Practical, low-effort equivalence verification of real code

David A. Ramos and Dawson R. Engler

Stanford University
ramos@cs.stanford.edu, engler@csl.stanford.edu

Abstract. Verifying code equivalence is useful in many situations, such as checking: yesterday's code against today's, different implementations of the same (standardized) interface, or an optimized routine against a reference implementation. We present a tool designed to easily check the equivalence of two arbitrary C functions. The tool provides guarantees far beyond those possible with testing, yet it often requires less work than writing even a single test case. It automatically synthesizes inputs to the routines and uses bit-accurate, sound symbolic execution to verify that they produce equivalent outputs on a finite number of paths, even for rich, nested data structures. We show that the approach works well, even on heavily-tested code, where it finds interesting errors and gets high statement coverage, often exhausting all feasible paths for a given input size. We also show how the simple trick of checking equivalence of *identical* code turns the verification tool chain against itself, finding errors in the underlying compiler and verification tool.

1 Introduction

Historically, code verification has been hard. Thus, implementors rarely make any effort to do it. We present UC-KLEE, a modified version of the KLEE [2] tool designed to make it easy to verify that two routines are equivalent. This ability is useful in many situations, such as checking: different implementations of the same (standardized) interface, different versions of the same implementation, optimized routines against a reference implementation, and finding compiler bugs by comparing code compiled with and without optimization. Comparing identical code against itself finds bugs in our own tool.

Previously, cross checking code that takes inputs with complex invariants or complicated data structures required tediously constructing these inputs by hand. From experience, the non-trivial amount of code needed to do so can easily dwarf the size of the checked code (e.g., as happens when checking small library routines). Manual construction also leads to missed errors caused by over-specificity. For example, when manually building a linked list containing symbolic data, should it have one entry? Two? A hash table should have how many collisions and in which buckets? Creating all possible instances is usually difficult or even impossible. Further, manually specifying pointers (by assigning the concrete address returned by `malloc`) can limit paths that check relationships

on them, such as when an if-statement checks whether one pointer is less than another. In general, if input has many constraints, a human tester will miss one.

In contrast, using our tool is easy: rather than requiring users to manually construct inputs or write a specification to check code against, they simply give our tool two routines (written in raw, unannotated C) to cross check. The tool automatically synthesizes the routines' inputs (even for rich, nested data structures) and systematically explores a finite number of their paths using sound, bit-accurate symbolic execution. It verifies that the routines produce identical results when fed identical inputs on these explored paths by checking that they either (1) write the same values to all escaping memory locations or (2) terminate with the same errors. If one path is correct, then verifying equivalence proves the other is as well. If the tool terminates, then with some caveats (discussed in § 3.4), it has verified equivalence up to a given input size.

Because UC-KLEE leverages the underlying KLEE system to automatically explore paths and reason about all values feasible on each path, it gives guarantees far beyond those of traditional testing, yet it often requires less work than writing even a single test case. We show that the approach works well even on heavily-tested code, by using it to cross check hundreds of routines in two mature, widely-used open source `libc` implementations, where it:

1. Found numerous interesting errors.
2. Verified the equivalence of 300 routines (150 distinct pairs) by exhausting all their paths up to a fixed input size (8 bytes).
3. Got high statement coverage — the lowest median coverage for any experiment was 90% and the rest were 100%.

A final contribution is a simple, novel trick for finding bugs in the compiler and checking tool by turning the technique on itself, which we used to detect a serious LLVM optimizer bug and numerous errors in UC-KLEE.

2 Overview

Cross checking implementations simplifies finding correctness violations because, rather than requiring that users write a functional specification, it lets the tool use a second implementation as a reference — functional differences will show up as mismatches. A natural concern is what happens on invalid inputs. In our experience, real code often shows *error equivalence*, where an illegal input causes the same behavior in both (e.g., when given a null pointer, both cross checked routines crash). Our tool exploits this fact and treats equivalent crashes as equivalent behavior, but flags when one implementation crashes and the other does not. (In general, cross checking cannot detect when two routines make equivalent mistakes.) This finesse works well in practice. In the rare cases where inputs are allowed to produce differing results, it is easy for simple, user-written C code to filter these inputs (discussed further in § 2.1).

We show how UC-KLEE works by walking through the simple but complete example in Figure 1, which gives two trivial routines intended to add a value

to a structure field and the cross checking harness that UC-KLEE provides to compare them. The user compiles the routines using UC-KLEE's compiler (LLVM) and gives the resultant bitcode modules and two routine names to UC-KLEE, which links the code against a checking harness and runs the result. At a high level, the cross checking harness executes as follows:

1. It marks all function parameters as containing unconstrained symbolic data (i.e., they can contain any value representable given their size). UC-KLEE will lazily allocate memory on demand if this symbolic data is used as a pointer and dereferenced (discussed below).
2. It uses symbolic execution to explore (ideally all) paths in the two implementations, checking that they produce identical effects when run on the same values.
3. If a path's constraints permit a value that causes an error (such as a division by zero, null pointer dereference, or assertion failure), UC-KLEE verifies that the other routine terminates with the same error when run with the same input values. UC-KLEE also forks execution and explores a path on which the error does not occur so that it can cross-check the routine on the remaining values.
4. At the end of each path, UC-KLEE traverses all reachable memory, and uses its constraint solver to prove this memory has equivalent contents at the end of both paths. If this check fails, it generates a concrete input to demonstrate the difference. If the check succeeds, then with some caveats (see § 3.4) UC-KLEE has verified the two routines as equivalent since the constraints it tracks are accurate and exact (down to the level of a single bit). Thus, if one path is correct, UC-KLEE has verified that the other path is correct as well.

