BLT: Better Low-effort Testing

Aaron Bembenek
bembenek@g.harvard.edu

Todd Lubin
tlubin@college.harvard.edu

Lily Tsai
lilliantsai@college.harvard.edu

***Abstract*—Thoroughly testing the behavioral equivalence of two implementations of the same API is difficult. Various tools and techniques are available to analyze functions in isolation, but reasoning on the function level fails to account for possible interactions between functions. The use of shared state warrants a different approach to code analysis that has a global view of the program. The task at hand is especially challenging as the client of the API can call functions in an entirely arbitrary order, necessitating an analysis focused on the interleaving of function calls. Our paper makes two contributions: (1) we evaluate the hypothesis that the combination of concrete swarm testing with symbolic execution is effective for testing API code, and (2) we provide a tool, BLT, that makes it easy to combine these powerful testing techniques to compare the functional equivalence of two C++ classes that share an API.**[1]

I. Introduction

Thoroughly testing the behavioral equivalence of two implementations of the same API is difficult. However, in many situations it is very important to prove semantic equivalence of implementations. For example, one might refactor the code for a web server in order to support a new type of query. This change should often be indistinguishable to clients in that making a sequence of old request types should return the same sequence of values as before. It is crucial to keep the same semantics because clients have already written code against the old implementation. In another example, imagine refactoring for performance optimizations; it is important to verify that correctness has not been compromised. One common approach to verify code is to run a series of regression tests. While this is an easy and fast way to give some assurance of correctness, it is entirely dependent on the quality of the often hand-written tests. The code is checked over a small number of paths, but remains vulnerable to bugs and other unexpected behavior in unexplored paths. In fact, part of the challenge of testing for API equivalence is that it quickly becomes infeasible to explore all the possible paths through the code of an API. Even for a small, fixed length sequence of calls, there is a exponential blowup in the number of paths due to the interleaving of functions and the arguments given to each function. There are multiple existing tools and techniques that explore paths in different ways. We focus our attention on two existing techniques that explore paths using contrasting approaches: swarm testing[1] and symbolic execution[2]. In the context of testing API code, we conjecture that a careful combination of the two will result in the exploration of the most interesting paths in order to find bugs.

Swarm testing is a form of random testing predicated on the idea that it is more effective to test many different randomly generated test configurations (“swarms”) than to repeatedly test some single “optimal” test configuration. A configuration might be, for example, a list of API calls that can be randomly made during a test. Whereas in traditional random testing every API call might be included in the single configuration, in swarm testing multiple configurations are run, each of which only makes calls from a subset of the API. The originators of swarm testing found that this type of feature omission actually increases the diversity of the tests and consequently improves coverage over traditional random testing. Swarm testing is able to efficiently simulate long sequences of calls and achieves diverse path exploration by randomizing which functions are included in a given swarm.

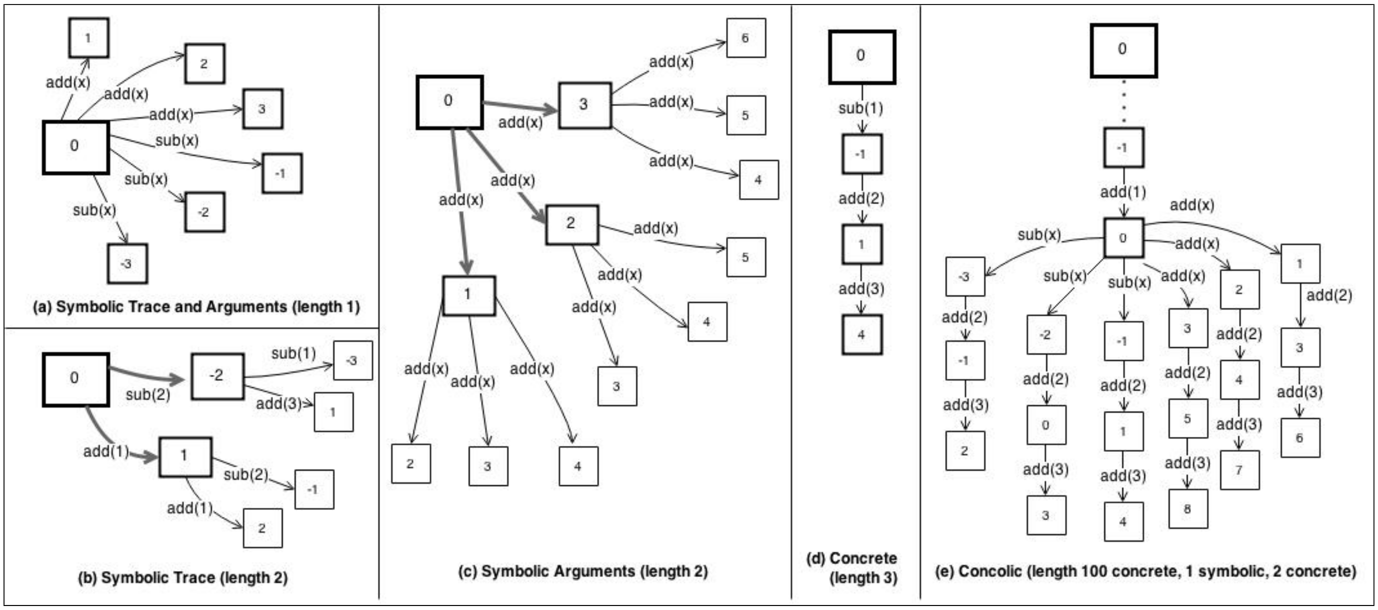
Symbolic execution is a form of testing that searches all paths that can be taken through a given piece of code. Certain values (such as inputs) are designated to be symbolic, meaning that they are completely unconstrained. In the execution of the program, the use of a symbolic value can branch execution, allowing simultaneous exploration of all feasible program paths. Symbolic execution can be used to test API code by making symbolic the order of function calls, the arguments to the calls, or both. Consequently, from a given program state, it is able to widely explore the next couple steps that can be taken. However, symbolic execution can be very inefficient due to path explosion, and consequently it is not a practical technique for exploring long sequences of function calls.

