Efficient Load Test Generation

Abstract

A crucial part of any software testing regimen is *performance testing*: a program must be put under load, *e.g.,* by executing a critical loop of the program many times, to ensure that the program behaves as expected at large scales. Unfortunately, generating precise performance tests, where a specific part of the program can be exercised at a controllable load level, is challenging. The standard approach to generating performance tests using dynamic symbolic execution incurs too much overhead to effectively generate large-scale test inputs. We propose LANCET, a dynamic symbolic execution–based tool that sidesteps the overheads. LANCET has two key innovations: (i) a path exploration heuristic for dynamic symbolic execution that more effectively generates scaling inputs for loops in complex programs; and (ii) a novel *constraint inference* method, that uses statistical inference on small-scale inputs to automatically infer constraints that describe large-scale inputs, allowing programmers to generate large inputs without performing symbolic execution at large scales. We demonstrate the effectiveness of LANCET by using it to infer large-scale, targeted inputs for synthetic benchmarks and real-world distributed applications such as Memached and Redis.

1. Introduction

Writing correct and performant software running at scale has always been a challenge, especially so given the rising popularity of multithreaded and distributed systems. Scalability turns out to be the Achilles heel of many, otherwise well-designed software systems, which suffer from correctness or performance problems when executed at a large scale. For example, Foursquare, a geosocial network for sharing experiences of places, had an unprecedented 17 hours of downtime because the data stored in one of its two MongoDB shards reached the hosting computer's RAM capacity [14]. This is an example of a scaling problem with the data size. As another example, the Hadoop Distributed File System (HDFS) runs into performance problems when the number of clients becomes too large, relative to the processing power of the namespace server [20]. This is an example of a scaling problem that is triggered by a large number of nodes, and indirectly, by large sizes of data.

Bugs often happen out of the frequent paths of a program, rather in places where less attention has been paid to in the development process. As a remedy, unit tests are often introduced to cover both hot and cold paths and check for errors in the runtime. An important quality criterion for a unit test suite is code coverage, or how much portion of the entire code base of the system under test (SUT) is touched by a test suite. By the conventional definition of code coverage, a line of code is considered covered if it is executed for at least once by a test. Such definition of code coverage is fraudulent in two means. First, it is purely a control flow concept and therefore completely ignorant to the data flow. For example, a null pointer dereference error may never be triggered by a unit test that exercises the line of code but with a valid pointer. Furthermore, running every part of SUT for once is not sufficient to reveal many performance issues. For example, there was a scalability issue in a LRU cache implementation of MySQL [1], which can only be detected by running SQL SELECT in a loop for a large number of times.

Performance testing, as one kind of system testing that performs end-to-end black-box testing, is designed to find performance issues in software. The idea in its essence is to run SUT through a large input, measure the sustainable performance and observe if any failure is triggered in the runtime. For simple programs, it is obvious that a larger input will invoke a longer execution. For example a string copy takes more time for a longer input string. However, it requires years of practice before one can master the black art of finding "large" inputs for complex real-world software systems. Sheer increasing the amount of data contained in the input does not necessarily lead to a longer execution, depending on how the program processes the input data. Take the classic binary search algorithm for example, its run time grows as a logarithmic function of the input size. To double the run time of binary search, the input must be an order of magnitude larger in size. On the other hand, more data means more logs, which are more difficult to analyze if a bug does occur. Numerous research projects invested in log analysis [18] have proved the task a difficult one.

It is only natural to ponder if it is possible to achieve a long execution or a load spike with a reasonably sized input for SUT, or how to push an execution to the limit with inputs of a certain size. Unfortunately, there exists no systematic method to the best of our knowledge that leads to a performance test that induces a significant level of workload on SUT, not to mention striking a balance between execution length and input size. QA engineers often need to understand the internals and interfaces of complex software systems that consist of a pile of distributed components and millions of lines of code written by other developers, then endure a lengthy trial-and-error session to find a performance test of good quality. If we take a closer look, the QA workflow consists of the following steps: (a) making the test plan, i.e. starting with simple tests focused on individual code paths, followed by combination of different paths to simulate real user behaviors; (b) understanding the relevant code path and every other part of SUT that interacts with it to get sufficient knowledge for creating the performance tests; (c) developing and running the performance tests, while measuring the resultant performance using profiling tools. Out of these steps, (a) defines the policy, or the goal for the performance tests, for which human effort is indispensable, while (b) and (c) provide the mechanisms that implement and verify the resultant performance tests, which also form the most laborious part of the process for QA engineers.

We introduce LANCET, a performance test generation tool, to address the pain in the search process of performance tests by automating the latter two steps in the QA workflow where computers can be more efficient than human beings. LANCET is built on top of symbolic execution and statistical prediction, providing a tool with: (a) low entry barriers, i.e. little knowledge of SUT internals is required to create good performance tests, (b) systematic algorithms to reason the performance impact of values in the input, (c) integrated statistical models to extrapolate the scaling trend of SUT, (d) support for widely used concurrent and event-driven programming libraries, (e) on-the-fly SUT instrumentation for measurement and verification of generated performance tests. By default, LANCET inspects every loop in a program and tries to find inputs to run each loop for the number of iterations specified by the user. In the case where the user is familiar with the implementation of SUT, he can designate a set of target loops or even a single loop to reduce the scope of code considered by LANCET and speed up the test generation process. The reason for choosing loops is twofold. First, programs spend a large part of their execution time in loops. Second, we observe, through examination of applications that have had doc-

umented scalability problems [**?**], as the application is run at larger scales, the negative impact of inefficient or incorrect code inside a loop often gets amplified into severe performance regressions or cascading failures.