Note that UC-KLEE's equivalence guarantee only holds on the finite set of paths that it explores. Like traditional testing, it cannot make statements about paths it misses. However, in many cases, even if there are too many paths, KLEE can at least show total equivalence up to a given input size.

At a more detailed level, the code in Figure 1 works as follows:

Lines 14–18: stack allocates two variables to pass as the routine's parameters (`f` and `v`) and marks them as symbolic.

Line 21: creates a copy of the current address space, which it will restore later so that `add_bad` runs on identical values.

Line 22: uses `klee_eval` to run `add`. This call returns once for each path explored in `add`. If `add` terminates with an error, the error is stored in `e1`.

Line 3: at the first dereference `f→val`, UC-KLEE checks if `f` can be null. Since `f` has no constraints on its value, it can, so UC-KLEE forks execution and continues as follows:

Error path: adds the constraint that `f` is null, records in `e1` that a null dereference error occurred, and returns from `klee_eval`.

Non-error path: adds the constraint that `f` is not null and attempts to resolve the dereference. It determines that `f` is an unbound symbolic pointer, so it allocates memory (of size `foo`), marks it as symbolic, binds it to `f`, and continues executing until the path completes. It then returns from `klee_eval`.

```
1 : // two routines to cross−check.
2 : int add(foo *f, int v) {
3 :   f−>val = f−>val + v;
4 :   return f−>val;
5 : }
6 : int add_bad(foo *f, int v) {
7 :   f−>val = f−>val + 1;
8 :   return f−>val;
9 : }
10:
11: // harness provided by uc−klee
12: main() {
13:   klee_err e1,e2;
14:   int retv, v;
15:   foo *f;
16:
17:   klee_make_symbolic(&f);
18:   klee_make_symbolic(&v);
19:
20:   // record memory state "add" runs on.
21:   int s0 = klee_snapshot();
22:   klee_eval(retv = add(f,v), &e1);
23:   int s1 = klee_snapshot();
24:
25:   // discard writes, keep path constraints
26:   klee_restore(s0);
27:   klee_eval(retv = add_bad(f,v), &e2);
28:   int s2 = klee_snapshot();
29:
30:   // compare results.
31:   if (!klee_compare_errors(&e1,&e2)
32:       || !klee_compare(s1, s2, &f)
33:       || !klee_compare(s1, s2, &v)
34:       || !klee_compare(s1, s2, &retv))
35:     klee_error("Mismatch!\n");
36: }
```

Fig. 1. Trivial but complete checking example.

Line 22 (after `klee_eval`): the two paths execute independently through the remaining code.

Line 23: records the memory state produced by running `add`, which it later compares against the memory state produced by running `add_bad`.

Line 26: restores the values of `f` and `v` that the current path ran `add` on so that `add_bad` runs on identical values. It discards all writes `add` did (otherwise `add_bad` would run with a modified value for `f→val`), but preserves all constraints, including any pointers it lazily bound (i.e., the dereference of `f` on line 3).

Line 27: evaluates `add_bad` using `klee_eval`. The error path also returns with a null pointer error (since `f` is constrained to null and line 7 dereferences it). The non-error path executes without error; the dereferences of pointer `f` (lines 7,8) resolve to the same object lazily allocated at line 3.

Line 31: checks that both paths returned with the same error state (they did).

Lines 32–34: checks that the values transitively reachable from the routines' outputs in each memory state are equivalent (§ 3.2 describes this analysis in more detail). On the non-error path, the check for `f` (line 32) fails and produces a test case with `v` equal to some value other than 1 (the single value for which both routines return identical results).

Notes. While the example declares the input variables `f` and `v` with their static types for readability, as far as UC-KLEE is concerned they could have been untyped byte arrays (which is how UC-KLEE treats them in any case) since our implementation correctly handles casting between pointers and integers.

Although this example does not access environment variables, UC-KLEE additionally marks the global `environ` pointer as unbound in order to explore paths where its values are read. Section 4 gives an example difference found as a result.

For performance, UC-KLEE does not explore paths that have identical LLVM instructions since they must produce the same results. UC-KLEE silently prunes a path when it satisfies both of the following conditions: (1) all previously executed basic blocks are identical and (2) all reachable basic blocks are identical.

2.1 Handling uninteresting mismatches

A tight specification that maps each input value to a single output value provides the simplest use case for UC-KLEE since *any* difference between implementations constitutes a bug. For looser specifications that include "don't cares," user effort may be needed to suppress uninteresting differences that UC-KLEE would otherwise report. Examples include permitting code to do anything when fed illegal input values or representing "success" by any non-zero integer rather than a single, specific value (e.g., `1`). Note that the problems caused by permitting flexibility are not specific to UC-KLEE — any method (such as randomized or specification-based testing) that checks output values or behavior has to deal with them.