These two approaches to exploring paths are complementary to each other in the context of testing API code. While swarm testing can arrive at “interesting” program states by its use of random test configurations, it does not do thorough testing at these states. While symbolic execution does thorough testing, it cannot be used alone in order to explore deep program states. We introduce the idea of concolic traces, sequences of function calls consisting of both concrete and symbolic parts. Our paper makes two contributions: (1) we evaluate the hypothesis that the use of concolic traces is effective for testing API code, and (2) we provide a tool, BLT, that makes it easy to combine these powerful testing techniques to compare the functional equivalence of two C++ classes that share an API.

II. Design

In order to test for the behavioral equivalence of two implementations, we call the same sequence of functions with the same arguments on both implementations. The choice of functions, arguments, and the use of symbolic execution

[1] The source code of BLT is available at https://github.com/tlubin/blt.



(a) Symbolic Trace and Arguments (length 1)

(b) Symbolic Trace (length 2)

(c) Symbolic Arguments (length 2)

(d) Concrete (length 3)

(e) Concolic (length 100 concrete, 1 symbolic, 2 concrete)

Fig. 1. Different techniques for exploring program paths

is entirely specified by custom-written or default "concolic traces." If at any point in execution the return values differ across implementations, we output a C++ file that can be run deterministically to trigger the divergent behavior.

A concolic trace is a list of nodes, where each node is a 4-tuple (F, n, s_c, s_a). F is the set of functions that can be called from that node. n is an integer delimiting how many function calls should be made. s_c and s_a are booleans stating whether the functions to be called should be represented by symbolic values and whether symbolic arguments should be used when making the function calls, respectively. Running a concolic trace is equivalent to traversing the sequence of nodes and carrying out the directions specified in each node. If function calls are concrete, then each function call is a random element chosen from F. If arguments are concrete, random arguments are generated using either BLT's generators for primitive types or a user specified generator function. For example, assume we have an API that specifies the functions $\{add, sub\}$. Fig. 1 demonstrates the difference between the following approaches to testing: (a) symbolic arguments and function calls, (b) symbolic function calls, (c) symbolic arguments, (d) concrete arguments and function calls, and (e) mixture of symbolic and concrete. Arguments to add and sub range over the numbers 1 to 3 and x represents a symbolic variable. Our formulation of concolic traces allows one to easily specify each of these cases. Case (e) would be specified as:

$$\langle(\{add\}, 100, F, F), (\{add, sub\}, 1, T, T), (\{add\}, 2, F, F)\rangle$$

If the user doesn't provide customized traces, BLT generates default traces. Each generated trace consists of three nodes. The first node stipulates concrete function calls with concrete arguments for a default length (500). The set of functions to be run in this node is a random subset of all the functions in the API. The second node stipulates symbolic function calls with symbolic arguments over all the functions in the API for a default length (1). The third node, like the first, stipulates concrete function calls with concrete arguments for a default length (50). Again, the functions that can be called from this node are a random subset of the API functions. BLT generates multiple traces using this specification and executes all of them in search for divergent behavior. Formally, a default trace is specified as:

$$\langle(S_1, 500, F, F), (API, 1, T, T), (S_2, 50, F, F)\rangle$$

where S_1, S_2 are randomly generated subsets of API, the set of all functions to be tested.

There are many motivations for this choice of a default trace. The first node can be thought of as simply swarm testing. We allow for a long length in order to build up interesting state. At this point, the purely symbolic node of length one allows this state to be explored very thoroughly while controlling path explosion. For instance, on an implementation of a red-black tree, this trace explored 200 paths on average (depending on the number of inserted values), a very manageable number. Increasing the length of the symbolic node to two quickly increases the number of paths explored to 1300. While it would be more comprehensive to include a lengthier symbolic node, it is too expensive to do so. Having a short concrete node after the symbolic node allows further branching to continue as it is possible for state to become partially symbolic as a result of a function call with a symbolic argument. In addition, all these concrete calls must be performed on all the branches, further motivating a short length.

III. Implementation

On a high level, BLT works by generating a test harness that encodes a set of traces. BLT symbolically executes each trace in isolation looking for divergence in the behavior of the two classes being compared. Whenever it detects that such an error has occurred during the execution of a trace, BLT creates a fully-concretized version of the trace that can be deterministically replayed. More specifically, BLT is a Python script that compares the behavioral equivalence of two C++ classes that have a shared API. BLT requires the user to supply a JSON file specifying basic information such as the names of the classes to be compared, relevant source and header files, a list of functions to test and, if desired, a specification of custom traces to use.

A. Harness Generation

BLT generates the C++ test harness by injecting the encoding of the user-supplied or default traces into Mako templates. The harness is modularized by trace, to the effect that a single execution of the harness will only run the encoding of a single trace. A trace is translated into the harness by individually translating the relevant trace nodes and sequencing them together. The encoding of a particular trace node in the harness depends on whether the function calls and arguments have been specified to be symbolic. Consider the following cases for the trace node (F, n, s_c, s_a):

- $s_c = 0 \land s_a = 0$. Here both the function calls and the arguments are concrete. BLT randomly picks a function from F n times and for each choice encodes a function call to that function with randomly generated arguments.
- $s_c = 1 \land s_a = 0$. The function calls are symbolic but the arguments are concrete. BLT encodes n symbolic branch points in the harness, where the first statement of each path resulting from the branch point is a call to a different function in F. This is akin to symbolically picking a function from F n times. Randomly generated arguments are supplied at each function call.
- $s_c = 0 \land s_a = 1$. The function calls are concrete but the arguments are symbolic. BLT randomly picks a function from F n times, but each choice is encoded as a function call to that function with new symbolic arguments.
- $s_c = 1 \land s_a = 1$. Both the function calls and the arguments are symbolic. As above, BLT encodes in the harness n symbolic branching points, and new symbolic arguments are supplied at each function call.

