on 31 of them.

We also used KLEE as a bug finding tool, applying it to 452 applications (over 430K total lines of code), where it found 56 serious bugs, including three in COREUTILS that had been missed for over 15 years. Finally, we used KLEE to crosscheck purportedly identical BUSYBOX and COREUTILS utilities, finding functional correctness errors and a myriad of inconsistencies.

1 Introduction

Many classes of errors, such as functional correctness bugs, are difficult to find without executing a piece of code. The importance of such testing — combined with the difficulty and poor performance of random and manual approaches — has led to much recent work in using *symbolic execution* to automatically generate test inputs [11, 14–16, 20–22, 24, 26, 27, 36]. 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, on each path maintaining 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 for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

Results are promising. However, while researchers have shown such tools can sometimes get good coverage and find bugs on a small number of programs, it has been an open question whether the approach has any hope of consistently achieving high coverage on real applications. Two common concerns are (1) the exponential number of paths through code and (2) the challenges in handling code that interacts with its surrounding environment, such as the operating system, the network, or the user (colloquially: "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 for robust, deep checking of a broad range of applications, leveraging several years of lessons from our previous tool, EXE [16]. KLEE employs a variety of constraint solving optimizations, represents program states compactly, and uses search heuristics to get high code coverage. Additionally, it uses a simple and straightforward approach to dealing with the external environment. These features improve KLEE's performance by over an order of magnitude and let it check a broad range of system-intensive programs "out of the box."

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

Second, we show that KLEE's automatically-generated tests get high coverage on a diverse set of real, complicated, and environmentally-intensive programs. Our most in-depth evaluation applies KLEE to all 89 programs [1] in the latest stable version of GNU COREUTILS (version 6.10), which contains roughly 80,000 lines of library code and 61,000 lines in the actual utilities [2]. These programs interact extensively 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 many Unix systems. They are used daily by millions of people, bug fixes are handled promptly, and new releases are pushed regularly. Moreover, their extensive interaction with the environment stress-tests symbolic execution where it has historically been weakest.

Further, finding bugs in COREUTILS is hard. They are arguably the single most well-tested suite of open-source applications available (e.g., is there a program the reader has used more under Unix than "`ls`"?). In 1995, random testing of a subset of COREUTILS utilities found markedly fewer failures as compared to seven commercial Unix systems [35]. The last COREUTILS vulnerability reported on the SecurityFocus or US National Vulnerability databases was three years ago [5, 7].

In addition, we checked two other UNIX utility suites: BUSYBOX, a widely-used distribution for embedded systems [1], and the latest release for MINIX [4]. Finally, we checked the HISTAR operating system kernel as a contrast to application code [39].

Our experiments fall into three categories: (1) those where we do intensive runs to both find bugs and get high coverage (COREUTILS, HISTAR, and 75 BUSYBOX utilities), (2) those where we quickly run over many applications to find bugs (an additional 204 BUSYBOX utilities and 77 MINIX utilities), and (3) those where we crosscheck equivalent programs to find deeper correctness bugs (67 BUSYBOX utilities vs. the equivalent 67 in COREUTILS).

In total, we ran KLEE on more than 452 programs, containing over 430K total lines of code. To the best of our knowledge, this represents an order of magnitude more code and distinct programs than checked by prior symbolic test generation work. Our experiments show:

1 KLEE gets high coverage on a broad set of complex programs. Its automatically generated tests covered 84.5% of the total lines in COREUTILS and 90.5% in BUSYBOX (ignoring library code). On average these tests hit over 90% of the lines in each tool (median: over 94%), achieving perfect 100% coverage in 16 COREUTILS tools and 31 BUSYBOX tools.
2 KLEE can get significantly more code coverage than a concentrated, sustained manual effort. The roughly 89-hour run used to generate COREUTILS line coverage beat the developers' own test suite — built incrementally over fifteen years — by 16.8%!
3 With one exception, KLEE achieved these high-coverage results on unaltered applications. The sole exception, `sort` in COREUTILS, required a single edit to shrink a large buffer that caused problems for the constraint solver.
4 KLEE finds important errors in heavily-tested code. It found ten fatal errors in COREUTILS (including three that had escaped detection for 15 years), which account for more crashing bugs than were reported in 2006, 2007 and 2008 combined. It further found 24 bugs in BUSYBOX, 21 bugs in MINIX, and a security vulnerability in HISTAR– a total of 56 serious bugs.
5 The fact that KLEE test cases can be run on the raw version of the code (e.g., compiled with `gcc`) greatly simplifies debugging and error reporting. For example, all COREUTILS bugs were confirmed and fixed within two days and versions of the tests KLEE generated were included in the standard regression suite.
6 KLEE is not limited to low-level programming errors: when used to crosscheck purportedly identical BUSYBOX and GNU COREUTILS tools, it automatically found functional correctness errors and a myriad of inconsistencies.
7 KLEE can also be applied to non-application code. When applied to the core of the HISTAR kernel, it achieved an average line coverage of 76.4% (with disk) and 67.1% (without disk) and found a serious security bug.

The next section gives an overview of our approach. Section 3 describes KLEE, focusing on its key optimizations. Section 4 discusses how to model the environment. The heart of the paper is Section 5, which presents our experimental results. Finally, Section 6 describes related work and Section 7 concludes.

2 Overview

This section explains how KLEE works by walking the reader through the testing of MINIX's `tr` tool. Despite its small size — 169 lines, 83 of which are executable — it illustrates two problems common to the programs we check:

1 *Complexity.* The code aims 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. Figure 1 gives a representative snippet.

[1] We ignored utilities that are simply wrapper calls to others, such as `arch` ("`uname -m`") and `vdir` ("`ls -l -b`").

2 *Environmental Dependencies.* Most of the code is controlled by values derived from 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. Since inputs can be invalid (or even malicious), the code must handle these cases gracefully. It is not trivial to test all important values and boundary cases.

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: in two minutes it generates 37 tests that cover all executable statements. [2]

KLEE has two goals: (1) hit every line of executable code in the program and (2) detect at each dangerous operation (e.g., dereference, assertion) if *any* input value exists that could cause an error. KLEE does so by running programs *symbolically*: unlike normal execution, where operations produce concrete values from their operands, here 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 (called its *path condition*) to produce a test case that will follow the same path when rerun on an unmodified version of the checked program (e.g, compiled with `gcc`).

KLEE is designed so that the paths followed by the unmodified program will always follow the same path KLEE took (i.e., there are no false positives). However, non-determinism in checked code and bugs in KLEE or its models have produced false positives in practice. The ability to rerun tests outside of KLEE, in conjunction with standard tools such as `gdb` and `gcov` is invaluable for diagnosing such errors and for validating our results.