UC-KLEE provides a simple yet general mechanism for eliminating uninteresting mismatches. Instead of invoking checked code directly (lines 22 and 27 in Figure 1), it passes the checked routine and its arguments to a user-supplied function, which calls the checked routine after filtering its input (e.g., by using an if-statement to skip illegal values) and then returns the (possibly canonicalized) return value.

Figure 2 shows an example filter for the `isdigit` function in the C library, specified to return non-zero if its input represents a digit in ASCII and 0 otherwise. The filter first rejects input values that fall outside the range specified in the C standard (line 2). It then invokes the passed-in `isdigit` function (line 4) and canonicalizes all non-zero return values to 1. In our experiments, this routine eliminated all mismatches for `isdigit` and 11 analogous routines.

```
1 : int isdigit_f(int (*f)(int), int c){
2 :   if (c < EOF || c > 255)
3 :     return 0;
4 :   return ((*f)(c) != 0);
5 : }
```

Fig. 2. Simple filter routine.

In practice, even if a specification permits variable behavior, code tends to behave similarly. In fact, the most wide-spread use for UC-KLEE we envision — checking new versions of code against old versions — suffers from this problem the least since such decisions are consistent across revisions. Even where we would expect the most variance in behavior — independently-developed code bases fed error inputs — implementations tend to behave similarly. For example, in our experiments, checked routines typically crashed on illegal pointer inputs rather than returning differing values.

Many of the differences UC-KLEE found illustrated needless ambiguities in the underlying standard, which permitted divergent behavior without a subsequent gain in speed, power, or simplicity. In future work, we plan to explore the use of UC-KLEE as an automatic tool for finding such specification imprecisions.

In a sense, UC-KLEE inverts the typical work factor for checking code: a traditional specification-based approach requires specifying what behavior the user

cares about (i.e., the functionality the code should implement), whereas UC-KLEE infers this information "for free" by cross checking implementations. On the other hand, UC-KLEE (may) require specifying the "don't care" behaviors (when code is allowed to differ), which typically takes orders of magnitude less effort than specifying functionality. Further, users only need to specify these details on demand, after UC-KLEE detects an uninteresting mismatch. In contrast, specification verification requires non-trivial work before doing any checking.

3 Implementation

In this section we discuss the trickiest part of implementing UC-KLEE: tracking whether a piece of memory contains a pointer and if so, to which memory object. We then describe how it uses this ability both to compare the results of two procedure invocations (§ 3.2) and to lazily allocate memory when an unbound pointer is dereferenced (§ 3.3). We then discuss limitations.

3.1 Referents: tracking which memory contains pointers

UC-KLEE tracks pointers using a *referent*-based approach similar to [15, 21], but modified to support symbolic execution. Each register, stack location, global, and heap object has a corresponding piece of shadow memory. Whenever code writes a pointer to memory or a register, UC-KLEE writes the starting address of the pointed-to object (its referent) to this shadow memory at the same offset. When code writes a non-pointer value, UC-KLEE clears these shadow values, indicating that the memory does not contain a pointer. The key advantage this approach is that we can determine what object a pointer was *intended* to point to even if the pointer's *actual* value refers to an address outside the bounds of the object (and potentially inside a different, allocated object).

For space reasons we elide most details of this tracking. It roughly mirrors that of a concrete tool, with the one difference that we must reason about reads and writes at symbolic locations — i.e., those calculated using symbolic expressions and hence representing a set of values rather than a single concrete value (a constant). For example, suppose variable `i` has the constraint $0 <= i < n$ where n is the size of array `a`. Then the write `a[i] = &p` conceptually creates a set of n possible (exclusive) arrays since the write of address `&p` could be to `a[0]` or `a[1]` or ... `a[n]` depending on the value of `i`. Subsequent stores using symbolic indices create more possibilities; reads cause similar explosions. Thus, we cannot just trivially calculate which shadow location to propagate this information to or read it from. Fortunately, the solution is easier to describe than the problem: KLEE already has the ability to reason about the reads and writes checked code performs at symbolic locations; by handling and representing shadow memory in the same way as arrays in checked code, we can reuse this same machinery.

Low-level C code can egregiously violate any sensible notion of typing. To handle such code we took the extreme position of completely ignoring static types and treating all memory as potentially containing a pointer. (We may

revisit this decision in the future.) While there are a variety of details to get right, the most common problem is that given a dereference `*(p+q)`, we do not robustly know which value (`p` or `q`) holds the address, and which the offset. We determine this information based on usage rather than type, and mark any location as not holding a pointer if the code performs operations such as: bitwise operations that lose upper bits (losing the lower few bits is okay), negation, left shift, multiplication, division, modulus, and subtracting two pointer values. As a result, we can reliably check code that does type debasements far beyond merely casting between pointers and integers.