At the return of every function call, whether symbolic or concrete, BLT encodes an assertion that the two return values are the same.

BLT generates all random values in Python and hard codes them into the C++ template. This includes the sequence of concrete function calls and any concrete arguments. On the other hand, no symbolic work is performed by BLT when creating the harness. Rather, symbolic directives are encoded in the C++ code to be handled at runtime with KLEE. Where symbolic arguments are needed, variables of the proper type are created and then christened as symbolic. Branch points representing symbolic function calls are encoded as `switch` statements over symbolic integers. For example, Fig. 2 demonstrates a possible harness that could be generated for the trace $\langle(\{insert, remove\}, 5, F, F), (\{member, remove\}, 2, T, T)\rangle$. Lines 5-9 encode the concrete calls specified in the first node and the `for` loop from lines 11-25 encodes the two symbolic function calls from the second node.

```
1  void trace0() {
2    Bag1* v1 = new Bag1();
3    Bag2* v2 = new Bag2();
4
5    call_remove(v1, v2, false);
6    call_insert(v1, v2, false);
7    call_insert(v1, v2, false);
8    call_remove(v1, v2, false);
9    call_insert(v1, v2, false);
10
11   for (int i = 0; i < 2; ++i) {
12     unsigned idx1;
13     klee_make_symbolic(&idx1, sizeof(idx1), "idx1");
14     klee_assume(idx1 < 2);
15     switch (idx1) {
16       case 0: {
17         call_member(v1, v2, true);
18         break;
19       }
20       case 1: {
21         call_remove(v1, v2, true);
22         break;
23       }
24     }
25   }
26 }
```

Fig. 2. Auto-generated trace encoding

Fig. 3 shows the wrapper used when testing a particular function call. The function `blt_args::get_arg` returns a primitive value that was randomly generated in Python. BLT's default random argument generator chooses between potential corner case values and a randomly generated value selected from a limited range. For example, the generator for `int` chooses equally between `INT_MAX`, `INT_MIN`, 0, and integers within the range -50 to 50. The default generators are designed to be used across a wide variety of programs while having no specific knowledge of the implementation of the particular application being tested. However, users also have the option of supplying custom random argument generators. Since these generators can reference the internal state of the class implementations they might be able to provide tailored inputs that are more interesting than those generated by BLT.

B. Test Execution and Replay

The harness and relevant source and header files are compiled to LLVM bitcode and linked together. BLT uses KLEE as a black box to symbolically execute this bitcode. If the harness encodes n traces, n distinct calls to KLEE are made, ensuring that there is no contamination from one test run to the next. BLT symbolically executes each trace encoding from a given harness in isolation. Branch points corresponding to symbolic

```
void call_member(Bag1* v1, Bag2* v2, bool is_sym) {
  int arg0;
  if (is_sym) {
    klee_make_symbolic(&arg0, sizeof(arg0), "arg0");
  } else {
    arg0 = *(int*)(blt_args::get_arg("int"));
  }

  bool r1 = v1->member(arg0);
  bool r2 = v2->member(arg0);

  if (r1 != r2) {
    failure();
  }
}
```

Fig. 3. Auto-generated "wrapper" for testing calls to `member`

function calls and the use of symbolic variables generate new paths in the symbolic execution.

If a symbolic execution path leads to a cross-implementation assertion failure or another type of fault, BLT creates a completely concrete trace encoding that allows for a deterministic replay of the failing run. Users can fix the bug they believe led to the failure and then replay the encoding to check their work.

To identify failures, BLT scans KLEE's output files for errors that were found during symbolic execution. To reconstruct the trace encoding for each path that led to the error, BLT needs to obtain two pieces of information. First, BLT must find concrete values to replace any symbolic values that were used during execution, including both symbolic arguments and the symbolic values associated with function calls in a symbolic trace. Since KLEE provides test inputs for each explored path, this information is easily extracted from KLEE's output files. Second, BLT must find and identify the exact function call that caused the test to fail. This is done by generating a simple harness that reruns the function calls in order using the concrete values gleaned from KLEE and exits immediately upon failure. Finally, the failing trace can be then encoded as a C++ source file that can be independently compiled and executed. It has no dependencies specific to BLT or KLEE. Fig. 4 presents a replay file for the example trace from earlier in this section for a test that failed when the first symbolic call was to `member` with a symbolic argument that could be concretized to 0.

C. Enhancements and Extra Features

BLT can also be used to find various types of bugs in a single C++ class that implements an API by simply specifying BLT to check divergent behavior between two instances of the same class. In this way, any error that KLEE finds (e.g. assertion error, null pointer dereference) will be detected, as well as nondeterministic behavior in the class implementation.