We next show how to use KLEE, then give an overview of how it works.

2.1 Usage

A user can start checking many real programs with KLEE in seconds: KLEE typically requires no source modifications or manual work. Users first compile their code to bytecode using the publicly-available LLVM compiler [33] for GNU C. We compiled `tr` using:

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

Users then run KLEE on the generated bytecode, optionally stating the number, size, and type of symbolic inputs to test the code on. For `tr` we used the command:

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

```
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: }
```

Figure 1: Code snippet from MINIX's `tr`, representative of the programs checked in this paper: 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 numbered on the right hand side.

The first option, `--max-time`, tells KLEE to check `tr.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. [3] The option `--sym-files 2 2000` says to use standard input and one file, each holding 2000 bytes of symbolic data. The option `--max-fail 1` says to fail at most one system call along each program path (see § 4.2).

2.2 Symbolic execution with KLEE

When KLEE runs on `tr`, it finds a buffer overflow error at line 18 in Figure 1 and then produces a concrete test

[2] The program has one line of dead code, an unreachable return statement, which, reassuringly, KLEE cannot run.

[3] Since strings in C are zero terminated, this essentially generates arguments of *up to* that size.

case (`tr [ "" ""`) that hits it. Assuming the options of the previous subsection, KLEE runs `tr` as follows:

1 KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.
2 When KLEE hits 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. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint `argc` > 1 on the false path and `argc` ≤ 1 on the true path.
3 Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a "`*`"): twice on line 33, and then on lines 3, 4, and 15 in `expand`.
4 At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating `'\0'` and points to invalid memory.
5 KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ "" ""`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).

3 The KLEE Architecture

KLEE is a complete redesign of our previous system EXE [16]. At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter. Each symbolic process has a register file, stack, heap, program counter, and path condition. To avoid confusion with a Unix process, we refer to KLEE's representation of a symbolic process as a *state*. Programs are compiled to the LLVM [33] assembly language, a RISC-like virtual instruction set. KLEE directly interprets this instruction set, and maps instructions to constraints without approximation (i.e. bit-level accuracy). [4]

[4] KLEE does not currently support: symbolic floating point, `longjmp`, threads, and assembly code. Additionally, memory objects are required to have concrete sizes.

3.1 Basic architecture

At any one time, KLEE may be executing a large number of states. The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state. This loop continues until there are no states remaining, or a user-defined timeout is reached.

Unlike a normal process, storage locations for a state — registers, stack and heap objects — refer to expressions (trees) instead of raw data values. The leaves of an expression are symbolic variables or constants, and the interior nodes come from LLVM assembly language operations (e.g., arithmetic operations, bitwise manipulation, comparisons, and memory accesses). Storage locations which hold a constant expression are said to be *concrete*.

Symbolic execution of the majority of instructions is straightforward. For example, to symbolically execute an LLVM add instruction:

```
%dst = add i32 %src0, %src1
```

KLEE retrieves the addends from the `%src0` and `%src1` registers and writes a new expression `Add(%src0, %src1)` to the `%dst` register. For efficiency, the code that builds expressions checks if all given operands are concrete (i.e., constants) and, if so, performs the operation natively, returning a constant expression.

Conditional branches take a boolean expression (branch condition) and alter the instruction pointer of the state based on whether the condition is true or false. KLEE queries the constraint solver to determine if the branch condition is either provably true or provably false along the current path; if so, the instruction pointer is updated to the appropriate location. Otherwise, both branches are possible: KLEE clones the state so that it can explore both paths, updating the instruction pointer and path condition on each path appropriately.

Potentially dangerous operations implicitly generate branches that check if any input value exists that could cause an error. For example, a division instruction generates a branch that checks for a zero divisor. Such branches work identically to normal branches. Thus, even when the check succeeds (i.e., an error is detected), execution continues on the false path, which adds the negation of the check as a constraint (e.g., making the divisor not zero). If an error is detected, KLEE generates a test case to trigger the error and terminates the state.

As with other dangerous operations, load and store instructions generate checks: in this case to check that the address is in-bounds of a valid memory object. However, load and store operations present an additional complication. The most straightforward representation of the memory used by checked code would be a flat byte array. In this case, loads and stores would simply map to

array read and write expressions respectively. Unfortunately, our constraint solver STP would almost never be able to solve the resultant constraints (and neither would the other constraint solvers we know of). Thus, as in EXE, KLEE maps every memory object in the checked code to a distinct STP array (in a sense, mapping a flat address space to a segmented one). This representation dramatically improves performance since it lets STP ignore all arrays not referenced by a given expression.

Many operations (such as bound checks or object-level copy-on-write) require object-specific information. If a pointer can refer to many objects, these operations become difficult to perform. For simplicity, KLEE sidesteps this problem as follows. When a dereferenced pointer `p` can refer to N objects, KLEE clones the current state N times. In each state it constrains `p` to be within bounds of its respective object and then performs the appropriate read or write operation. Although this method can be expensive for pointers with large points-to sets, most programs we have tested only use symbolic pointers that refer to a single object, and KLEE is well-optimized for this case.

3.2 Compact state representation

The number of states grows quite quickly in practice: often even small programs generate tens or even hundreds of thousands of concurrent states during the first few minutes of interpretation. When we ran COREUTILS with a 1GB memory cap, the maximum number of concurrent states recorded was 95,982 (for `hostid`), and the average of this maximum for each tool was 51,385. This explosion makes state size critical.

Since KLEE tracks all memory objects, it can implement copy-on-write at the object level (rather than page granularity), dramatically reducing per-state memory requirements. By implementing the heap as an immutable map, portions of the heap structure itself can also be shared amongst multiple states (similar to sharing portions of page tables across `fork()`). Additionally, this heap structure can be cloned in constant time, which is important given the frequency of this operation.

This approach is in marked contrast to EXE, which used one native OS process per state. Internalizing the state representation dramatically increased the number of states which can be concurrently explored, both by decreasing the per-state cost and allowing states to share memory at the object (rather than page) level. Additionally, this greatly simplified the implementation of caches and search heuristics which operate across all states.

3.3 Query optimization

Almost always, the cost of constraint solving dominates everything else — unsurprising, given that KLEE generates complicated queries for an NP-complete logic. Thus, we spent a lot of effort on tricks to simplify expressions and ideally eliminate queries (no query is the fastest query) before they reach STP. Simplified queries make solving faster, reduce memory consumption, and increase the query cache's hit rate (see below). The main query optimizations are:

Expression Rewriting. The most basic optimizations mirror those in a compiler: e.g., simple arithmetic simplifications (`x + 0 = x`), strength reduction (`x * 2`^{n} `= x << n`), linear simplification (`2*x - x = x`).