3.2 Object comparison

We define two routines as being equivalent on a path if *they write identical values to all memory transitively reachable from their return value and each of their formal arguments*. That is, pointer values (addresses) can differ as long as (1) the objects they point to do not, and (2) the pointer is to the same offset within the object. UC-KLEE checks this property by doing a mark and sweep of all reachable memory and using the constraint solver to prove that all non-pointer bytes are equal. In the concrete case, comparisons reduce to constants, avoiding expensive satisfiability queries. For symbolic bytes that neither routine modifies, the values in each address space snapshot contain identical symbolic expressions, which are trivially equivalent. If UC-KLEE detects a pointer, it adds the referenced objects (from each snapshot) to a queue for later traversal, rather than comparing the objects' addresses, which may differ between the two procedures. In the case of pointers stored into memory at symbolic offsets, it is possible for a particular value to resolve to multiple objects. In this case, UC-KLEE examines every pair of objects to which the two pointers could resolve. If a single pair of objects differs, UC-KLEE flags the error. Note that unless a function's return value or one of its formal arguments contains a pointer to a global variable, UC-KLEE does not automatically compare global variables because multiple implementations typically utilize an incompatible set of globals.

3.3 Lazy initialization

UC-KLEE uses a variation on *lazy initialization* [16] to dynamically allocate objects on an as-needed basis. This prior work was in the context of checking a single class method in type-safe Java; our implementation aims at cross checking non-type-safe C functions.

When the test harness marks function arguments as symbolic (the call to `klee_make_symbolic` on lines 17–18 of Figure 1), UC-KLEE also sets an "unbound" bit in shadow memory. At each pointer dereference `*p`, UC-KLEE examines this shadow memory to check whether `p` is an unbound pointer and hence must be lazily allocated. If a dereferenced pointer `p` is unbound, UC-KLEE:

1. Allocates an object of size n (discussed below) and marks it as containing unbound, symbolic bytes. Any dereference of this memory will similarly (recursively) allocate an object.

2. Constrains `p`'s referent to equal the address m of the allocated object ($p_{base} = m$), ensuring future dereferences map to the same object.
3. Constrains `p`'s "unbound" bit to *false* (bound). This prevents UC-KLEE from unintentionally allocating a new object during a subsequent dereference of the same pointer. Note that once a referent is bound, it can never become unbound, even if the object is explicitly `free`'ed by the procedure.
4. Constrains `p` to point within the allocated object: $m <= p < m + n$.

Each lazy allocation creates a unique memory object. A different approach would allow unbound pointers to resolve to existing objects of the same type. We did not use this method since it significantly increases the state space and can lead to many false positives. The main drawback of our current approach is that UC-KLEE cannot satisfy address constraints of code that specifically targets pointers to overlapping memory blocks, which limits coverage. We plan to revisit this decision as we encounter more real examples that require it.

Address constraints. Additional complexity arises when code compares unbound pointers. Suppose we take the true path of an equality comparison `x == y` and both `y` and `x` are unbound. If `x` is subsequently dereferenced and bound to a new object, we want `y` to be bound to the same object (and vice versa). We initially thought doing so would require constructing dynamic dependency graphs to determine where a pointer comes from (difficult in the general case given complex symbolic expressions). Fortunately, our shadow memory scheme makes the solution simple: merely constrain `x`'s shadow memory to equal `y`'s ($x_{base} = y_{base}$ and $x_{unbound} = y_{unbound}$). A subsequent binding of one will make the other bound as well. If only one pointer is unbound, we do the same thing, with the same effect. Note that, on the false path (where $x \neq y$), we do *not* add a constraint that their shadow memory differs because `x` and `y` may point to different bytes in the same object (and thus may share a referent).

Code sometimes compares two pointers to different objects using greater-than or less-than conditions (such as a binary tree sorted by address). While not strictly legal, UC-KLEE must nonetheless support such comparisons in order to be effective. Unlike equality, these comparisons add no additional constraints on referents. Instead, when a subsequent dereference causes UC-KLEE to lazily allocate an object, the object's address must satisfy all existing path constraints. To accomplish this, UC-KLEE allocates two large (e.g., 32 megabyte) memory pools on startup at high and low address ranges. Each allocation searches both of these pools for a block whose address does not violate the path constraints. If it cannot find one, the path terminates and UC-KLEE reports a runtime error. The most common cause we observed was code that specifically checks whether two pointers overlap. As we mentioned above, UC-KLEE does not allocate overlapping objects; thus, such constraints cannot be satisfied.

Allocation size. When UC-KLEE lazily allocates an object, it must choose a fixed size for that object. When an unbound pointer references a type of known size (e.g., an `int` or a `struct`), we simply allocate the exact size necessary to store that type. However, we cannot do the same when the pointer might refer to an array (since C arrays do not have statically known sizes). For example,

when a '`char *`' (string) is dereferenced, we do not know the size of the string. Unfortunately, making it too small will limit statement coverage, since many (or all) paths would terminate with out-of-bounds dereferences. Making it too large will disguise legitimate out-of-bounds errors in the code. Our current solution is a hack, but it seems to work well enough in practice. We first consider allocating an object of a user-specified minimum size (for our experiments, we found that a minimum size of 8 bytes works in most cases). UC-KLEE queries the constraint solver to test whether this allocation size would satisfy the current memory operation. If not, UC-KLEE iteratively doubles the guess. Within logarithmic time, UC-KLEE either finds a satisfying allocation size or reaches a user-specified maximum (e.g., 2 kilobytes). If the maximum fails, UC-KLEE terminates the path and reports a runtime error to the user.