Lastly, BLT supports the use of pre- and post-conditions that may be specified as functions within the source files of each implementation. The pre-conditions prevent KLEE from symbolically exploring paths that are infeasible (i.e. calling a function when the preconditions to the function have not yet been satisfied), and therefore cut down the search space that is symbolically explored. The post-conditions act as future assertions that can be checked after every run to a certain function, such as whether a self-balancing binary tree is still balanced after an insertion or deletion.

```
Bag1 *v1= new Bag1;
Bag2 *v2= new Bag2;

int arg0_0 = -19;
(void) v1->remove(arg0_0);
(void) v2->remove(arg0_0);

int arg1_0 = 0;
(void) v1->insert(arg1_0);
(void) v2->insert(arg1_0);

int arg2_0 = 2147483647;
(void) v1->insert(arg2_0);
(void) v2->insert(arg2_0);

int arg3_0 = 0;
(void) v1->remove(arg3_0);
(void) v2->remove(arg3_0);

int arg4_0 = 0;
(void) v1->insert(arg4_0);
(void) v2->insert(arg4_0);

int arg5_0 = 0;
bool r1 = v1->member(arg5_0);
bool r2 = v2->member(arg5_0);
if (r1 != r2)
  failure();
```

Fig. 4. Excerpt from auto-generated replay file

IV. Limitations and Future Work

While BLT does provide assurances of functional equivalence, it is limited in its scope by the support BLT provides for different types of functions and argument generation, as well as the limitations of symbolic execution.

A. Primitive Return Values

BLT is currently relatively constrained in the scope of programs it can support. Verification of functional equivalence is currently implemented as an assertion of equivalent return values from calling the same function in both implementations. However, since the BLT harness has no knowledge of any internally declared composite types (such as a `struct`), verification is limited to only those functions with return values that are primitive (or references to primitive values). For example, when comparing two implementations of a binary tree in which `get_member()` returns a `tree_node`*, BLT cannot verify that the two return values are indeed equivalent because the encoding of a `tree_node` is not visible to BLT. In order to extend the applicability of BLT, future work may include parsing the program source code for information of these complex types, or allowing the programmer to specify a comparison function for types defined within the implementation.

B. Argument Generation

Another limitation of BLT is in its argument generation. BLT's current default argument generator only supports `int`, although other primitive types could easily be supported. BLT does not have the capability to generate non-primitive argument types, and so the programmer *must* supply her own argument generator for any custom data types defined within the program (such as a `tree_node` of a binary tree). Arguably, the requirement for the programmer to write an argument generator for non-primitive types is a limitation faced by any testing harness that wishes to automate calling functions that require non-primitive arguments.

Furthermore, BLT cannot intermix concrete and symbolic arguments within a function. For example, if function f takes two arguments `int x` and `node* root`, the programmer cannot specify that `x` should be symbolic and `root` should be concrete. While there might be cases in which the user would like to generate some arguments concretely (to limit the symbolic search space, or to use an argument generator to generate interesting inputs), but also keep other arguments symbolic, the user is currently limited to either making both arguments concrete or both symbolic. Future versions of BLT intend to allow a higher granularity with which the user can specify what type of argument (concrete or symbolic) she wishes to use for each argument of a function.

C. Limitations of KLEE

While symbolic execution allows BLT to explore a wider breadth of paths and provide stronger guarantees than would a purely concrete harness, symbolic execution also brings with it several limitations. Due to the exponential nature of symbolic path exploration, BLT's harness with symbolic nodes (with traces symbolic, arguments symbolic, or both symbolic) runs significantly slower than does a purely concrete harness. A user of BLT must be cautious to avoid enormous exponential path explosion when specifying any custom traces. While default traces do not require the programmer to know the details of symbolic execution, one of the current limitations of BLT is the requirement for the user to have an adequate understanding of how symbolic execution works in order to take full advantage of customized trace specification. Furthermore, KLEE does not know how to make certain common data structures symbolic (e.g. linked list), so the user must deal with these arguments concretely.

D. Usability

One of the major focuses of improving BLT will be to reduce the required annotations and effort from the user. Future versions intend to provide a cleaner, clearer way for the user to specify traces and program information, perhaps by providing templates or a GUI of some sort, which may help the user visualize and understand how to use symbolic nodes. In addition, BLT could parse the LLVM IR of the program implementations to gather data about the functions (e.g. argument types and return types) to reduce the work of the user.

E. Extra Features

While BLT can be used to find various bugs (e.g. assertion errors or segmentation faults) in a single C++ class by comparing the behavior of two instances of the C++ class, such a test could lead to incorrect results if the objects share and reference mutable class variables.

BLT's support for pre- and post-conditions is very limited. First, BLT does not allow for the specification of a pre- or post-condition on one implementation but not the other. Second, because the concrete traces are generated within the Python script and hard-coded into the generated harness, BLT may still produce concrete traces that are infeasible with respect to the required preconditions. Thus, preconditions are most useful when running the symbolic nodes of traces to cut down on the search space. Future versions of BLT intend to account for preconditions even within the concrete trace, by either moving the generation of the trace to the C++ code or providing some template through which the user can specify a limited set of preconditions that can be checked in the Python code. Preconditions can also be extended to check an additional feature of equivalence, i.e. if the preconditions of calling a function f of implementation 1 are satisfied, but the preconditions of calling f of implementation 2 are not satisfied, BLT could treat this as an error to report.

F. Error Reports and Replays

BLT also has the potential to generate debugging information for the user beyond failing traces and trace argument inputs. While the focus of BLT has been in generating these failing inputs, future versions of BLT intend to make the replay files a more useful tool for the user. In particular, a technique similar to delta debugging[3] could be used on the trace encoded in the replay file to minimize the number of functions presented to the user as necessary for failure. By returning to the user a minimal sequence of function calls that lead to a failure, the user might more easily reason about the source of the failure.

In addition, BLT could execute different function calls of the replay trace with symbolic arguments in order to present to the user a group of input arguments that cause the trace to fail, and a group of input arguments at a particular point in the failing trace; these inputs which may help the user more easily detect the error in her code.