Constraint Set Simplification. Symbolic execution typically involves the addition of a large number of constraints to the path condition. The natural structure of programs means that constraints on same variables tend to become more specific. For example, commonly an inexact constraint such as $x < 10$ gets added, followed some time later by the constraint $x = 5$. KLEE actively simplifies the constraint set by rewriting previous constraints when new equality constraints are added to the constraint set. In this example, substituting the value for x into the first constraint simplifies it to `true`, which KLEE eliminates.

Implied Value Concretization. When a constraint such as $x + 1 = 10$ is added to the path condition, then the value of `x` has effectively become concrete along that path. KLEE determines this fact (in this case that $x = 9$) and writes the concrete value back to memory. This ensures that subsequent accesses of that memory location can return a cheap constant expression.

Constraint Independence. Many constraints do not overlap in terms of the memory they reference. Constraint independence (taken from EXE) divides constraint sets into disjoint independent subsets based on the symbolic variables they reference. By explicitly tracking these subsets, KLEE can frequently eliminate irrelevant constraints prior to sending a query to the constraint solver. For example, given the constraint set $\{i < j, j < 20, k > 0\}$, a query of whether $i = 20$ just requires the first two constraints.

Counter-example Cache. Redundant queries are frequent, and a simple cache is effective at eliminating a large number of them. However, it is possible to build a more sophisticated cache due to the particular structure of constraint sets. The counter-example cache maps sets of constraints to counter-examples (i.e., variable assignments), along with a special sentinel used when a set of constraints has no solution. This mapping is stored in a custom data structure — derived from the UBTree structure of Hoffmann and Hoehler [28] — which allows efficient searching for cache entries for both subsets and supersets of a constraint set. By storing the cache in this fashion, the counter-example cache gains three additional ways to eliminate queries. In the example below, we assume that the counter-example cache

Optimizations	Queries	Time (s)	STP Time (s)
None	13717	300	281
Independence	13717	166	148
Cex. Cache	8174	177	156
All	699	20	10

Table 1: Performance comparison of KLEE's solver optimizations on COREUTILS. Each tool is run for 5 minutes without optimization, and rerun on the same workload with the given optimizations. The results are averaged across all applications.

currently has entries for $\{i < 10, i = 10\}$ (no solution) and $\{i < 10, j = 8\}$ (satisfiable, with variable assignments $i \rightarrow 5, j \rightarrow 8$).

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. For example, given the cache above, $\{i < 10, i = 10, j = 12\}$ is quickly determined to be unsatisfiable.

2 When a superset of a constraint set has a solution, that solution also satisfies the original constraint set. Dropping constraints from a constraint set does not invalidate a solution to that set. The assignment $i \rightarrow 5, j \rightarrow 8$, for example, satisfies either $i < 10$ or $j = 8$ individually.

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. Because 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. For example, the constraint set $\{i < 10, j = 8, i \neq 3\}$ can still be satisfied by $i \rightarrow 5, j \rightarrow 8$.

To demonstrate the effectiveness of these optimizations, we performed an experiment where COREUTILS applications were run for 5 minutes with both of these optimizations turned off. We then deterministically reran the exact same workload with constraint independence and the counter-example cache enabled separately and together for the same number of instructions. This experiment was done on a large sample of COREUTILS utilities. The results in Table 1 show the averaged results.

As expected, the independence optimization by itself does not eliminate any queries, but the simplifications it performs reduce the overall running time by almost half (45%). The counter-example cache reduces both the running time and the number of STP queries by 40%. However, the real win comes when both optimizations are enabled; in this case the hit rate for the counter-example cache greatly increases due to the queries first being simplified via independence. For the sample runs, the average number of STP queries are reduced to 5% of the original number and the average runtime decreases by more than an order of magnitude.

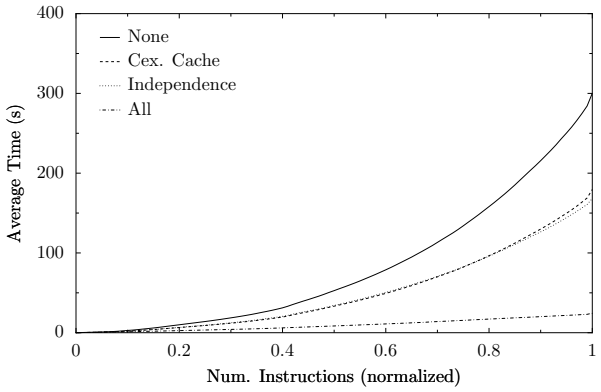

Figure 2: The effect of KLEE's solver optimizations over time, showing they become more effective over time, as the caches fill and queries become more complicated. The number of executed instructions is normalized so that data can be aggregated across all applications.

It is also worth noting the degree to which STP time (time spent solving queries) dominates runtime. For the original runs, STP accounts for 92% of overall execution time on average (the combined optimizations reduce this by almost 300%). With both optimizations enabled this percentage drops to 41%. Finally, Figure 2 shows the efficacy of KLEE's optimizations increases with time — as the counter-example cache is filled and query sizes increase, the speed-up from the optimizations also increases.

3.4 State scheduling

KLEE selects the state to run at each instruction by interleaving the following two search heuristics.

Random Path Selection maintains a binary tree recording the program path followed for all active states, i.e. the leaves of the tree are the current states and the internal nodes are places where execution forked. States 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 states in each subtree has equal probability of being selected, regardless of the size of their subtrees. This strategy has two important properties. First, it favors states high in the branch tree. These states have less constraints on their symbolic inputs and so 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 ("fork bombing") as it happens when a tight loop contains a symbolic condition. Note that the simply selecting a state at random has neither property.

Coverage-Optimized Search tries to select states likely 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. Currently these heuristics take into account the minimum distance to an uncovered instruction, the call stack of the state, and whether the state recently covered new code.

KLEE uses each strategy in a round robin fashion. While this can increase the time for a particularly effective strategy to achieve high coverage, it protects against cases where an individual strategy gets stuck. Furthermore, since strategies pick from the same state pool, interleaving them can improve overall effectiveness.

The time to execute an individual instruction can vary widely between simple instructions (e.g., addition) and instructions which may use the constraint solver or fork execution (branches, memory accesses). KLEE ensures that a state which frequently executes expensive instructions will not dominate execution time by running each state for a "time slice" defined by both a maximum number of instructions and a maximum amount of time.