Depth limits. Lazy initialization allows UC-KLEE to dynamically support hierarchical data structures such as linked lists and trees. To control the resulting path explosion, our tool limits this type of allocation to a user-specified depth limit, incrementing a counter on each nested allocation. If the depth counter exceeds the limit, our tool terminates the path and outputs a warning. Without a depth limit, the path space would quickly become unmanageably large.

3.4 Limitations

This section enumerates the known limitations of UC-KLEE. We are of course vulnerable to bugs in UC-KLEE or its environment modeling code. We check at the implementation level, which makes it easy to work with real code. However, it makes any verification claims true only for the specific compiler and architecture we used for our experiments. These guarantees may not hold on code where the compiler must make a choice among several unspecified behaviors (such as function argument evaluation order) or when running on a machine that differs in some observable way (such as word size or endianness).

During cross checking we only invoke a routine a single time and check it in isolation, missing behaviors that require multiple invocations or coordination across routines. In general, we may miss behaviors for code that depends on the values of addresses (e.g., greater-than or less-than relationships among objects allocated by `malloc`). More specifically, as discussed § 3.3, there are several limitations in our approach to lazy initialization. We assume that lazily allocated objects cannot alias existing objects, so we will not exercise code that checks for overlap (such as `memmem`) or expects an unbound pointer to point to the middle of an existing object (e.g., a circular buffer). We will also miss paths that need objects larger than our maximum size, since these will always terminate with an error. Indirect calls to unbound function pointers are unsupported at this time. On paths that have errors, UC-KLEE is unable to identify the root cause. Thus, it will not detect when two checked routines terminate with the same error type, but from different causes.

The underlying system, KLEE, must replace a symbolic value with a single concrete example when used as an allocation size or in a floating point operation. Thus, our tool may miss bugs exposed by different concrete values later in the

execution path. In addition, KLEE cannot handle routines that return structures, contain inline assembly, call unresolved external routines, or call external routines using symbolic arguments. When a path encounters one of these, UC-KLEE flags the routine as unverified.

4 Evaluation

This section shows that UC-KLEE works well at verifying equivalence by cross checking recent versions of two heavily-tested open source C libraries: UCLIBC, an implementation of the C standard library targeted at embedded devices, and NEWLIB, an embedded `libc` implementation by Red Hat used by Cygwin and Google Native Client. We demonstrate its effectiveness on three common use cases, cross checking: different implementations of the same interface, different versions of the same code, and identical code to find errors in the verification tool chain (in our case: the LLVM compiler and UC-KLEE itself).

We measure the quality of cross checking in two ways: (1) crudely, by the statement coverage it achieves, and (2) by whether checking exhausts all paths and terminates, since that verifies that the routines are equivalent up to a fixed input size when invoked a single time (modulo the limitations discussed in § 3.4).

It is a bit tricky to measure statement coverage for library code. We compute coverage of a cross checked routine as a percentage of the total number of LLVM instructions reachable from it, with the exception that when routine a calls another exported routine b that we will also cross check, we exclude b's instructions from a's coverage statistics. Usually, such calls can only exercise a small fraction of b's code (e.g., when a calls `printf` with a format string that just contains "`hello world`"). On the other hand, if a calls c and we *do not* generate a test harness for c, we *do* count its instructions since we conservatively assume it is an internal helper function that a should thoroughly exercise. Note: every instruction is included in the coverage statistics for at least one procedure.

For all experiments, we ran UC-KLEE on each procedure for up to 10 minutes, and allowed each procedure to read from up to 2 symbolic files of 10 bytes each (KLEE argument `--sym-files 2 10`). This was in addition to the symbolic arguments and environment generated by the cross checker. Our machine was a quad-core 2.8 GHz Intel i7 machine with 12GB of RAM running Fedora Linux 12.

4.1 Different implementations: NEWLIB vs. UCLIBC

Our first experiment cross checks NEWLIB's source repository from July 2010 against UCLIBC version 0.9.31. We modified both libraries to use UC-KLEE's memory allocator. We also disabled several UCLIBC internal startup and shutdown tasks that interfered with UC-KLEE. Finally, to keep the experiments manageable, we disabled optional features, such as wide character and locale support.

We automatically generated a test harness for each routine that both libraries implemented with the exception of variadic routines or those whose prototypes

differed. We could extend our system to support the former by generating multiple test cases for different numbers of arguments. Our experiment tested all other exported procedures, even those that demonstrate weaknesses in our tool.