When applying LANCET to an application, the user can specify the loop he wants to test and the load level, *i.e.,* the number of iterations, for the loop. LANCET takes these parameters from the user as guidance to steer its symbolic execution engine and finds the inputs that reach the given number of iterations for the given loop. LANCET makes two changes to the state-of-the-art symbolic execution algorithm. First, to reach a specific number of iterations after entering the target loop, LANCET uses a loop-centric search heuristic that favors the paths that stay inside the loop over those that exit. This path prioritization strategy is in contrast to existing path strategies that favor finding new paths. Second, LANCET shortcuts the path exploration by applying statistical inference to derive the path constraints at a large number of iteration from training samples, which consist of path constraints from running the loop a small number of iterations. This way, to reach N iterations of a given loop, LANCET just needs to execute the loop for $1 \ldots M$ iterations where $M \ll N$ and collect the constraints at each iteration. Afterwards, the constraint solver is invoked for the extrapolated path constraints to get the input that would trigger N iterations of the target loop.

We apply LANCET to four disparate programs: a linear algebra code, `mvm`, a quantum computer simulator from SPEC, `libquantum`, a fluid dynamics program, `lbm`, and a utility from the GNU Coreutils suite, `wc`. Each of these programs needs to scale up to satisfy common use cases: `mvm` to tackle large matrices, `lq` to factorize large numbers, `lbm` to simulate flows for more timesteps, and `wc` to process large text corpuses. We show that for these programs, LANCET is able to generate more, and larger-scaling, inputs than a state-of-the-art test generation tool when given the same amount of computation time, and that in all cases, LANCET can generate even larger inputs through the use of statistical inference. Moreover, because of the regularity of the constraints that LANCET performs inference over, it is able to make its predictions perfectly, allowing programmers to correctly generate inputs of any size without incurring the cost of additional symbolic execution.

Outline Section 2 provides background on symbolic execution. Section 3 describes the basic design of LANCET, using a simple matrix-vector multiplication program as an example. Section 4 sketches the implementation of LANCET. Section 5 presents case studies of using LANCET to generate large-scale performance tests for several real-world applications. Section 6 discusses related work, and Section 7 concludes.

2. Background: Dynamic Symbolic Execution

Many state-of-the-art white-box test-generation tools rely on *dynamic symbolic execution* [9, 10, 12]. This section provides a brief overview of how these tools work, and discusses some of the shortcomings of current techniques that LANCET aims to ameliorate.

2.1 Dynamic Symbolic Execution Basics

Dynamic symbolic execution couples the traditional strengths of symbolic analysis—tracking constraints on variables to determine the feasibility of various program behaviors—with the efficiency of regular concrete execution. The inputs to a program are treated as symbolic variables. As the program executes, statements that involve symbolic variables are executed symbolically, adding constraints on symbolic variables. When a branch statement is reached (*e.g.* `if (x < y) goto L`), *one* of the paths is taken, and the appropriate constraint is added to the *path condition*, or set of constraints (*e.g.*, if the branch above is taken, the constraint $(x < y)$ would be added to the path condition). As new constraints are added to the path condition an SMT solver (Satisfiability Modulo Theory) is invoked to ensure that the path condition is still satisfiable; unsatisfiable constraints imply that the particular path that execution has taken is infeasible—no program execution could follow that path. Once a path through the program is found, the SMT solver is used to produce a *concrete* set of values for all the symbolic variables. These concrete values constitute an input. Note that unlike traditional symbolic execution, dynamic symbolic analysis can always fall back to concrete values for variables; if execution encounters a statement that cannot be analyzed using the underlying SMT solver, calling an external function for example, the variables involved can be concretized and the statement can be executed normally. What it gives up is completeness of the code coverage as a result of the concretization.

2.2 Path Exploration Heuristics

One of the key decisions in dynamic symbolic execution tool is how to choose the path through a program an execution should explore. In other words, how to choose directions for the branches encountered in execution. For example, a tool such as KLEE [9] executes multiple paths concurrently, in an attempt to generate a series of inputs to exercise different paths of the program. When encountering the branch statement described above, KLEE can fork the execution to two clones that follow the two branches respectively, choose one execution clone to explore first and save the other for later. The resultant paths would include the constraint $(x < y)$ in the clone that follows the true branch, and $\neg(x < y)$ in the one that follows the false branch.

There are numerous heuristics that can be used to choose which path to explore first at each branch in dynamic symbolic execution; the choice of the correct heuristic depends on the goals of the particular tool being developed. We abstract away the goal of a dynamic symbolic execution tool by describing it in terms of a *meta-constraint*. A meta-constraint is a higher level constraint that the path condition describing a particular execution attempts to satisfy. For example, in the case of generating high-coverage tests, the meta-constraint for a tool may be to produce a path that exercises program statements not seen by previously-generated inputs, in which case a path-exploration heuristic might prioritize flipping branches to generate a never-before-seen set of constraints. In the case of generating stress tests, as in LANCET, the meta-constraint would be to generate a path condition that exercises a particular loop body a certain, user-defined number of times, in which case the path-exploration heuristic might prioritize taking branches that cause the loop to execute again, till the user-defined limit is reached (Section 3.2.2). Note that just as a path condition may not be tight—there can be many possible concrete inputs that follow a particular path through the execution—a meta-constraint need not be tight: multiple path conditions may all satisfy a given meta-constraint.

2.3 Dynamic Symbolic Execution Overhead

The primary drawback of dynamic symbolic execution is its dependence on an underlying SMT solver to manage the path condition that an execution generates. At every branch, the SMT solver must be invoked to determine whether a particular choice for a branch is feasible or infeasible. Though the constraint solver is also invoked to concretize input variables when necessary, or to generate the final concrete input for a particular run, the majority of queries to the solver arise from these feasibility checks [10]. Though there are many techniques to reduce the expense of queries to the constraint solver, such as reducing the query size by removing irrelevant constraints and reusing the results of prior queries whenever possible [9], the fundamental expense of invoking the constraint solver at each branch in an execution remains. This overhead can lead to an order-of-magnitude slowdown in execution [9].

A second overhead of dynamic symbolic execution is the path-explosion problem. The path exploration heuristics of a particular tool may prioritize certain choices for branches in an attempt to satisfy the tool's meta-constraint. However, the exploration heuristics may be wrong: whenever a branch is encountered, if the heuristic makes the wrong choice, the meta-constraint may not be satisfiable, and the tool must return to the branch to make a different choice.