V. Evaluation

Our evaluation of BLT is broken into a performance evaluation and a usability evaluation. Throughout the evaluation, we used two bug-free implementations of a bag (i.e. multiset) data structure, one implemented as an array and the other as a red-black tree. We generated 500 mutated red-black tree implementations by injecting small changes at random into the source code for the correct tree implementation.[2] Any functional difference detected between the array implementation and the mutated tree implementation is thought of as "finding the bug."

[2] Faults were injected using an open-source Python script written by Arun Babu (http://archive.is/arun-babu.com).

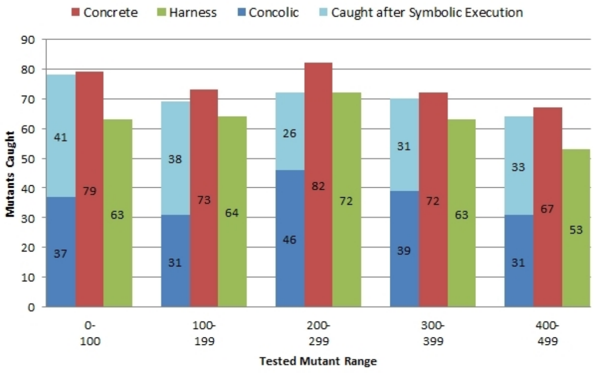


Fig. 5. Mutants caught per method, separated per 100 mutants

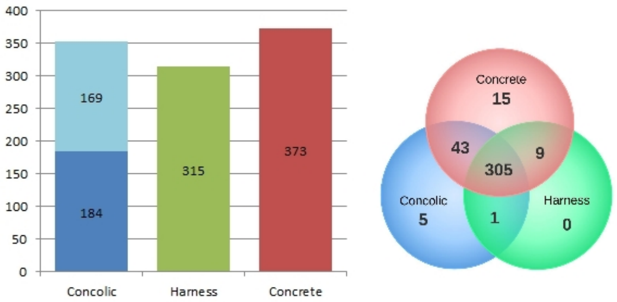


Fig. 6. Total mutants caught per method

A. Performance Evaluation

We tested our hypothesis that the use of concolic traces would achieve the benefits of deep testing characteristic of concrete testing as well as wide testing characteristic of symbolic execution. In doing so, we conjectured that there would be certain classes of bugs that concolic traces would be able to find that purely concrete would have a small chance of detecting. We used two different techniques of concrete testing to evaluate the bug-finding effectiveness of purely concrete tests. The first, a test harness custom-built for this specific implementation of the red-black tree, simulates a relatively comprehensive test suite that a programmer may write to test her program implementation. In this case, the programmer has full knowledge of the internals of her program, and can optimize the test harness for testing her exact implementation. The harness repeats the following task: randomly insert a random number, delete a random number, or check membership of a random number. It makes this choice 100,000 times. On calls to member, it asserts that the return value from the tree implementation is as expected (see Fig. 9 for the harness implementation). The second technique of concrete testing is a set of purely concrete traces constructed using BLT, generating random arguments with BLT's default argument generators. This testing is general in that it is agnostic to the meaning of the functions, but utilizes swarm testing, unlike the custom harness.

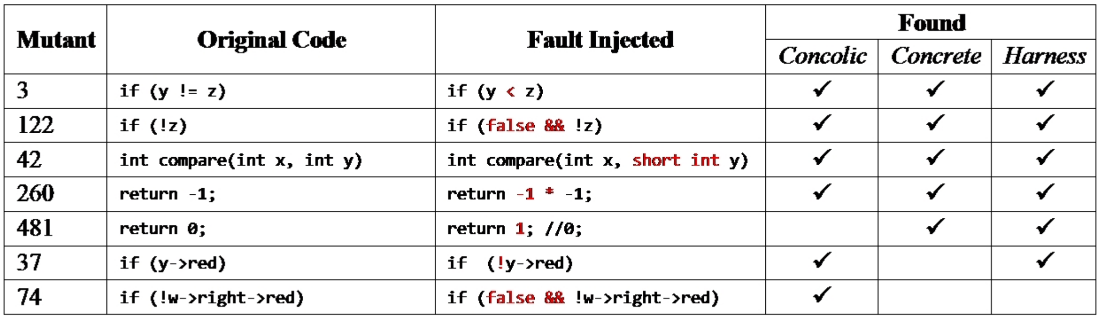
Concolic traces were run in KLEE with a timeout of 60 seconds. Each mutant was tested with 10 generated traces using BLT's default trace specification. The generated concrete traces mirrored the concolic traces in structure, utilizing two fully concrete nodes of length 500 and 50 (default trace without the symbolic middle node). The rationale behind this was to try to isolate the effect of having the single symbolic execution node. With no path explosion, timeouts were not an issue for the concrete traces (with the exception of mutants involving infinite loops). Including 2 concrete nodes allow for 2 different swarms to be used in any one trace, eliminating useless traces (e.g. ones that only include the ”insert” function). Due to the much greater execution speed of concrete execution compared to symbolic execution, 100 generated concrete traces were run for each mutant. The results are in Fig. 5 and Fig. 6.

Overall, concolic traces found 353 mutants, concrete swarms found 373 mutants, and harness found 315 mutants. Concolic traces found 6 mutants that concrete swarm testing failed to detect, 1 of which was caught by the custom harness. Concrete swarm testing found 24 mutants that concolic failed to detect, 9 of which were caught by the harness. Concolic traces found 48 mutants that the harness failed to detect, while the harness found only 9 mutants that concolic failed to detect. Concrete swarm testing found 58 mutants that the harness failed to detect, while the harness found only 1 mutant that swarm testing failed to detect; this mutant was successfully caught by concolic.