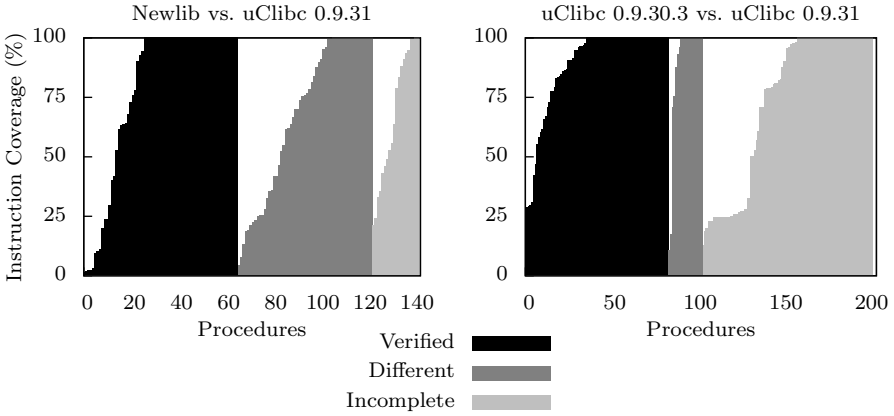

Fig. 3. Instruction coverage reported by our cross checking experiments. Each vertical bar represents a single procedure, sorted by coverage. The "incomplete" category includes routines whose analysis did not complete within 10 minutes or hit a limitation in KLEE or UC-KLEE. The median statement coverage for the left graph was over 90% (59 routines had 100%) and for the right was 100% (105 had 100%).

Figure 3 (left) shows the coverage reported by UC-KLEE. In the routines where UC-KLEE found no differences, it checked 66 to termination (versus 15 where it exceeded the time limit), thereby verifying equivalence for the given input size, despite many having entirely different structure and overall appearance to the human eye. The two implementations of `ffs` ("find first bit set") in Figure 4 are a good example: UC-KLEE exhausted all 33 paths in the test harness and terminated after 6.8 seconds, reaching 100% statement coverage.

```
1 : int ffs (int word) {
2 :   int i=0;
3 :   if (!word)
4 :     return 0;
5 :   for (;;)
6 :     if (((1 << i++)&word) != 0)
7 :       return i;
8 : }
```

(a) NEWLIB

```
1 : int ffs(int i) {
2 :   char n = 1;
3 :   if (!(i & 0xffff)) { n += 16; i >>= 16; }
4 :   if (!(i & 0xff))   { n += 8; i >>= 8; }
5 :   if (!(i & 0x0f))   { n += 4; i >>= 4; }
6 :   if (!(i & 0x03))   { n += 2; i >>= 2; }
7 :   return (i) ? (n+((i+1) & 0x01)) : 0;
8 : }
```

(b) UCLIBC

Fig. 4. Two implementations of `ffs` ("find first set bit") UC-KLEE verifies as equivalent despite their difference in appearance and approach.

UC-KLEE found differences in 57 of the 143 functions checked, at least 7 of which were real bugs—despite the code being heavily tested, actively used, and designed to do well-understood tasks. One interesting example was an error in NEWLIB's implementation of `remove` (Figure 5), which the POSIX standard mandates should work for both files and directories. UC-KLEE detects that NEWLIB returns `-1` (error) while UCLIBC returned `0` (success) when the symbolic

input `filename` could refer to a directory. This error would be difficult to detect statically.

```
1 : int _remove_r(struct _reent *ptr,
2 :               const char *filename) {
3 :   if (_unlink_r (ptr, path) == -1)
4 :     return -1;
5 :   return 0;
6 : }
7 :
8 : int remove(const char *filename) {
9 :   return _remove_r(_REENT,
10:                    filename);
11: }
```

(a) Newlib

```
1 : int remove(const char *filename)
2 : {
3 :   int saved_errno = errno;
4 :   int rv;
5 :   rv = rmdir(filename);
6 :   if ((rv < 0) && (errno == ENOTDIR)) {
7 :     _set_errno(saved_errno);
8 :     rv = unlink(filename);
9 :   }
10:   return rv;
11: }
```

(b) uClibc

Fig. 5. uc-klee detects that Newlib does not handle directory removal correctly.

We achieved high statement coverage in most but not all procedures. One common cause of low coverage is that we only cross check code using a single invocation. In certain cases, multiple invocations of a routine are required in order to reach additional code. In other cases, one routine may write values to globals or `static`s that are read by another. A good example of both is `atexit`, which registers routines to be run on program termination by `exit`. A simple extension would allow uc-klee to handle such cases.

4.2 Different versions of the same implementation: uClibc

To measure uc-klee's effectiveness at cross checking different versions of the same code, we used it to compare all functions that appeared in both uClibc 0.9.30.3 (March 2010) and uClibc 0.9.31 (April 2010) that were not byte-code identical. This selection yielded 203 routines (out of 399 possible), each of which uc-klee analyzed for up to 10 minutes.

Figure 3 (right) plots the instruction coverage. uc-klee revealed 2 previously unknown bugs and also detected 5 instances of bugs that were patched in the newer release. We elide a detailed discussion for space reasons and instead provide a brief example for each.