These two overheads make generating large-scale, long-running inputs using dynamic symbolic execution difficult. First, as the path being explored gets longer, more branches are encountered, resulting in more invocations to the SMT solver. Second, long-running inputs execute more branches, creating more opportunities for the path-exploration heuristic to "guess wrong," leading to unsatisfiable meta-constraints and ultimately leading to path explosion.

2.4 WISE

WISE is perhaps the most closely-related dynamic symbolic execution technique to LANCET. WISE attempts to generate worst-case inputs for programs (*e.g.*, a worst-case input for a binary-search-tree construction program would be a sorted list of integers, generating an unbalanced tree) [8]. In meta-constraint terms, WISE attempts to generate a path condition that produces worst-case inputs of a reasonably large size.

While it is possible to exhaustively search through all possible path conditions for a certain input size to find the worst-case inputs, it is clear that this approach will lead to path explosion when applied to larger inputs. To avoid the path explosion problem, WISE uses the following strategy. It exhaustively searches the input space for small input sizes to find worst-case behaviors. Then, by performing pattern matching on the worst-case path conditions generated for different input sizes, WISE learns what choices worst-case path conditions typically make for branches in the program (*e.g.*, always following the left child to add a new element in a BST). This pattern is then integrated into a path-exploration heuristic.

At large scales, when WISE encounters a branch, it looks through the patterns that it identified at small scales to choose the direction for the branch. In essence, WISE has a notion of what a worst-case path condition "looks like," and chooses directions for branches to make the path condition for a larger input match that worst-case pattern. By using this new path-exploration heuristic, WISE is able to find worst-case inputs for larger scales without exploring every possible path. In many cases, WISE need only explore a single path through the program to find a worst-case input!

WISE addresses the overheads of dynamic symbolic execution by tackling the second drawback described above: it controls path explosion by learning the pattern of path constraints from smaller runs where path explosion is less of an issue. Nevertheless, WISE still sits on top of dynamic symbolic execution, and must query the SMT solver at every branch; it does not tackle the first source of overhead. As a result, even though WISE has much better scalability than naïve symbolic execution, it still cannot generate particularly large inputs. Generating an input that runs a loop one million times still requires performing symbolic execution on a run that visits the loop test condition one million times.

Section 3.3 explains how LANCET tackles this problem. In particular, LANCET can generate large-scale inputs for a program *without ever performing symbolic execution at large scales*.

3. Design

LANCET is a dynamic symbolic execution tool that aims to generate *scaling inputs* for programs: inputs that cause programs to run for a long time. In contrast to black-box stress-generation tests, LANCET targets particular loops for scaling. It attempts to generate inputs that will cause a chosen loop to execute a specified (large) number of iterations. In this way, particular regions of code can be targeted to see how they behave under heavy load. This section first describes LANCET's behavior at a high level, and then explains LANCET's various components in more detail. For ease of exposition, this section assumes that the program under test takes a single symbolic string as input.

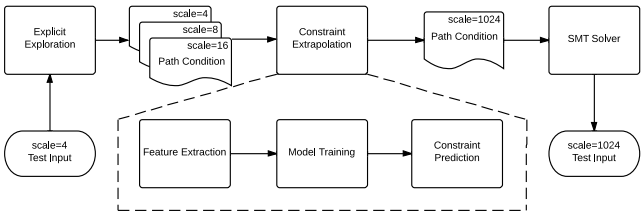

Figure 1. High level flow of LANCET's inference-mode approach for generating inputs for a given loop.

3.1 Overview of LANCET

LANCET has two modes of operation: *explicit* mode, which uses traditional dynamic symbolic execution techniques to generate inputs that run a target loop for a specified, small number of iterations, and *inference* mode, which uses statistical techniques to generate inputs that run a target loop for a large number of iterations. These modes are described in more detail in Sections 3.2 and 3.3, respectively. At a high level, they behave as described next.

LANCET's explicit mode begins by using programmer annotations to identify the target loops (Section 3.2.1). For each target loop, LANCET then generates a path condition that satisfy a *loop-iteration meta-constraint* that the target loop executes exactly N times. For small N, this is done by a custom *loop-centric* path-exploration heuristic (Section 3.2.2).

While the explicit mode suffice to generate path constraints that run a loop a small number of times, it is impractical for large N: *e.g.,* running a loop a thousand times will require two orders of magnitude more invocations to LANCET's constraint solver than running a loop ten times, and will be unacceptably slow. LANCET's novel approach to this problem is to determine the path constraints that satisfy the loop-iteration meta-constraint for a large N *without performing symbolic execution*.

LANCET's inference mode operates as shown in Figure 1. First, LANCET uses its explicit mode to generate *multiple* path conditions, for a pair of consecutive numbers of iterations M and $M+1$, using a symbolic input string of L bytes.[1] It then looks into pairs of path conditions for different numbers of iterations and identifies the *incremental set*, the set of path constraints that *only* exist in the $M+1$-iteration path conditions. LANCET then extrapolates the path condition of N iterations by appending $N-M$ copies of the incremental set to the M-iteration path condition, each projected to appropriate offsets beyond the initial L bytes of the input string using a linear regression model (Section 3.3.2). Finally, the predicted N-iteration path condition is solved with a constraint solver to generate a large-scale input for the program (Section 3.3.3), which is then verified in real execution of the program. Depending on the result of verification, LANCET may restart the explicit mode to generate more training data and refine the large-scale input based on new path conditions discovered in the explicit mode.

Crucially, once the initial training phase of LANCET is complete, inputs that target *any scale* can be generated at the same overhead (though potentially different levels of accuracy), making LANCET a truly scalable approach to generating large-scale, stress-test inputs.

[1] Multiple path conditions may result in the same number of iterations.