4 Environment Modeling

When code reads values from its environment — command-line arguments, environment variables, file data and metadata, network packets, etc — we conceptually want to return all values that the read could legally produce, rather than just a single concrete value. When it writes to its environment, the effects of these alterations should be reflected in subsequent reads. The combination of these features allows the checked program to explore all potential actions and still have no false positives.

Mechanically, we handle the environment by redirecting calls that access it to *models* that understand the semantics of the desired action well enough to generate the required constraints. Crucially, these models are written in normal C code which the user can readily customize, extend, or even replace without having to understand the internals of KLEE. We have about 2,500 lines of code to define simple models for roughly 40 system calls (e.g., `open`, `read`, `write`, `stat`, `lseek`, `ftruncate`, `ioctl`).

4.1 Example: modeling the file system

For each file system operation we check if the action is for an actual concrete file on disk or a symbolic file. For concrete files, we simply invoke the corresponding system call in the running operating system. For symbolic files we emulate the operation's effect on a simple symbolic file system, private to each state.

Figure 3 gives a rough sketch of the model for `read()`, eliding details for dealing with linking, reads on standard input, and failures. The code maintains a set of file descriptors, created at file `open()`, and records

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

for each whether the associated file is symbolic or concrete. If `fd` refers to a concrete file, we use the operating system to read its contents by calling `pread()` (lines 7-11). We use `pread` to multiplex access from KLEE's many states onto the one actual underlying file descriptor.[5] If `fd` refers to a symbolic file, `read()` copies from the underlying symbolic buffer holding the file contents into the user supplied buffer (lines 13-19). This ensures that multiple `read()` calls that access the same file use consistent symbolic values.

Our symbolic file system is crude, containing only a single directory with N symbolic files in it. KLEE users specify both the number N and the size of these files. This symbolic file system coexists with the real file system, so applications can use both symbolic and concrete files. When the program calls `open` with a concrete name, we (attempt to) open the actual file. Thus, the call:

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

sets `fd` to point to the actual configuration file `/etc/fstab`.

On the other hand, calling `open()` with an unconstrained symbolic name matches each of the N symbolic files in turn, and will also fail once. For example, given $N = 1$, calling `open()` with a symbolic command-line argument `argv[1]`:

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

will result in 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 an error.

[5] Since KLEE's states execute within a single Unix process (the one used to run KLEE), then unless we duplicated file descriptors for each (which seemed expensive), a `read` by one would affect all the others.

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 — writing models for each would be 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 [6]) and let it worry about correctness. As a side-effect, we simultaneously check the library for errors as well.

4.2 Failing system calls

The real environment can fail in unexpected ways (e.g., `write()` fails because of a full disk). Such failures can often lead to unexpected and hard to diagnose bugs. Even when applications do try to handle them, this code is rarely fully exercised by the regression suite. To help catch such errors, KLEE will optionally simulate environmental failures by failing system calls in a controlled manner (similar to [38]). We made this mode optional since not all applications care about failures — a simple application may ignore disk crashes, while a mail server expends a lot of code to handle them.

4.3 Rerunning test cases

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 (files, pipes, ttys, directories, links, etc.) containing the concrete values used in the test case. It then executes the unmodified program using the concrete command-line arguments from the test case. Our biggest challenge was making system calls fail outside of KLEE — we built a simple utility that uses the `ptrace` debugging interface to skip the system calls that were supposed to fail and instead return an error.

5 Evaluation

This section describes our in-depth coverage experiments for COREUTILS (§ 5.2) and BUSYBOX (§ 5.3) as well as errors found during quick bug-finding runs (§ 5.4). We use KLEE to find deep correctness errors by crosschecking purportedly equivalent tool implementations (§ 5.5) and close with results for HISTAR (§5.6).

5.1 Coverage methodology

We use line coverage as a conservative measure of KLEE-produced test case effectiveness. We chose executable line coverage as reported by `gcov`, because it is widely-understood and uncontroversial. Of course, it grossly underestimates KLEE's thoroughness, since it ignores the fact that KLEE explores many different unique paths with all possible values. We expect a path-based metric would show even more dramatic wins.

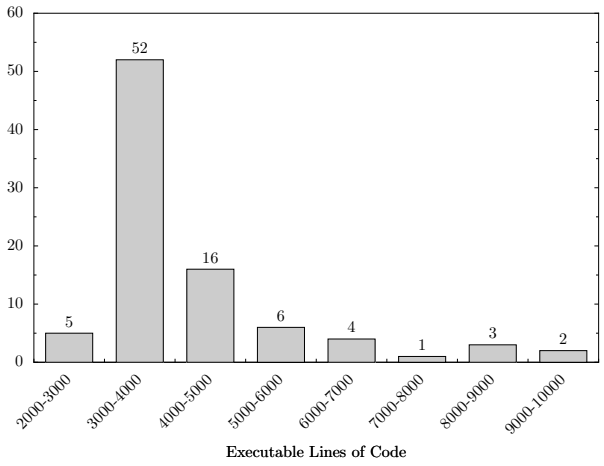

Figure 4: Histogram showing the number of COREUTILS tools that have a given number of executable lines of code (ELOC).

We measure coverage by running KLEE-generated test cases on a stand-alone version of each utility and using `gcov` to measure coverage. Running tests independently of KLEE eliminates the effect of bugs in KLEE and verifies that the produced test case runs the code it claims.

Note, our coverage results only consider code in the tool itself. They do not count library code since doing so makes the results harder to interpret:

1. It double-counts many lines, since often the same library function is called by many applications.
2. It unfairly under-counts coverage. Often, the bulk of a library function called by an application is effectively dead code since the library code is general but 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, we do include library code when measuring the raw size of the application: KLEE must successfully handle this library code (and gets no credit for doing so) in order to exercise the code in the tool itself. We measure size in terms of executable lines of code (ELOC) by counting the total number of executable lines in the final executable after global optimization, which eliminates uncalled functions and other dead code. This measure is usually a factor of three smaller than a simple line count (using `wc -l`).

In our experiments KLEE minimizes the test cases it

Coverage (w/o lib)	COREUTILS		BUSYBOX	
	KLEE tests	Devel. tests	KLEE tests	Devel. tests
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

generates by only emitting tests cases for paths that hit a new statement or branch in the main utility code. A user that wants high library coverage can change this setting.

5.2 GNU COREUTILS

We now give KLEE coverage results for all 89 GNU COREUTILS utilities.

Figure 4 breaks down the tools by executable lines of code (ELOC), including library code the tool calls. While relatively small, the tools are not toys — the smallest five have between 2K and 3K ELOC, over half (52) have between 3K and 4K, and ten have over 6K.