1) Comparison: Random Swarm Testing: This data validates the power of random swarm testing. Out of the mutants caught by concolic but not concrete random swarm testing, 5 of the 6 were related to a check on the color of the node (e.g. a change from `if (y->red)` to `if (!y->red)`, Table I mutant 37). Out of the mutants caught by concrete swarm testing but not concolic, 9 out of 9 were related to a check on the color of the node. While some of these bugs appear to be the same injected faults, they occur in different positions in the source code. From this, we concluded that bugs caught by one method but not another were due to random chance. This was further supported when rerunning the tests on the 6 mutants found with concolic traces but not found with concrete traces. Concrete swarms were successfully able to detect 5 of the 6 of these mutants when given more attempts. However, there does appear to be something subtle about mutant 74 (as shown in Table I) that suggests it is easier to find with a concolic approach than a purely concrete approach. Mutant 74 alters the conditional that checks whether or not the child of a node is colored red. The reason why this bug might be so hard to find concretely is because it is only triggered during a deletion of a node when the tree has a very specific structure. This structure is unlikely to be built up concretely, yet much more likely to be explored with the introduction of symbolic execution.

46.5% (169 out of 353) mutants caught by concolic traces

TABLE I
SAMPLE OF MUTANTS CAUGHT



Mutant	Original Code	Fault Injected	Found		
			Concolic	*Concrete*	*Harness*
3	`if (y != z)`	`if (y < z)`	✓	✓	✓
122	`if (!z)`	`if (false && !z)`	✓	✓	✓
42	`int compare(int x, int y)`	`int compare(int x, short int y)`	✓	✓	✓
260	`return -1;`	`return -1 * -1;`	✓	✓	✓
481	`return 0;`	`return 1; //0;`		✓	✓
37	`if (y->red)`	`if  (!y->red)`	✓		✓
74	`if (!w->right->red)`	`if (false && !w->right->red)`	✓		

occurred after the execution of a symbolic function call. The addition of the symbolic node appears to have compensated for the reduced number of traces (10) that concolic tests ran compared to concrete swarm testing (100). The inclusion of 10 times more traces in concrete swarm testing allowed for the concrete tests to generate many different sequences of arguments and function calls, increasing the probability that a particular mutant would be caught. However, the symbolic node, in its comprehensive search of all possible program states after a particular function call, allowed concolic traces to catch many of the same mutants with 10 times fewer traces.

Nevertheless, it remains clear that concrete swarm testing, given enough traces, slightly outperforms the default concrete-symbolic-concrete, i.e. concolic, traces that BLT provides as default. We conjecture that this result may have occurred because these constructed mutants were injected with exactly those bugs that are most easily found using concrete, random swarm testing. In Table I, we give examples representative of the 500 injected faults; these faults consisted in (1) changing comparisons from equality to inequality or negation (and vice versa) (mutant 3); (2) adding additional conditions to conditional statements (mutant 122); changing data types (mutant 42); (4) multiplying by an arbitrary integer (mutant 260); (4) and (5) altering return values (mutant 481).

These types of bugs can be categorized as the worst case use-case for our concolic trace approach to finding bugs. Arguably, the simplest method to detect these bugs would be random concrete testing: these bugs are better suited for random swarm testing with no use of symbolic execution, in particular since they require neither complex state, nor particular specific argument values, in order to be caught. Alteration of return values, for example, clearly causes a functional equivalence error that occurs regardless of what arguments are passed into the function. Changing conditional statements, for example $x == y$ to $x \leq y$, that then lead to changing function behavior, can be easily discovered by executing the function enough times with randomly chosen arguments. Including symbolic execution within the concolic trace significantly slows down the speed of the test execution and therefore limits the number of traces (random sequences of function calls) that can be explored in a given amount of time. However, it is not clear whether these randomly injected bugs resemble in any way the types of bugs commonly introduced by programmers.

We could certainly hand-inject bugs that symbolic execution would be able to find easily that concrete testing would have a lot of trouble finding. As a simple (and contrived) example of a best-case use for our concolic trace approach, consider an implementation of a bag, `DynamicBag`, that leverages an array implementation (`ArrayBag`) when the number of elements in the bag is relatively small, but switches to using a red-black tree once the bag has passed a certain threshold (e.g. 256 elements). Unbeknownst to the implementers of `DynamicBag`, the tree implementation they chose to use has the unusual feature of always returning `true` to the query `member(33550336)`. Consequently, if `33550336` has not actually been added to the bag and the bag is of size 256 or greater, `DynamicBag` and `ArrayBag` will return a different value to this query. This divergence in behavior is difficult to find using standard techniques. Because this bug cannot manifest until the bag has 256 elements, path explosion will most likely prevent pure symbolic execution from exploring a function call trace that is long enough to build a bag of the requisite size. Concrete execution can easily accomplish this, but on the other hand it is highly unlikely that a call to `member` with the argument `33550336` will be randomly chosen during a purely concrete test. Our technique has a much better chance of finding this hard-to-trigger bug by using concrete execution to build a large bag and then symbolically exploring to comprensively check if there is *any* operation from that state that can lead to a functional difference between the two implementations.

Although this best-case bug is clearly highly contrived, the simplicity of data structure APIs limits the types of bugs that will naturally appear in data structure implementations. Perhaps in more complex APIs and implementations, use cases for BLT's approach will be more prevalent. Our results indicate that the choice of injected faults and example API tested in our evaluation were both insufficient and likely poorly suited to demonstrate the full capacity of BLT to detect bugs; to fully evaluate BLT, more complex test cases and faults should be examined.