Running example To aid in the discussion of the components and operation of LANCET, we will use a running example of request parsing from Memcached [2]. A few salient points: (i) the code parses the request string that contains a *get* command followed by a list of keys separated by one or more spaces; (ii) the first function `tokenize_command` splits the command string (a symbolic string of configurable size received from the Socket layer of LANCET, see Section 4.2) into a list of tokens, stores them as a list of tokens terminated by a length 0 token, then returns the number of tokens retrieved; (iii) the second function `parse_get_command` parses the list of tokens in the target loop (line 29) and executes the *get* command for each key contained in `tokens`. (iv) the number of iterations executed by the target loop is determined by the loop in function `tokenize_command`. (v) in the following references to this example, we suppose the length of the `command` string is 8 bytes for the ease of exposition.

3.2 Explicit Mode

The goal of LANCET is to find performance tests that impose a certain load level precisely on a certain part of code in the given program. Specifically, LANCET's test generation is designed for loops and therefore the load level is determined by the trip count of a loop.

For a given trip count N and a target loop l, LANCET's explicit mode uses symbolic execution to generate an execution path that satisfies the meta-constraint that the path executes loop l exactly N times. The symbolic execution engine treats the input as a bitvector of symbolic variables, computes symbolic expressions for input-dependent variables, and accumulates the constraints at every branch to form the set of constraints, *i.e.,* the path condition, that must hold when the path is followed in an execution. LANCET obtains a test input for the program that will run l for N iterations by calling an external SMT solver to find concrete values that satisfy the path condition.

3.2.1 Targeting a loop

LANCET provides a simple yet powerful interface for user to specify which loops she wants to target using source code annotation. To mark a loop as a target for test generation, the attribute `[[loop_target]]` needs to be inserted right before the loop statement. In the running example, the loop at line 29 is targeted for test generation.

3.2.2 Loop-centric search heuristic

The powerful multi-path analysis enabled by symbolic execution comes with a price: the path explosion problem. In order to get meaningful results within a reasonable time frame, any symbolic execution tool must steer through the exponentially growing number of paths and prioritize the exploration of the more interesting ones. For example, as demonstrated by KLEE [9], path searching heuristics like random path selection and coverage-optimized search are effective for generating high-coverage tests for complex programs (like GNU COREUTILS). However, these heuristics, though good for discovering unexplored code, are ill-suited for the purpose of generating performance tests, because rather than exercising every line of code once, as a functional test suite might, a performance test should instead repeatedly execute critical pieces of code to simulate high loads.

LANCET employs a loop-centric heuristic to guide the search for paths that extend the target loop for a large number of iterations. Following many existing symbolic execution tools, LANCET encapsulates runtime execution information such as program counter, path condition, memory content in a symbolic process. The loop-centric search operates in two modes, the *explorer* mode and the *roller* mode.

In explorer mode, LANCET starts the execution with a single symbolic process from program entry, forking a new process at each branch that has a satisfiable path condition (this is the default execution mode for KLEE). If the loop header of the target loop, l is hit by any of these symbolic processes, that process enters roller mode and the other explorer processes are paused. Roller mode prioritizes symbolic processes that stay inside the target loop (*e.g.*, taking loop back edges to avoid exiting the loop) so that it can reach a high number of iterations more quickly.

Roller mode maintains a FIFO queue for all symbolic processes whose current program counters are inside the target loop and schedules the next process from the head of the queue whenever the queue is not empty. Each symbolic process tracks how many times it has executed the target loop. LANCET counts the number of times the loop has run *in the current calling context* (*i.e.*, the loop trip count is reset if the function is exited). This policy means that in nested loops, inner loops cumulatively count iterations across all iterations of any outer loop. If a symbolic process has executed exactly N iterations, roller mode attempts to exit the loop, yielding a path constraint for an input that will run the loop exactly N times. The explorer mode is agnostic to the search strategy and any effective code discovery strategy could be leveraged by LANCET for identifying a path from the input to the target loop.

Example: In explorer mode, LANCET will spawn symbolic processes that try every possible path through the program in Figure 2. Because `process_get_command` is called only for *get* requests where the `command` string starts with 'get␣', every process that reaches the target loop at line 29 would include the following constraints:

$$command[0] =' g' \wedge$$
$$command[1] =' e' \wedge$$
$$command[2] =' t' \wedge$$
$$command[3] =' \text{␣}'$$

A process that executes the target loop for one iteration will end up with the following additional constraints:

$$command[4] \neq' \text{␣}' \wedge$$
$$command[5] \neq' \text{␣}' \wedge$$
$$command[6] \neq' \text{␣}' \wedge$$
$$command[7] \neq' \text{␣}'$$

Another process that executes the target loop for two iterations will accumulate constraints as follows:

$$command[4] \neq' \text{␣}' \wedge$$
$$command[5] =' \text{␣}' \wedge$$
$$command[6] \neq' \text{␣}' \wedge$$
$$command[7] \neq' \text{␣}'$$

A direct comparison between the constraints of 1-iteration and 2-iteration processes would reveal that, omitting the case of consecutive spaces, the number of times the condition at line 8 is true is determined by the number of tokens the string contains, thereby the number of iterations the target loop executes. This observation will lead to our key insight for the inference mode.