Previous work, ours included, has evaluated constraint-based execution on a small number of hand-selected benchmarks. Reporting results for the entire COREUTILS suite, the worst along with the best, prevents us from hand-picking results or unintentionally cheating through the use of fragile optimizations.

Almost all tools were tested using the same command (command arguments explained in § 2.1):

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

As specified by the `--max-time` option, we ran each tool for about 60 minutes (some finished before this limit, a few up to three minutes after). For eight tools where the coverage results of these values were unsatisfactory, we consulted the `man` page and increased the number and size of arguments and files. We found this easy to do, so presumably a tool implementer or user would as well. After these runs completed, we improved them by failing system calls (see § 4.2).

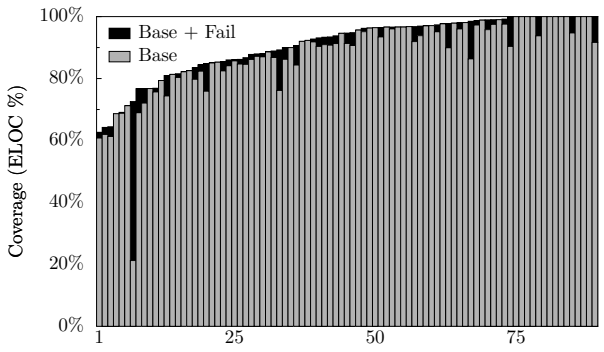

Figure 5: Line coverage for each application with and without failing system calls.

5.2.1 Line coverage results

The first two columns in Table 2 give aggregate line coverage results. On average our tests cover 90.9% of the lines in each tool (median: 94.7%), with an overall (aggregate) coverage across all tools of 84.5%. We get 100% line coverage on 16 tools, over 90% on 56 tools, and over 80% on 77 tools (86.5% of all tools). The minimum coverage achieved on any tool is 62.6%.

We believe such high coverage on a broad swath of applications "out of the box" convincingly shows the power of the approach, especially since it is across the entire tool suite rather than focusing on a few particular applications.

Importantly, KLEE generates high coverage with few test cases: for our non-failing runs, it needs a total of 3,321 tests, with a per-tool average of 37 (median: 33). The maximum number needed was 129 (for the "`[`" tool) and six needed 5. As a crude measure of path complexity, we counted the number of static branches run by each test case using `gcov`[6] (i.e., an executed branch counts once no matter how many times the branch ran dynamically). The average path length was 76 (median: 53), the maximum was 512 and (to pick a random number) 160 were at least 250 branches long.

Figure 5 shows the coverage KLEE achieved on each tool, with and without failing system call invocations. Hitting system call failure paths is useful for getting the last few lines of high-coverage tools, rather than significantly improving the overall results (which it improves from 79.9% to 84.5%). The one exception is `pwd` which requires system call failures to go from a dismal 21.2% to 72.6%. The second best improvement for a single tool is a more modest 13.1% extra coverage on the `df` tool.

[6] In `gcov` terminology, a branch is a possible branch direction, i.e. a simple if statement has two branches.

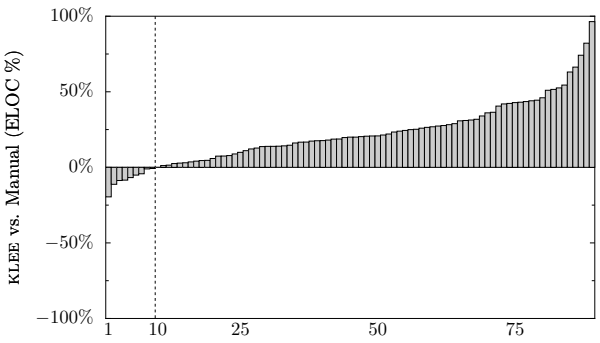

Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 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
ptx x t4.txt
seq -f %0 1
```

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

5.2.2 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. [7] As Table 2 shows, KLEE beats developer tests handily on all aggregate measures: overall total line coverage (84.5% versus 67.7%), average coverage per tool (90.9% versus 68.4%) and median coverage per tool (94.7% versus 72.5%). At a more detailed level, KLEE gets 100% coverage on 16 tools and over 90% coverage on 56 while the developer tests get 100% on a single utility (`true`) and reach over 90% on only 7. Finally, the developers tests get below 60% coverage on 24 tools while KLEE always achieves over 60%. In total, an 89 hour run of KLEE (about one hour per application) exceeds the coverage of a test suite built over a period of fifteen years by 16.8%!

Figure 6 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 shows the opposite. KLEE beats manual testing, often significantly, on the vast majority of the applications.

To guard against hidden bias in line coverage, we also compared the taken branch coverage (as reported by `gcov`) of the manual and KLEE test suites. While the absolute coverage for both test suites decreases, KLEE's relative improvement over the developers' tests remains: KLEE achieves 76.9% overall branch coverage, while the developers' tests get only 56.5%.

Finally, it is important to note that although KLEE's runs significantly beat the developers' tests in terms of coverage, KLEE only checks for low-level errors and violations of user-level asserts. In contrast, developer tests typically validate that the application output matches the expected one. We partially address this limitation by validating the output of these utilities against the output produces by a different implementation (see § 5.5).

5.2.3 Bugs found

KLEE found ten unique bugs in COREUTILS (usually memory error crashes). Figure 7 gives the command lines used to trigger them. The first three errors existed since at least 1992, so should theoretically crash any COREUTILS distribution up to 6.10. The others are more recent, and do not crash older COREUTILS distributions. While one bug (in `seq`) had been fixed in the developers' unreleased version, the other bugs were confirmed and fixed within two days of our report. In addition, versions of the KLEE-generated test cases for the new bugs were added to the official COREUTILS test suite.

As an illustrative example, we discuss the bug in `pr` (used to paginate files before printing) hit by the invocation "`pr -e t2.txt`" in Figure 7. The code containing the bug is shown in Figure 8. On the path that hits the bug, both `chars_per_input_tab` and `chars_per_c` equal tab width (let's call it T). Line 2665 computes `width` $= (T -$ `input_position` $\bmod T)$ using the macro on line 602. The root cause of the bug is the incorrect assumption that $0 \leq x \mod y < y$, which only holds for positive integers. When `input_position` is positive, `width` will be less than T since $0 \leq$ `input_position` $\bmod T < T$. However, in the pres-

[7] We ran the test suite using the commands: `env RUN_EXPENSIVE_TESTS=YES RUN_VERY_EXPENSIVE_TESTS=YES make check` and `make check-root` (as root). A small number of tests (14 out of 393) which require special configuration were not run; from manual inspection we do not expect these to have a significant impact on our results.