The newer version of uClibc introduced a bug in `ctime` (used to convert a time record to a string). The older version used a persistent internal structure (i.e., `static`) for storage that lacked thread safety. The newer version instead used a stack-allocated buffer that it never initialized. A sufficiently large input value caused the returned string to differ, which uc-klee detected.

uc-klee confirmed that a number of bugs present in uClibc 0.9.30.3 were corrected in version 0.9.31. One example is `unsetenv` (below). The old code (left) terminated with an out-of-bounds read when `__environ` is `NULL` (e.g., after a call to the function `clearenv`), while the new code (right) exited gracefully:

```
1 : char **ep = _environ;
2 : while (*ep != NULL) { ... }
```

(a) `unsetenv`: uClibc 0.9.30.3

```
1 : char **ep = _environ;
2 : if (ep) while (*ep != NULL) { ... }
```

(b) `unsetenv`: uClibc 0.9.31

4.3 Checking the checker: finding bugs in UC-KLEE and LLVM

A standard caveat in verification papers is that their claims are contingent on the correctness of the verifier and underlying compiler. One of our contributions is the realization that one can detect errors in both by simply attempting to prove the equivalence of identical code, thus turning the verification system on itself.

Finding compiler optimizer bugs. We check that an optimizer has correctly transformed a program path by compiling the same routine both with and without optimization and cross checking the results. With the usual caveats, if any possible value exists that would cause the path to give different results, UC-KLEE will detect it. If there is no such value, it has verified that the optimizer worked correctly on the checked path. If it terminates, it has shown that the optimizer transformed the entire routine correctly. Any discrepancies it finds are due to either compiler bugs or the routine depending on unspecified behavior (e.g., function evaluation order between sequence points). Because the library code we checked intends to be portable, even use of unspecified compiler behavior almost certainly constitutes an error.

We compared all 622 procedures in uCLIBC 0.9.31, compiled with no optimization (`-O0`) versus high optimization (`-O3`). This check uncovered at least one bug in LLVM 2.6's optimizer but did not expose its root cause. For `memmem`, UC-KLEE reported a set of concrete inputs where the unoptimized code returned an offset within `haystack` (the correct result), while the optimized code returned `NULL`, indicating that `needle` was not found in `haystack`. We confirmed this bug with a small program. Since LLVM is a mature, production compiler, the fact that we immediately found bugs in this simple way is a strong result. We found a total of 70 differences, but because of time constraints could not determine whether they were due to this bug or others. Future work will be necessary to test optimization levels between these two extremes and attempt to automatically find a minimal set of optimization passes that yield an observable difference.

Finding UC-KLEE bugs. In general, tool developers can detect verifier bugs by simply cross checking a routine against another identical copy of itself (i.e., compiled at the same optimization level). This check has been a cornerstone of debugging UC-KLEE—it often turned up tricky errors after development pushes.

The UC-KLEE bugs we found fell into two main categories: (1) unwanted nondeterminism in UC-KLEE and its environmental models, which makes it hard to replay paths or get consistent results, and (2) bugs in our initial pointer tracking approach. In fact, as a direct result of the tricky cases cross checking exposed in this pointer tracking implementation, we threw it away and instead designed the much simpler and robust method in Section 3.1.

4.4 Results summary

Figure 6 summarizes the results presented in this section. The "KLEE Limitations" row describes procedures that resulted in incomplete testing due to

limitations in the underlying KLEE tool: inline assembly (141 procedures), external calls with symbolic arguments (206), and unresolved external calls (17). "UC-KLEE Limitations" are cases where the tool failed to lazily allocate objects either because the required size of the object exceeded our specified maximum of 2KB (20 procedures) or UC-KLEE was unable to allocate an object whose address satisfied the path constraints (117 procedures). Note that limitations resulted in individual paths being terminated. As a result, certain procedures encountered a variety of limitations on different paths. In particular, a procedure deemed "KLEE limited" may have also encountered UC-KLEE limitations, although the converse is not true.

	NEWLIB/ UCLIBC	UCLIBC Versions	LLVM Optimizer	UC-KLEE Self-check
Procedures Checked	143	203	622	622
Procedures Verified	66	84	335	335
Differences Detected	57	20	70	12
No Differences (timeout)	15	30	85	91
KLEE Limitations	4	56	94	147
UC-KLEE Limitations	1	13	38	37
100% Coverage	59	105	367	375
Mean Coverage	72.2%	80.7%	85.6%	85.6%
Median Coverage	90.1%	100.0%	100.0%	100.0%

Fig. 6. Breakdown of procedures checked in each experiment.

5 Related Work

This paper builds on the many recent research projects in symbolic execution, such as [2, 3, 12, 16, 22], as well as several pieces of our past work. About a decade ago, we showed how to avoid the need for manual specification by cross checking multiple implementations in the context of static bug finding [9], an idea we later used with symbolic execution [2, 3]. This latter work only handled complete applications or routines run on manually constructed symbolic input; this paper shows how to easily check code fragments with unbound inputs and how to use cross checking to find bugs in the checking infrastructure itself. This paper is related to under-constrained execution [8], but modified to support the cross checking context, where many of the tricky issues are elided.