3.3 Inference Mode

A strawman approach to performance test generation would use LANCET's explicit mode exclusively to generate large-scale inputs, targeting loop l to run N times for some large N. This approach could generate tests that accurately trigger the target loop for N times if given indefinite amount of time. However, nontrivial loops

```
1  size_t tokenize_command(char *command, token_t *tokens, size_t max_tokens) {
2      char *s, *e;
3      size_t ntokens = 0;
4      size_t len = strlen(command);
5      unsigned int i = 0;
6      s = e = command;
7      for (i = 0; i < len; i++) {
8          if (*e == ' ') {
9              if (s != e) {
10                 /* add a new token into tokens */
11                 ntokens++;
12                 if (ntokens == max_tokens - 1) { e++; s = e; break; }
13             }
14             s = e + 1;
15         }
16         e++;
17     }
18     if (s != e) {
19         /* add the last token into tokens */
20         ntokens++;
21     }
22     /* add a terminal token of length 0 into tokens */
23     ntokens++;
24     return ntokens;
25 }
26
27 void process_get_command(token_t *tokens, size_t ntokens) {
28     token_t *key_token = &tokens[KEY_TOKEN]; /* KEY_TOKEN is the offset to the first key */
29     [[loop_target]] while(key_token->length != 0) {
30         /* retrieve the key from cache */
31         key_token++;
32     }
33 }
```

Figure 2. Running example: request parsing in Memcached.

that contain complex control flow structure may cause the path explosion problem after a large number of iterations even if LANCET only considers the code enclosed by these loops. Secondly, the symbolic execution engine needs to consult with the constraint solver at every branch instruction to determine if the current path condition is satisfiable. In a state-of-the-art symbolic execution tool, more than half of the time is spent by the constraint solver [9]. It is simply impractical to run a symbolic execution engine for more than a handful of iterations of the target loop.

Since our goal is not to verify every possible execution path, but merely to generate a large-scale input, it is unnecessary and wasteful to execute every iteration of the target loop through the symbolic execution engine. LANCET's inference mode takes a more efficient approach that skips symbolic execution of these intermediate iterations and simply generates the path condition for the Nth iteration. In further detail, the training of LANCET's interference is done for various small scale inputs that execute the target loop up to M times, $M << N$, and then skips executing the loop between M and N times.

Recall the running example where the number of iterations of the target loop is determined by the number of times the true branch is taken at line 8. This observation leads to our key insight that for many loops, there is a statistical correlation between the desired trip count for a loop and the number of constraints generated by a set of critical branches, and this correlation can be used for inference of the path condition for the Nth iteration. In its essence, a path condition is just a document that contains a set of constraints represented by strings. However, it is difficult and inaccurate to generate them directly from the inference model using general text mining techniques if we treat a set of constraints as an unstructured document. LANCET first extracts features from the path conditions based on the structural properties of path condition, and trains a regression model to capture the correlation between the trip count of the target loop and each feature of the path conditions using the data from small-scale training runs. The structural features of the N-iteration path condition are then predicted using the regression models and the N-iteration path condition is generated based on the predicted features. Finally, LANCET solves the N-iteration path condition to obtain a concrete input using a SMT solver, and verifies the input in real execution. In case the input verification fails, LANCET switches back to the explicit mode to generate more training data before running the inference mode again. We will present each of these steps of the inference mode in the following sections.

3.3.1 Extracting features from path conditions

LANCET transforms path conditions into constraint templates and numerical vectors, which are then used to train the statistical models LANCET builds to capture the relationship between the trip count of a loop and the resultant path condition. As a preprocessing step, LANCET first puts the constraints of each path condition into groups introduced by the same branch instruction, then sorts each group by the lowest offset of symbolic byte each constraint accesses. Each ordered group of constraints constitute a *feature* in LANCET's inference mode. For a series of path conditions, $\{P_i \mid i \leq M+1\}$, where P_i represents the path condition ensued by i iterations, and each path condition is processed into a set of features $\{P_i^j\}$, where P_i^j represents the jth feature of P_i, LANCET finds the *incremental set* D_{i+1}^j, the residual part of P_{i+1}^j after removing the longest common prefix between P_{i+1}^j and P_i^j. In the running example, the incremental set between the 1-iteration and the 2-iteration path conditions contains the following constraints:

$$command[5] =' \sqcup' \wedge$$
$$command[6] \neq' \sqcup' \wedge$$
$$command[7] \neq' \sqcup'$$

Intuitively, the incremental set starts at the first byte where two path features differ and continues till the end in the feature of the more number of iterations.

LANCET extracts from a incremental set the following information: (a) the set of constraint templates; (b) the offsets of symbolic bytes referenced by each constraint; (c) the values of the concrete numbers in each constraint. The constraint templates can be obtained from a incremental set by replacing offsets of symbolic bytes and concrete numbers in each constraint with abstract terms numbered by their appearances. The sequence of offsets of symbolic bytes and concrete numbers are also recorded in the meantime. For example, the above incremental set from the running example can be abstracted into constraint templates:

$$command[x_1] = x_2 \wedge$$
$$command[x_3] \neq x_4 \wedge$$
$$command[x_5] \neq x_6$$

The corresponding sequence of symbolic variable offsets is 5, 6, 7, and the sequence of concrete numbers 32, 32, 32 (32 is the ASCII code for space).

It is possible to reach the same number of iterations with different path conditions. For example, another path that finishes the target loop for one iteration in the running example may require this condition:

$$command[4] =' \text{␣}' \wedge$$
$$command[5] \neq' \text{␣}' \wedge$$
$$command[6] \neq' \text{␣}' \wedge$$
$$command[7] \neq' \text{␣}'$$

Furthermore, the incremental set between this path condition and the previous 2-iteration one is:

$$command[4] \neq' \text{␣}' \wedge$$
$$command[5] =' \text{␣}' \wedge$$
$$command[6] \neq' \text{␣}' \wedge$$
$$command[7] \neq' \text{␣}'$$

which is longer than the aforementioned incremental set. In light of this case where multiple paths are possible for the same trip count of a loop, LANCET uses the minimal incremental set, which contains the shortest list of constraints.

3.3.2 Inference over path conditions

LANCET infers the next $N - M$ incremental sets based on the constraint templates and the sequences of terms, i.e. symbolic variable offsets or concrete numbers, extracted from the current incremental set for M iterations. It first extrapolates these sequences into the next $N - M$ iterations and then fill $N - M$ copies of the templates with predicted values. Since both kinds of sequences contain numbers, LANCET uses the same algorithm to predict them. For the common case when there are multiple constraints in the current incremental set, LANCET employs a regression model to predict the sequence. For the case when there is a single constraint in the current incremental set, LANCET applies two extrapolation heuristics: (i) repeating the single number in the sequence; (ii) extending the sequence with a series of consecutive numbers.