```
 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 8: 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`.

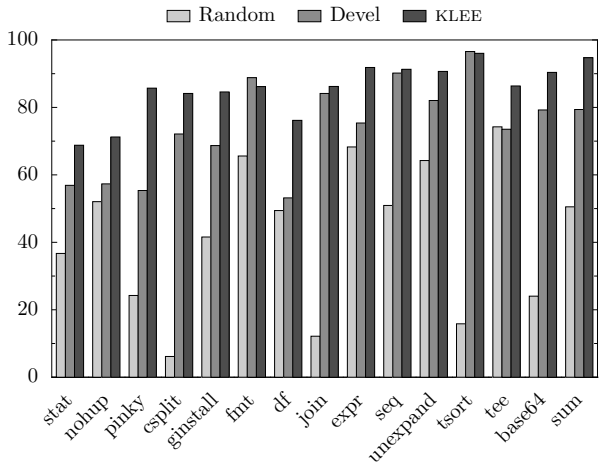

Figure 9: Coverage of random vs. manual vs. KLEE testing for 15 randomly-chosen COREUTILS utilities. Manual testing beats random on average, while KLEE beats both by a significant margin.

ence of backspaces, `input_position` can become negative, so ($-T <$ `input_position` $\bmod T < T$). Consequently, `width` can be as large as $2T - 1$.

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 $2T - 1$, a memory overflow is possible.

This is a prime example of the power of symbolic execution in finding complex errors in code which is hard to reason about manually — this bug has existed in `pr` since at least 1992, when COREUTILS was first added to a CVS repository.

5.2.4 Comparison with random tests

In our opinion, the COREUTILS manual tests are unusually comprehensive. However, we compare to random testing both to guard against deficiencies, and to get a feel for how constraint-based reasoning compares to blind random guessing. We tried to make the comparison apples-to-apples by building a tool that takes the same command line as KLEE, and generates random values for the specified type, number, and size range of inputs. It then runs the checked program on these values using the same replay infrastructure as KLEE. For time reasons, we randomly chose 15 benchmarks (shown in Figure 9) and ran them for 65 minutes (to always exceed the time given to KLEE) with the same command lines used when run with KLEE.

Figure 9 shows the coverage for these programs achieved by random, manual, and KLEE tests. Unsurprisingly, given the complexity of COREUTILS programs and the concerted effort of the COREUTILS maintainers, the manual tests get significantly more coverage than random. KLEE handily beats both.

Because `gcov` introduces some overhead, we also performed a second experiment in which we ran each tool natively without `gcov` for 65 minutes (using the same random seed as the first run), recorded the number of test cases generated, and then reran using `gcov` for that number. This run completely eliminates the `gcov` overhead, and overall it generates 44% more tests than during the initial run.

However, these 44% extra tests increase the average coverage per tool by only 1%, with 11 out of 15 utilities not seeing any improvement — showing that random gets stuck for most applications. We have seen this pattern repeatedly in previous work: random quickly gets the cases it can, and then revisits them over and over. Intuitively, satisfying even a single 32-bit equality requires correctly guessing one value out of four billion. Correctly getting a sequence of such conditionals is hopeless. Utilities such as `csplit` (the worst performer), illustrate this dynamic well: their input has structure, and the difficulty of blindly guessing values that satisfy its rules causes most inputs to be rejected.

One unexpected result was that for 11 of these 15 programs, KLEE explores paths to termination (i.e., the checked code calls `exit()`) only a few times slower than random does! KLEE explored paths to termination in roughly the same time for three programs and, in fact, was actually faster for three others (`seq`, `tee`, and `nohup`). We were surprised by these numbers, because we had assumed a constraint-based tool would run orders of magnitude more slowly than raw testing on a per-path basis, but would have the advantage of exploring more unique paths over time (with all values) because it did not get stuck. While the overhead on four programs matched this expectation (where constraint solver overhead made paths ran 7x to 220x more slowly than

native execution), the performance tradeoff for the others is more nuanced. Assume we have a branch deep in the program. Covering both true and false directions using traditional testing requires running the program from start to finish twice: once for the true path and again for the false. In contrast, while KLEE runs each instruction more slowly than native execution, it only needs to run the instruction path before the branch once, since it forks execution at the branch point (a fast operation given its object-level copy-on-write implementation). As path length grows, this ability to avoid redundantly rerunning path prefixes gets increasingly important.

With that said, the reader should view the per-path costs of random and KLEE as very crude estimates. First, the KLEE infrastructure random uses to run tests adds about 13ms of per-test overhead, as compared to around 1ms for simply invoking a program from a script. This code runs each test case in a sandbox directory, makes a clean environment, and creates various system objects with random contents (e.g., files, pipes, tty's). It then runs the tested program with a watchdog to terminate infinite loops. While a dedicated testing tool must do roughly similar actions, presumably it could shave some milliseconds. However, this fixed cost matters only for short program runs, such as when the code exits with an error. In cases where random can actually make progress and explore deeper program paths, the inefficiency of rerunning path prefixes starts to dominate. Further, we conservatively compute the path completion rate for KLEE: when its time expires, roughly 30% of the states it has created are still alive, and we give it no credit for the work it did on them.

5.3 BUSYBOX utilities

BUSYBOX is a widely-used implementation of standard UNIX utilities for embedded systems that aims for small executable sizes [1]. Where there is overlap, it aims to replicate COREUTILS functionality, although often providing fewer features. We ran our experiments on a bug-patched version of BUSYBOX 1.10.2. We ran the 75 utilities [8] in the BUSYBOX "`coreutils`" subdirectory (14K lines of code, with another 16K of library code), using the same command lines as when checking COREUTILS, except we did not fail system calls.

As Table 2 shows, KLEE does even better than on COREUTILS: over 90.5% total line coverage, on average covering 93.5% per tool with a median of 97.5%. It got 100% coverage on 31 and over 90% on 55 utilities.

BUSYBOX has a less comprehensive manual test suite than COREUTILS (in fact, many applications don't seem to have any tests). Thus, KLEE beats the developers tests by roughly a factor of two: 90.5% total line coverage versus only 44.8% for the developers' suite. The developers do better on only one benchmark, `cp`.

[8] We are actually measuring coverage on 72 files because several utilities are implemented in the same file.