Many previous approaches to software checking have been specification based, requiring extensive work on the part of the user. One example is the use of model checking to find bugs in both the design and the implementation of software [1, 4, 5, 11, 13, 14], which requires manually building test harnesses. A second example is recent verification work that checks code manipulating complex data structures against manually constructed specifications [6, 7, 10, 18]. While both can exploit their specifications to reduce the state space, they require far more user effort than UC-KLEE.

Similar work has attempted to cross check largely identical code by using over-approximation to filter out unchanged portions of Java code [20]. While their technique is sound with respect to verification, a consequence of over-approximation is that reported differences may not concretely affect the output.

In contrast, UC-KLEE generates test cases that supply concrete inputs to expose behavioral differences in the code.

Earlier work in cross checking has focused on combinational circuits in hardware [4, 17, 19]. While an important milestone, hardware verification is simpler than general purpose software equivalence checking, which includes loops, complex pointer relationships, and other difficult constructs.

Smith and Dill [23] recently verified the correctness of real-world block cipher implementations. Their work exploits the key properties that block ciphers have fixed input sizes and loop iterations, enabling full loop unrolling. They developed several constraint optimizations that we hope to adapt for cross-checking general-purpose code.

6 Conclusion

We have presented UC-KLEE, a tool that often makes cross checking two implementations of the same interface easier than writing even a single test case. The preliminary results demonstrate the usefulness of our approach, which often exhaustively explores all paths and verifies two procedures as equivalent up to a given input size.

We are currently building an improved version of UC-KLEE that is capable of cross checking individual code patches rather than complete routines, thereby reducing the problems of path explosion. Further, by jumping to the start of a patch, it will more robustly support code not easily checked by dynamic tools (such as device driver code). We plan to use this ability to check that kernel patches only remove errors or refactor code (for simplicity or performance) but do not otherwise change existing functionality.

Acknowledgements

The authors would like to thank Philip Guo, Diego Ongaro, and Amanda Ramos for their valuable feedback. This work is supported by the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024 and by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-0645962. The paper's words only represent the views of the authors.

References

1. Brat, G., Havelund, K., Park, S., and Visser, W. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE 2000)*.
2. Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)* (Dec. 2008), pp. 209–224.
3. Cadar, C., and Engler, D. Execution generated test cases: How to make systems code crash itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN '05)* (August 2005).

4. Clarke, E., and Kroening, D. Hardware verification using ANSI-C programs as a reference. In *Proc. of the Asia and South Pacific Design Automation Conference (ASP-DAC 2003)*.
5. Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd International Conference on Software Engineering (ICSE '00)* (June 2000), pp. 439–448.
6. Deng, X., Lee, J., and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. of the 21st IEEE International Conference on Automated Software Engineering* (2006), pp. 157–166.
7. Elkarablieh, B., Marinov, D., and Khurshid, S. Efficient solving of structural constraints. In *Proc. of the International Symposium on Software Testing and Analysis* (2008), pp. 39–50.
8. Engler, D., and Dunbar, D. Under-constrained execution: making automatic code destruction easy and scalable. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)* (July 2007).
9. Engler, D., Yu Chen, D., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (2001).
10. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., and Marinov, D. Test generation through programming in UDITA. In *Proc. of the 32nd International Conference on Software Engineering (ICSE '10)* (2010), pp. 225–234.
11. Godefroid, P. Model Checking for Programming Languages using VeriSoft. In *Proc. of the 24th Annual Symposium on Principles of Programming Languages (POPL '97)* (Jan. 1997), pp. 174–186.
12. Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)* (June 2005), pp. 213–223.
13. Holzmann, G. J. The model checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.
14. Holzmann, G. J. From code to models. In *Proc. of the Second International Conference on Applications of Concurrency to System Design (ACSD '01)* (June 2001).
15. Jones, R., and Kelly, P. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. of the International Workshop on Automatic Debugging* (1997).
16. Khurshid, S., Pasareanu, C. S., and Visser, W. Generalized symbolic execution for model checking and testing. In *Proc. of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2003).
17. Kuehlmann, A., and Krohm, F. Equivalence checking using cuts and heaps. In *Proc. of the 34th annual Design Automation Conference* (1997), DAC '97, pp. 263–268.
18. Marinov, D., Andoni, A., Daniliuc, D., Khurshid, S., and Rinard, M. An evaluation of exhaustive testing for data structures. Tech. rep., MIT Computer Science and Artificial Intelligence Laboratory Report MIT-LCS-TR-921, 2003.
19. Mishchenko, A., Chatterjee, S., Brayton, R., and Een, N. Improvements to combinational equivalence checking. In *Proc.of the 2006 IEEE/ACM international conference on Computer-aided design* (2006), ICCAD '06, pp. 836–843.
20. Person, S., Dwyer, M. B., Elbaum, S., and Păsăreanu, C. S. Differential symbolic execution. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)* (2008), pp. 226–237.
21. Ruwase, O., and Lam, M. S. A practical dynamic buffer overflow detector. In *Proc. of the 11th Annual Network and Distributed System Security Symposium* (2004), pp. 159–169.
22. Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *Proc. of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)* (Sept. 2005), pp. 263–272.
23. Smith, E. W., and Dill, D. L. Automatic formal verification of block cipher implementations. In *Proc. of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD '08)* (2008), pp. 1–7.