In our running example, there are 3 constraints in the current incremental set, therefore LANCET will use a regression model to capture the relationship between the trip count and the sequences of terms, and predicts the following incremental set for the 3rd iteration of the target loop:

$$command[8] =' \text{␣}' \wedge$$
$$command[9] \neq' \text{␣}' \wedge$$
$$command[10] \neq' \text{␣}'$$

Note although the size of the symbolic input string is constant due to the lack of support for string length operation in the underlying SMT solver [11], LANCET is able to infer constraints beyond the fixed range of input in its inference mode and generate path condition that references input of arbitrary length.

3.3.3 Generating a large-scale input

Once LANCET predicts a path condition to execute the loop N times, it calls the SMT solver [11] to solve the predicted path condition to generate an appropriate large-scale input. For the generated test inputs, it also verifies the actual number of iterations achieved for the target loop in the runtime. LANCET compiles the program under test into a native x86-64 executable with lightweight instrumentation inserted around the target loop in compile time, to record the number of times the loop header is observed during an execution. If the actual trip count does not reach the set goal, LANCET returns to its explicit mode to explore the loop for more iterations and covers more code paths within the loop body, then feeds the new data into the inference mode to get an improved prediction of the path condition for N iterations. This process repeats until the actual trip count are within a small range of N, which is a configurable option of LANCET.

4. Implementation

This section presents the implementation of LANCET. The symbolic execution engine used in the explicit mode of LANCET is built on top of KLEE, with added support for pthread-based multithreaded programming, libevent-based asynchronous event processing, socket-based network communication and various performance enhancements for the symbolic execution engine. LANCET uses the build system of Cloud9 [6], based on Clang [3], allowing it to compile makefile-based C programs into LLVM bitcode [4] required by the symbolic execution engine. We now discuss the changes made to baseline KLEE.

4.1 POSIX Thread

LANCET supports most of PThread API for thread management, synchronization and thread-specific store (Table 1). A thread is treated the same as a process except that it shares address space and path condition with the parent process that has created it. And since threads are treated as processes, they are scheduled sequentially to execute unless a thread is blocked in a synchronization calls, such as calling `pthread_mutex_lock` on a lock that has been acquired by another thread. When a thread encounters a branch with a symbolic condition, it will forked all threads belonging to the same process, essentially making a copy for the entire process. Both mutex and condition variable are implemented as a wait queue. When the current owner releases the synchronization resource, LANCET will pop a thread from the wait queue to take control of the resource and mark the thread runnable. LANCET applies the *wait morphing* technique to reduce wasted wake-up calls in condition variables. When the condition variable has an associated mutex, `pthread_cond_signal/broadcast` does not wake up the threads, but rather move them from waiting on the condition variable, to waiting on the mutex.

4.2 Socket

Socket API is a network programming interface, provided by Linux and other operating systems, that allows applications to control and use network communication. Server applications, like Apache,

Table 1. LANCET supports most of PThread API for thread management, synchronization and thread-specific store.

Category	API
Thread Management	pthread_create
	pthread_join
	pthread_exit
	pthread_self
Synchronization	pthread_mutex_init
	pthread_mutex_lock
	pthread_mutex_trylock
	pthread_mutex_unlock
	pthread_cond_init
	pthread_cond_signal
	pthread_cond_broadcast
	pthread_cond_wait
Thread Specific Store	pthread_key_create
	pthread_key_delete
	pthread_setspecific
	pthread_getspecific

MySQL and Memcached, consume data received from the network via sockets, therefore cannot be tested in solitude. Previous work [6] that generates tests for Memcached using symbolic execution combines a client program into the server code to address this problem. However, this approach requires deep knowledge of the server program under test to write the client code and cannot be easily generalized. Inspired by Unix's design that treats sockets the same as regular files, LANCET takes a similar approach that implements symbolic sockets as regular symbolic files of a fixed size configurable in the command line. Essentially, instead of simulating sporadic network traffic in real world, LANCET sends a single symbolic packet to the server program under test. Leveraging the existing symbolic file system of KLEE, our implementation creates a fixed number of symbolic files during system initialization, and allocates a free symbolic file to associate with a socket when the socket is created.

4.3 Libevent

Libevent is a software library that provides asynchronous event notification and provides a mechanism to execute a callback function when a specific event occurs on a file descriptor. LANCET implements Libevent as part of its runtime, centering around a core structure, *event base*, the hub for event registration and dispatch. It simulates event base as a concurrent queue that supports multiple producers and consumers. The queue implementation is based on PThread condition variable and mutex for thread synchronization. To register a event, LANCET inserts a new item into the corresponding event queue.

A Libevent-based application usually contains a thread that calls the function `event_base_loop` to execute callback functions for events in a loop. In LANCET's implementation of Libevent, `event_base_base` runs a loop forever to go through an event queue and call registered callback functions for activated events until all events are deleted from the queue. `event_base_loop` also employs PThread condition variable to synchronize with event registration and dispatch.

4.4 Various Optimizations

Simplified libraries Certain functions in the C standard library consume a significant amount of time in symbolic execution. One case is `atoi()`, which is frequently used in C programs to transform a NULL-terminated string into an integer. The standard implementation for `atoi()` supports different bases for the integer and tolerates illegal characters in the input string, all of which add complexity to the code and slow down symbolic execution with a huge number of execution paths. Since LANCET is not concerned with looking for corner cases that trigger bugs—unlike KLEE—we opt to simplify some of these common functions in the runtime. For example, in a simplified `atoi()`, only characters between '0' and '9' is allowed except for the trailing NULL in the string. Note that this simplification applies only to KLEE's handling of `atoi()`; the program under test need not be changed. This simplification also helps keep constraints from different runs in a consistent form and eases the identification of constraint templates.