2) Comparison: Customized Harness: The custom harness, as expected, executed its tests much more quickly than did the concolic traces, which were run through KLEE and included symbolic execution. However, concolic outperformed the harness, detecting 48 mutants that the harness missed, compared to the 9 that the harness detected that concolic missed.

31 out of the 48 mutants caught by concolic and missed by the harness were caught after the symbolic execution call in the concolic trace. The majority of these were also detected by concrete swarm testing (2 were not). Again, this supports the notion that the symbolic node in the concolic trace is useful inasmuch as it allows our concolic traces, with only 10 different traces, to explore nearly as many paths as did concrete swarm testing; in comparison, the harness, customized to explore only specific "interesting" paths (as determined by the programmer), fails to cover as many different execution sequences. This demonstrates how the randomness of swarm testing creates the largest benefits over a customized harness, at least for these types of bugs.

B. Usability Evaluation

In order to assess the usability of BLT as a general bug-finding tool, we wrote a custom harness (described above and in Fig. 9) that might traditionally be coded by a programmer to test the red-black tree implementation. We compare (1) the effort required to use BLT to that of writing a custom harness, and (2) the debugging information provided by BLT to the information gained from running the harness.

1) Ease of Use: To use BLT in the place of a custom harness, a programmer is required, at minimum, to perform the following steps:

1. Install KLEE, Python 2.7
2. Write a JSON file as specified in Fig. 7
3. Run: `python blt.py --trace [json_file]`

The specification of the JSON in Fig. 7 uses BLT's default settings (i.e. default traces and argument generators). It can be easily extended to run traces customized by the programmer by adding a `traces` attribute to the JSON as in Fig. 8.

```
{
  "class1" : "RBTreeBag",
  "class2" : "ArrayBag",
  "header_files" : ["ArrayBag.hpp", "RBTreeBag.hpp"],
  "source_files" : ["ArrayBag.cpp", "RBTreeBag.cpp"],
  "funcs": [
    {"name":"insert", "args":["int"], "return":"void"},
    {"name":"remove", "args":["int"], "return":"void"},
    {"name":"member", "args":["int"], "return":"bool"}
  ]
}
```

Fig. 7. JSON File used for testing bag implementations

```
"traces": [
  { "symbolic_trace":"false", "symbolic_args":"false",
    "len":30, "funcs":["insert"]
  },
  { "symbolic_trace":"true", "symbolic_args":"true",
    "len":3, "funcs":["remove", "get_member"]}
]
```

Fig. 8. Example of customized traces

```
RBTreeBag *bag;
multiset<int> myset;

void insert() {
  int randNum = rand() % 100;
  myset.insert(randNum);
  bag->insert(randNum);
}

void remove() {
  int randNum = rand() % 100;
  unsigned count = myset.count(randNum);
  if (count > 0) {
    myset.erase(myset.find(randNum));
    bag->remove(randNum);
  }
  else
    bag->remove(randNum);
}

void member() {
  int randNum = rand() % 100;
  unsigned count = myset.count(randNum);
  assert(count == 0 || bag->member(randNum));
}

int main(int argc, char **argv) {
  int steps = NUM_STEPS;
  bag = new RBTreeBag;

  while (steps--) {
    int op = rand() % 100;
    if (op < INSERT)
      insert();
    else if (op < INSERT + REMOVE)
      remove();
    else
      member();
  }
```

Fig. 9. Customized Test Harness for Bag Implementations

Writing the JSON file compared to constructing the test harness (Fig. 9) clearly demonstrates the lack of effort required on the part of the programmer to use BLT: the programmer does not have to reason about the working of her program, as she does when constructing a customized test harness, but rather simply lists out for BLT the files and function signatures of her code. Our results in Fig. 5 and Fig. 6 demonstrate that the default traces perform equally as well (if not better) than a customized harness on these types of bugs, and therefore the effort required to specify traces is not necessary if the programmer merely wishes to do as well as a custom testing harness.

2) Debugging Information: BLT provides information about the traces and sequences of events that caused a failure in the form of replay files, one for each failing trace. These replay files are C++ files with the trace and the trace arguments hard-coded, such that running this replay file is running a deterministic replay of the failing trace. Therefore, the programmer can inspect the replay file code to determine which arguments and sequences of function calls caused the failure.

The following commands can be used to run replay files:

1. To run a specific replay file, run

```
python blt.py --replay [replay_file] [json_file]
```

2. To run all the replay files, run

```
python blt.py --replay_all [replay_dir] [json_file]
```

Command 1 (replay) provides useful information about specific failing instances. For example, when the programmer wishes to test whether particular changes to argument values or removal of specific events (function calls) within a trace leads to failure. Command 2 (replay-all) provides a useful way to test whether a change made to the program code has resolved all previously failing traces. Command 2 essentially provides a set of test cases to ensure that the change to the program code (most likely to fix the bug causing failing traces) has indeed fixed the failure-inducing bug, and these replay files can act as a regression test for any further changes to the program as a test for determining that the bug has not been introduced.

In contrast, the customized harness, when detecting an error, returns an assertion error to the programmer. In order to see why the failure occurred, the programmer would have to record the event and arguments used in the failing trace (which BLT does automatically). Common techniques to provide debugging information require editing the source code — for example, adding `printf` statements — or using tools such as `gdb` to see what arguments are called. In addition, the programmer is not assured of a deterministic replay with which to debug an assertion error if randomness is included in the testing harness.

Although BLT has not been used by programmers in the wild, it seems clear that BLT's ability to provide useful debugging information and ease of use is significantly greater than that of a simple customized testing harness. Further use cases including programmers unfamiliar with BLT's implementation and usage should be evaluated in order to provide support for this claim.