`date -I`
`ls --co`
`chown a.a -`
`kill -l a`
`setuidgid a ""`
`printf "% *" B`
`od t1.txt`
`od t2.txt`
`printf %`
`printf %Lo`
`tr [`
`tr [=`
`tr [a-z`

t1.txt: `a`
t2.txt: `A`
t3.txt: `\t\n`

`cut -f t3.txt`
`install --m`
`nmeter -`
`envdir`
`setuidgid`
`envuidgid`
`envdir -`
`arp -Ainet`
`tar tf_ /`
`top d`
`setarch "" ""`
`<full-path>/linux32`
`<full-path>/linux64`
`hexdump -e ""`
`ping6 -`

Figure 10: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in BUSYBOX. When multiple applications crash because of the same shared (buggy) piece of code, we group them by shading.

5.4 Bug-finding: MINIX + all BUSYBOX tools

To demonstrate KLEE's applicability to bug finding, we used KLEE to check all 279 BUSYBOX tools and 84 MINIX tools [4] in a series of short runs. These 360+ applications cover a wide range of functionality, such as networking tools, text editors, login utilities, archiving tools, etc. While the tests generated by KLEE during these runs are not sufficient to achieve high coverage (due to incomplete modeling), we did find many bugs quickly: 21 bugs in BUSYBOX and another 21 in MINIX have been reported (many additional reports await inspection). Figure 10 gives the command lines for the BUSYBOX bugs. All bugs were memory errors and were fixed promptly, with the exception of `date` which had been fixed in an unreleased tree. We have not heard back from the MINIX developers.

5.5 Checking tool equivalence

Thus far, we have focused on finding generic errors that do not require knowledge of a program's intended behavior. We now show how to do much deeper checking, including verifying full functional correctness on a finite set of explored paths.

KLEE makes no approximations: its constraints have perfect accuracy down to the level of a single bit. If KLEE reaches an `assert` and its constraint solver states the false branch of the `assert` cannot execute given the current path constraints, then it has *proved* that no value exists on *the current path* that could violate the assertion,

```
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: }
```

Figure 11: Trivial program illustrating equivalence checking. KLEE proves total equivalence when $y \neq 0$.

modulo bugs in KLEE or non-determinism in the code. [9] Importantly, KLEE will do such proofs for any condition the programmer expresses as C code, from a simple non-null pointer check, to one verifying the correctness of a program's output.

This property can be leveraged to perform deeper checking as follows. Assume we have two procedures `f` and `f'` that take a single argument and purport to implement the same interface. We can verify functional equivalence on a per-path basis by simply feeding them the same symbolic argument and asserting they return the same value: `assert(f(x) == f'(x))`. Each time KLEE follows a path that reaches this assertion, it checks if any value exists on that path that violates it. If it finds none exists, then it has proven functional equivalence on that path. By implication, if one function is correct along the path, then equivalence proves the other one is as well. Conversely, if the functions compute different values along the path and the `assert` fires, then KLEE will produce a test case demonstrating this difference. 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.

Note that equivalence results only hold on the finite set of paths that KLEE explores. Like traditional testing, it cannot make statements about paths it misses. However, if KLEE is able to exhaust all paths then it has shown total equivalence of the functions. Although not tractable in general, many isolated algorithms can be tested this way, at least up to some input size.

We help make these points concrete using the contrived example in Figure 11, which crosschecks a trivial modulo implementation (`mod`) against one that optimizes for modulo by powers of two (`mod_opt`). It first makes the inputs `x` and `y` symbolic and then uses the `assert` (line 14) to check for differences. Two code paths reach this `assert`, depending on whether the test for power-of-two (line 2) succeeds or fails. (Along the way, KLEE generates a division-by-zero test case for when $y = 0$.) The true path uses the solver to check that the constraint $(y\& - y) == y$ implies $(x\&(y-1)) == x\%y$ holds for all values. This query succeeds. The false path checks the vacuous tautology that the constraint $(y\& - y) \neq y$ implies that $x\%y == x\%y$ also holds. The KLEE checking run then terminates, which means that KLEE has proved equivalence for all non-zero values using only a few lines of code.

This methodology is useful in a broad range of contexts. Most standardized interfaces — such as libraries, networking servers, or compilers — have multiple implementations (a partial motivation for and consequence of standardization). In addition, there are other common cases where multiple implementations exist:

1. `f` is a simple reference implementation and `f'` a real-world optimized version.
2. `f'` is a patched version of `f` that purports only to remove bugs (so should have strictly fewer crashes) or refactor code without changing functionality.
3. `f` has an inverse, which means we can change our equivalence check to verify $f^{-1}(f(x)) \equiv x$, such as: `assert(uncompress(compress(x))==x)`.

Experimental results. We show that this technique can find deep correctness errors and scale to real programs by crosschecking 67 COREUTILS tools against their allegedly equivalent BUSYBOX implementations. For example, given the same input, the BUSYBOX and COREUTILS versions of `wc` should output the same number of lines, words and bytes. In fact, both the BUSYBOX and COREUTILS tools intend to conform to IEEE Standard 1003.1 [3] which specifies their behavior.

We built a simple infrastructure to make crosschecking automatic. Given two tools, it renames all their global symbols and then links them together. It then runs both with the same symbolic environment (same symbolic arguments, files, etc.) and compares the data printed to `stdout`. When it detects a mismatch, it generates a test case that can be run to natively to confirm the difference.

Table 3 shows a subset of the mismatches found by KLEE. The first three lines show hard correctness errors (which were promptly fixed by developers), while the others mostly reveal missing functionality. As an example of a serious correctness bug, the first line gives the inputs that when run on BUSYBOX's `comm` causes it to behave as if two non-identical files were identical.

[9] Code that depends on the values of memory addresses will not satisfy determinism since KLEE will almost certainly allocate memory objects at different addresses than native runs.

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[infinite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/: Is a directory"`
`split /`	`"/: Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[ 0 ''<'' 1 ]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

Table 3: Very small subset of the mismatches KLEE found between the BUSYBOX and COREUTILS versions of equivalent utilities. The first three are serious correctness errors; most of the others are revealing missing functionality.

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 disk. For comparison we did the same runs using random inputs for one million trials.

5.6 The HiStar OS kernel

We have also applied KLEE to checking non-application code by using it to check the HiStar [39] kernel. We used a simple test driver based on a user-mode HISTAR kernel. The driver creates the core kernel data structures and initializes a single process with access to a single page of user memory. It then calls the test function in Figure 12, which makes the user memory symbolic and executes a predefined number of system calls using entirely symbolic arguments. As the system call number is encoded in the first argument, this simple driver effectively tests all (sequences of) system calls in the kernel.