Scheduling changes The explorer mode of LANCET leverages a code discovery strategy that biases for new code coverage. We found this searching strategy often results in breadth-first traversal of execution paths, causing excessive memory utilization during the search. This phenomenon is because programs usually allocate memory at the beginning of execution and release memory towards the end. Breadth-first search strategies thus perform allocations for every symbolic process, consuming significant amounts of memory. KLEE has a copy-on-write memory sharing mechanism in place to mitigate this problem. However, it cannot skip the duplication of memory allocation when memory initialization is used after allocation. For example, in `libquantum`, `calloc()` is used to acquire memory and initialize the content by zeroing the allocated memory immediately.

To address this problem, LANCET employs an auxiliary search strategy in explorer mode to delay the execution of symbolic processes that are about to make an external function call (*e.g.,*`malloc()` or `calloc()`) This strategy keeps all processes with imminent external calls in a separate queue and prioritizes the execution of processes that stay inside the application's own code. When the only processes left are those in the queue, LANCET dequeues a waiting process and begins execution. Thus, LANCET makes sure that earlier symbolic processes have a chance to release resources before another process allocates additional resources.

5. Evaluation

We have used LANCET to generate inputs for several applications, both in explicit mode and inference mode. The benchmarks we use to evaluate LANCET's explicit mode are `mvm`, a simple program for matrix-vector multiplication, `lq` (libquantum), a simulation of quantum factoring, `lbm`, a Lattice-Boltzmann computational simulation, and `wc`, the Unix word-count utility. We take `Memcached`, a distributed in-memory object caching system as a case study for the inference mode. The first three benchmarks are *numeric*: the inputs to the benchmarks are simple numerical values, which LANCET naturally handles. The fourth benchmark, `wc`, is *structured*: it takes a list of words separated by spaces and line breaks as the input. `Memcached`, on the other hand, serves a variety of different commands defined by its client-server communication protocol. Both `lq` and `lbm` are drawn from the SPEC CPU2006 benchmark suite. In all benchmarks, we targeted the main (outer) loop of the program for scaling, except with `mvm`, where we targeted both the inner (row) loop and outer (column) loop (labeled as `mvm(i)` and `mvm(o)` in the following discussion). For the case study with `Memcached`, we targeted the loop that processes a "get" command, which is used to retrieve data objects associated with the list of keys given in the command.

5.1 General Observations with Benchmarks

5.1.1 LANCET's Explicit Mode versus KLEE

In our first experiment, we compared the effectiveness of LANCET's explicit mode for generating scaling inputs to baseline KLEE. This comparison primarily highlights the modifica-

Table 2. Effectiveness of LANCET and KLEE at generating scaling inputs for target programs. KLEE is unable to generate any inputs for `lbm`, as it runs out of memory.

Bench	LANCET		KLEE	
	# of tests	Max scale	# of tests	Max scale
`mvm(i)`	50	100	285	307
`mvm(o)`	81	81	355	4
`lq`	168	27	4	N/A
`lbm`	14	5	0	N/A
`wc`	67	89	502	375

tions made to the symbolic execution engine in LANCET: the loop-centric search heuristic and the scheduling changes.

For each program, we ran LANCET in explicit mode for 1 hour (with the exception of `lbm`, which we ran for 24 hours, as the target loop is deep in the code), and determined (i) how many tests LANCET was able to generate (only counting tests that execute the loop at least once), and (ii) the largest-scale input LANCET was able to generate (*i.e.,* the largest number of iterations the target loop executed in any synthesized input). We then performed the same experiment using baseline, unmodified KLEE. The results are given in Table 2.

We see that for the more complex programs, `lq` and `lbm`, LANCET is able to generate more test inputs that stress the loop than KLEE, and that those inputs run the loop for more iterations. In `lbm`, KLEE cannot even generate test inputs, as it runs out of memory. LANCET's optimized process scheduling (Section 4) avoids this pitfall. In `lq`, although KLEE successfully generates 4 inputs in one hour, none of them reach the target loop, while LANCET generates 168 different inputs. This advantage arises because of LANCET's "roller" path exploration heuristic. While searching for the loop itself, LANCET behaves similarly to KLEE. However, once a path reaching the loop is found, LANCET builds upon that path as much as possible to run the loop for additional iterations.

Note that the `mvm(i)` numbers are misleading. LANCET was configured to stop generating inputs when it reached 100 iterations. Although KLEE may appear to have generated more, and larger, inputs, LANCET generated the `mvm(i)` inputs in $\sim$1 second, and clearly could have outpaced KLEE. We will perform a more equitable comparison in the final version.

Note that because KLEE does not perform symbolic allocation (allocations are concretized), its normal execution is faster than LANCET; LANCET's advantage arises purely from its optimized path exploration heuristics and process scheduling. In programs where path explosion is attenuated, KLEE can be faster. We see this effect in `wc`, where KLEE generates larger inputs than LANCET. Note, however, that LANCET's inference mode will still allow it to generate large inputs faster, as it need only generate a handful of small inputs to start predicting path conditions for larger inputs.

5.1.2 Inferring Large-scale Inputs

Using the path conditions generated in the explicit mode for each benchmark, we used LANCET's inference mode to predict input values that would run the loop for larger scales. In particular, we attempt to use LANCET to generate an input that will run a loop exactly N times, where N is larger than the scales seen during training. In all cases, LANCET *successfully generates the large-scale input*. This is despite the very different scaling trends that different programs have. For example, `lbm` scales linearly with the input, while `lq` scales *logarithmically* with its input (the loop runs for $\log_2(N)$ iterations), and LANCET is able to correctly capture both trends.

Interestingly, for all benchmarks, LANCET predictions are perfect: *the generated input runs the program for exactly the specified number of iterations*. This is because the behaviors of these applications are deterministic, hence the *constraints* that govern scaling tend to be highly predictable, even if any individual set of inputs may not be.

Table 3. Examples of path conditions generated for `Memcached`. ID, number of iterations and path condition are listed for each generated test. A character or space represents an equality constraint for the byte where it appears. A '*' symbol represents a constraint that enforces a non-space character for the byte where it appears.