VI. Related Work

Substantial research has been done on how to generate test inputs that explore interesting execution paths in order to discover unexpected program behavior. One of the most commonly used techniques is random testing with concrete inputs, using methodologies such as fuzz testing [4], [5] and swarm testing [1]. Another common technique has been modern symbolic execution, implemented using a concolic approach or, as in the case with KLEE, an execution-generated testing approach [6]. Further work has been done in directing symbolic execution to explore paths that relate to particularly relevant portions of code. Examples include shadow symbolic execution, a technique that drives execution toward the behavior that has changed from one version of a program to the next [7]; differential symbolic execution, which uses a concept of "symbolic summaries" to compute a precise behavioral characterization of a program change [8]; and demand-driven compositional symbolic execution, which consists of composing symbolic executions of feasible intraprocedural paths while also symbolically executing as few intraprocedural paths as possible by directing execution toward a specific target branch or statement of interest [9]. These examples of "longitudinal" symbolic execution identify statements that differ between one version of a program and the next with heuristics, static analysis, and dynamic analysis.

To overcome some of the limitations of pure symbolic execution, recent symbolic execution engines intermix concrete and symbolic execution in the form of concolic or execution-generated testing approaches. For example, DART, which uses concolic execution, performs symbolic execution dynamically while the program is executed on some concrete input values [2], and KLEE, which uses execution-generated testing, intermixes concrete and symbolic execution by dynamically checking before every operation if the values involved are all concrete, and if so, executing the operation concretely [2]. BLT treats concrete and symbolic execution separately, first exploring very specific paths with concrete traces, and then performing a short symbolic trace (leveraging KLEE) to explore comprehensively all paths possible at the fringes of the concrete traces. Thus, BLT's mix of concrete and symbolic does not alter the symbolic execution itself, but rather uses concrete execution as a way to target particular areas of interest to explore symbolically, areas that symbolic execution alone would be unable to reach within a reasonable amount of time.

Tools for determining semantic equivalence of functions have also been developed. For example, regression verification [10], as well as a language-agnostic tool, SymDiff [11], use SMT solvers to solve verification conditions generated from the modular (individual) comparison of procedures within a program. Similarly, UC-KLEE [12] leverages KLEE to check the equivalence of two arbitrary C functions. These approaches are orthogonal to the approach of BLT, which checks for both 1) equivalent behavior across two different versions of a function run in isolation, and 2) equivalent behavior of function traces across two different implementations. Thus, a tool such as SymDiff can be used along with BLT to provide a stronger guarantee of functional equivalence.

VII. Conclusion

BLT is a tool that combines concrete swarm testing with symbolic execution to test the functional equivalence of two implementations of the same C++ API. It uses swarm testing to efficiently create an interesting program state and then explores the logic of this state comprehensively through symbolic execution.

From our evaluation, we drew the following conclusions: **(1)** Concrete random swarm testing, at least for this category of injected bugs, appears to be the best bug-detection technique: running 100 concrete traces with random swarms caught the greatest number of mutants compared to 10 concolic traces with random swarms and a custom harness that uses randomness to choose a function call, rather than to choose a random swarm of functions to call; furthermore, the 10 concolic traces with random swarms also significantly outperformed the custom harness. **(2)** Although we encountered some mutants that appeared to be more easily detected with concolic traces than with concrete ones, it is not at all clear whether the success of these traces can be directly attributable to the inclusion of a symbolic node. **(3)** Whether testing with concrete swarms or with concolic traces, in both cases a general technique that utilized random swarms outperformed a harness customized to the programmers' exact implementation. Therefore, in testing for these types of injected bugs, we found that random swarms are much more powerful than we had expected and produce impressive results.

Technically, BLT can be used as a tool to implement concrete swarm testing; however, the full power of BLT is in its ability to combine concrete and symbolic execution. BLT's full capabilities were shown to be unnecessary for the purpose of detecting the bugs that were injected at random into the red-black tree implementation. Nevertheless, there plausibly still exist faults too complex to be randomly injected by a Python script that can be detected with BLT's concolic traces, but not by concrete swarm testing. In order to discover if such faults exist, BLT requires more extensive testing on more complex code, such as that of a web server; to do so, the functionality of BLT must be extended and several limitations overcome.

VIII. Acknowledgements

We would like to acknowledge Eddie Kohler for his constant support and guidance throughout the development of this project.

References

[1] E. E. Y. C. J. R. Alex Groce, Chaoqiang Zhang, "Swarm testing," *Proceedings of the 2012 International Symposium on Software Testing and Analysis, Minneapolis, MN, USA*, Jul. 2012.

[2] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[3] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10. New York, NY, USA: ACM, 2002, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/587051.587053

[4] M. W. James H. Andrews, Alex Groce and R.-G. Xu, "Random test run length and effectiveness," *Proc. ASE*, pp. 19–28, Sep. 2008.

[5] P. G. Saswat Anand and N. Tillmann, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 23–32, Nov. 2011.

[6] D. E. Cristian Cadar, Daniel Dunbar, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," *Proc. of the Eighth Symposium on Operating Systems Design and Implementation (OSDI 08)*, pp. 209–224, Dec. 2008.

[7] C. Cadar and H. Palikareva, "Shadow symbolic execution for better testing of evolving software," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 432–435. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591104

[8] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 226–237. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453131

[9] P. G. Saswat Anand and N. Tillmann, "Demand-driven compositional symbolic execution," *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008).*, 2008.

[10] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Partition-based regression verification," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 302–311. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486829

[11] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *Proceedings of the 24th International Conference on Computer Aided Verification*, ser. CAV'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 712–717. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31424-7_54

[12] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," *Proceedings of the 23rd international conference on Computer aided verification, Snowbird, UT*, pp. 669–685, Jul. 2001.