Although the setup is restrictive, in practice we have found that it can quickly generate test cases — sequences of system call vectors and memory contents — which cover a large portion of the kernel code 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. As a quick comparison, we ran one million random tests through the same driver (similar to § 5.2.4). As Table 4 shows, KLEE's tests achieve significantly more coverage than random testing both for runs with (+17.0%) and without (+28.4%) a disk.

```
1 : static void test(void *upage, unsigned num_calls) {
2 :   make_symbolic(upage, PGSIZE);
3 :   for (int i=0; i<num_calls; i++) {
4 :     uint64_t args[8];
5 :     for (int j=0; j<8; j++)
6 :       make_symbolic(&args[j], sizeof(args[j]));
7 :     kern_syscall(args[0], args[1], args[2], args[3],
8 :                  args[4], args[5], args[6], args[7]);
9 :   }
10:   sys_self_halt();
11: }
```

Figure 12: Test driver for HISTAR: it makes a single page of user memory symbolic and executes a user-specified number of system calls with entirely symbolic arguments.

KLEE's constraint-based reasoning allowed it to find a tricky, critical security bug in the 32-bit version of HISTAR. Figure 13 shows the code for the function containing the bug. The function `safe_addptr` is supposed to set `*of` to true if the addition overflows. However, because the inputs are 64 bit long, 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 validates 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.

6 Related Work

Many recent tools are based on symbolic execution [11, 14–16, 20–22, 24, 26, 27, 36]. We contrast how KLEE deals with the environment and path explosion problems.

To the best of our knowledge, traditional symbolic ex-

```
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 : }
```

Figure 13: 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`.

ecution systems [17, 18, 32] are static in a strict sense and do not interact with the 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 [16, 26, 36] does allow external interactions, but forces them to use entirely concrete procedure call arguments, which limits the behaviors they can explore: a concrete external call will do exactly what it did, rather than all things it could potentially do. In KLEE, we strive for a functional balance between these two alternatives; we allow both interaction with the outside environment and supply a model to simulate interaction with a symbolic one.

The path explosion problem has instead received more attention [11, 22, 24, 27, 34]. Similarly to the search heuristics presented in Section 3, search strategies proposed in the past include Best First Search [16], Generational Search [27], and Hybrid Concolic Testing [34]. Orthogonal to search heuristics, researchers have addressed the path explosion problem by testing paths compositionally [8, 24], and by tracking the values read and written by the program [11].

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 [36], and the *constraint subsumption* optimization discussed in [27].

Similar to symbolic execution systems, model checkers have been used to find bugs in both the design and the implementation of software [10, 12, 19, 25, 29, 30]. These approaches often require a lot of manual effort to build test harnesses. However, the approaches are somewhat complementary to KLEE: the tests KLEE generates can be used to drive the model checked code, similar to the approach embraced by Java PathFinder [31, 37].

Previously, we showed that symbolic execution can find correctness errors by crosschecking various implementations of the same library function [15]; this paper shows that the technique scales to real programs. Subsequent to our initial work, others applied similar ideas to finding correctness errors in applications such as network protocol implementations [13] and PHP scripts [9].

7 Conclusion

Our long-term goal is to take an arbitrary program and routinely get 90%+ code coverage, crushing it under test cases for all interesting inputs. While there is still a long way to go to reach this goal, our results show that the approach works well across a broad range of real code. Our system KLEE, automatically generated tests that, on average, covered over 90% of the lines (in aggregate over 80%) in roughly 160 complex, system-intensive applications "out of the box." This coverage significantly exceeded that of their corresponding hand-written test suites, including one built over a period of 15 years.

In total, we used KLEE to check 452 applications (with over 430K lines of code), where it found 56 serious bugs, including ten in COREUTILS, arguably the most heavily-tested collection of open-source applications. To the best of our knowledge, this represents an order of magnitude more code and distinct programs than checked by prior symbolic test generation work. Further, because KLEE's constraints have no approximations, its reasoning allow it to prove properties of paths (or find counter-examples without false positives). We used this ability both to prove path equivalence across many real, purportedly identical applications, and to find functional correctness errors in them.

The techniques we describe should work well with other tools and provide similar help in handling a broad class of applications.

8 Acknowledgements

We thank the GNU COREUTILS developers, particularly the COREUTILS maintainer Jim Meyering for promptly confirming our reported bugs and answering many questions. We similarly thank the developers of BUSYBOX, particularly the BUSYBOX maintainer, Denys Vlasenko. We also thank Nickolai Zeldovich, the designer of HISTAR, for his great help in checking HISTAR, including writing a user-level driver for us. We thank our shepard Terence Kelly, the helpful OSDI reviewers, and Philip Guo for valuable comments on the text. This research was supported by DHS grant FA8750-05-2-0142, NSF TRUST grant CCF-0424422, and NSF CAREER award CNS-0238570-001. A Junglee Graduate Fellowship partially supported Cristian Cadar.

References

[1] Busybox. `www.busybox.net`, August 2008.

[2] Coreutils. `www.gnu.org/software/coreutils`, August 2008.

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

[4] MINIX 3. `www.minix3.org`, August 2008.

[5] SecurityFocus, `www.securityfocus.com`, March 2008.

[6] uCLibc. `www.uclibc.org`, May 2008.

[7] United States National Vulnerability Database, `nvd.nist.gov`, March 2008.

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

[9] ARTZI, S., KIEŻUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. D. Finding bugs in dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2008)*.

[10] BALL, T., AND RAJAMANI, S. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN 2001)*.

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

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

[13] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of USENIX Security Symposium (USENIX Security 2007)*.

[14] 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 (IEEE S&P 2006)*.

[15] 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 (SPIN 2005)*.

[16] 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 (CCS 2006)*.

[17] CLARKE, E., AND KROENING, D. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2003)*.

[18] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*.

[19] CORBETT, J., DWYER, M., HATCLIFF, J., LAUBACH, S., PASAREANU, C., ROBBY, AND ZHENG, H. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering (ICSE 2000)*.

[20] 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 2007)*.

[21] 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 2005)*.

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

[23] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*.

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

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

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

[27] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security (NDSS 2008)*.

[28] HOFFMANN, J., AND KOEHLER, J. A new method to index and query sets. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 1999)*.

[29] HOLZMANN, G. J. From code to models. In *Proceedings of 2nd International Conference on Applications of Concurrency to System Design (ACSD 2001)*.

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

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

[32] KROENING, D., CLARKE, E., AND YORAV, K. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC 2003)*.

[33] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO 2004)*.

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

[35] 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.

[36] 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 Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*.

[37] VISSER, W., PASAREANU, C. S., AND KHURSHID, S. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*.

[38] YANG, J., SAR, C., AND ENGLER, D. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*.

[39] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*.