Test ID	Iterations	Path Condition
test000001	1	`get *`
test000006	2	`get * *`
test000007	3	`get ** * ****`
test000008	4	`get ** * * *`
test000023	5	`get *** * * *** **** ***`
test000036	6	`get *** * * **** * ** *`

5.2 Case Study with Memcached

`Memcached` runs as a caching server that communicates with clients through TCP connections. Clients send commands to the server to store and fetch data objects. The server reads commands and sends back responses through a socket file descriptor for each client. In LANCET's symbolic socket layer, a socket is implemented as a symbolic file so that a server application can be tested by it self and all its interaction with clients through the socket can be recorded in the content of the symbolic file. For the case study of `Memcached`, we added code to establish a connection with a single client which sends a single symbolic packet of configurable size to the server. Later on, the command processing functions are exercised with the symbolic packet and eventually a response will be written back to the symbolic file.

We targeted at the processing function for a common command for `Memcached`, the "get" command, used for retrieving one or more data objects currently stored in the cache server. The command takes the form of a space-delimited string that contains "get" or "bget" followed by a list of keys, IDs to address specific data objects. The command string is first split into an array of tokens. The tokens are then analyzed by the processing function for the "get" command. We targeted at the main loop in the processing function, `process_get_command`, as shown in Figure 2. In the explicit mode, we set a limit of 10 iterations so LANCET can explore different paths thoroughly at small scales. After running in the explicit mode for around 10 hours, LANCET generated 590 tests that exercise the target loop between 1 and 10 iterations.

Table 3 shows some of the path conditions LANCET generated for `Memcached`. We applied the inference mode on these generated tests and successfully deduced the path condition for exactly 100 iterations by extrapolating the difference between test000001 and test000006. Note there are multiple ways to reach 100 iterations by inferring over the generated tests. For example, we were able to do the same based on test000007 and test000008. However, there also exist path conditions that lead to inaccurate inference if employed as the base cases for path condition extrapolation. For example, the incremental set of test000006 and test000007 is `'* * ****'` starting at the 6th byte of the input. The inference based on these two tests ends up with a path condition that exercises the target loop for 293 iterations. The reason for this inaccuracy is that LANCET only runs the explicit mode for a certain amount of time and could not cover every path for a given number of iterations. As a result, the incremental set between test000006 and test000007 is not the *minimal* incremental set between 2-iteration and 3-iteration path conditions.

6. Related Work

Stress/load test generation The typical method to generate load tests is to induce load by increasing the input size (e.g., a larger query or a larger number of queries) or the rate at which input is provided (e.g., more query requests per unit of time) [19]. However, such techniques are not very discriminating with respect to the kind of load that is introduced, *e.g.,* the kind of query can make a big difference to the response time.

Previous work in workload synthesis [5, 15, 16] applies genetic algorithms to synthesize benchmarks for stress testing. The key idea is to form an abstract space parameterized on a finite number of workload characteristics such as instruction mix, instruction level parallelism, or working set size, and employ genetic algorithms to search for the optimal point with regard to a target metric such as execution time, energy consumption, or voltage fluctuation. Nevertheless, the application is treated as a black box that accepts the generated tests as input and emits the performance metrics as output. In the absence of correlation between tests and code, it is arduous to analyze the testing results and fix the issues found from running these tests.

Zhang *et al.* design a tool that generates tests that target certain types of load [21]. In their scheme, the user is asked for the metric that characterizes load, such as memory consumption. Based on this information, the tool searches for paths, using symbolic execution, that stress that load metric. For example, if the user is interested in memory consumption, then paths containing memory allocation actions are favored. In the background section, we have already presented discussion of another technique in this category, WISE [8], which generates worst-case inputs, *i.e.,* inputs that cause the program to run the longest. FOREPOST [13] uses runtime monitoring for a short duration of testing. The data that is collected is fed through a machine learning algorithm and automated test scripts provide insights into properties of test input data that lead to increased computational loads.

Dynamic techniques for detecting loop problems There are some solutions that apply dynamic techniques to detect problems with a loop, such as, the loop will never terminate, or that the loop has a performance issue. For example, [7] describes LOOPER, which dynamically analyzes a program to detect non-termination. For this, it uses symbolic execution to generate a non-terminating path and then uses the SMT solver, to create concrete variables that will cause such non-termination, if such a situation exists. Some techniques like [17] seek to cure such loop problems.

7. Summary

LANCET is the first system that can generate accurate, targeted, large-scale stress tests for programs. Though it builds upon dynamic symbolic execution, it sidesteps many of the fundamental scaling problems of such techniques through a novel use of statistical inference, allowing LANCET to generate large-scale test inputs without having to run large-scale dynamic symbolic execution. Through a series of case studies, we have demonstrated that LANCET is general, efficient and effective.

References

[1] `http://bugs.mysql.com/bug.php?id=49177`.

[2] `http://memcached.org/`.

[3] `http://clang.llvm.org/`.

[4] `http://llvm.org/docs/BitCodeFormat.html`.

[5] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC '10*.

[6] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys '11*.

[7] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE '09*.

[8] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *ICSE '09*.

[9] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08*.

[10] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[11] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV '07*.

[12] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05*.

[13] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE '12*.

[14] High Scalability. Troubles with Sharding - What can we learn from the Foursquare Incident? `http://goo.gl/D5ixm4`.

[15] A. M. Joshi, L. Eeckhout, L. K. John, and C. Isen. Automated microprocessor stressmark generation. In *HPCA '08*.

[16] Y. Kim, L. K. John, S. Pant, S. Manne, M. Schulte, W. L. Bircher, and M. S. S. Govindan. Audit: Stress testing the automatic way. In *MICRO '12*.

[17] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *OOPSLA '12*.

[18] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, February 2012.

[19] Rick Hower. Web Site Test Tools and Site Management Tools: Load and Performance Test Tools. `http://www.softwareqatest.com/qatweb1.html#LOAD`.

[20] K. V. Shvachko. HDFS scalability: The limits to growth. *USENIX login*, 35(2):6–16, 2010.

[21] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *ASE '11*.