A Scalable Distributed Concolic Testing Approach

Moonzoo Kim, Yunho Kim
Department of Computer Science
KAIST, South Korea
moonzoo@cs.kaist.ac.kr, kimyunho@kaist.ac.kr

Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
grother@cse.unl.edu

***Abstract*—Although testing is a standard method for improving the quality of software, conventional testing methods often fail to detect faults. Concolic testing attempts to remedy this by automatically generating test cases to explore execution paths in a program under test, helping testers achieve greater coverage of program behavior in a more automated fashion. Concolic testing, however, consumes a significant amount of computing time to explore execution paths, which is an obstacle toward its practical application. In this paper we describe a distributed concolic testing framework that utilizes large numbers of computing nodes to generate test cases in a scalable manner. We present the results of an empirical study that shows that the proposed framework can achieve a several orders-of-magnitude increase in test case generation speed compared to the original concolic approach, and also demonstrates clear potential for scalability.**

I. Introduction

Dynamic testing is a de-facto standard method for improving the quality of software in industry. Conventional testing methods, however, often fail to detect faults in programs. One reason for this is that a program can have an enormous number of different execution paths due to conditional and loop statements. Thus, it is practically infeasible for a test engineer to manually create test cases sufficient to detect subtle bugs in specific execution paths. In addition, it is a technically challenging task to generate effective test cases in an automated manner.

To address such limitations, concolic (CONCrete + symbOLIC) [1] testing (also known as dynamic symbolic execution [2] and white-box fuzzing [3]) combines concrete dynamic analysis and static symbolic analysis to automatically generate test cases to explore execution paths of a program, to achieve full path coverage (or at least, coverage of paths up to some bound). Concolic testing can be effective for generating test cases; however, it also can consume a significant amount of time exploring execution paths, and this is an obstacle toward its practical application [4].

In this paper we present the Scalable COncolic testing for REliability (SCORE) framework; a test case generation approach aimed at addressing the limitations caused by the heavy computational cost of concolic testing. The SCORE framework employs a distributed concolic algorithm that can utilize a large number of computing nodes in a scalable manner so as to achieve:

- a linear increase in the speed of test case generation as the number of distributed nodes increases;
- low communication overhead among distributed nodes.

The SCORE framework can utilize computing nodes in P2P networks or cloud computing platforms. In doing so, SCORE can potentially generate test cases significantly faster than traditional concolic testing as more computing nodes are employed in the testing task.

To investigate the effectiveness and scalability of the SCORE framework, we conducted a controlled experiment in which we applied the framework to several C programs, using numbers of Amazon EC2 nodes ranging up to 256. Our results show that SCORE can greatly increase the effectiveness of test case generation, and that it is scalable as the numbers of nodes utilized increases.

The rest of the paper is organized as follows. Section II describes related work. Section III describes the SCORE framework. Section IV presents our empirical study, and Section V discusses observations from the study. Section VI concludes and discusses future work.

II. Related Work

The core idea behind concolic testing is to obtain symbolic path formulas from concrete executions and solve them to generate test cases by using constraint solvers. Various concolic testing tools have been implemented to realize this core idea (see [5] for a survey). Existing tools can be classified in terms of the approach they use to extract symbolic path formulas from concrete executions.

The first approach for extracting symbolic path formulas is to use modified virtual machines. The concolic testing tools that use this approach are implemented as modified virtual machines on which target programs execute. An advantage of this approach is that the tools can exploit all execution information at run-time, since the virtual machine possesses all necessary information. PEX [2] targets C# programs that are compiled into Microsoft .Net binaries, KLEE [6] targets LLVM [7] binaries, and jFuzz [8] targets Java bytecode on top of Java PathFinder [9], [10].

The second approach for extracting symbolic path formulas is to instrument the target source code to insert probes that extract formulas from concrete executions at run-time. Compared to the first approach, this approach is more light-weight, because adding probes is simpler than modifying virtual machines. Tools that use this approach include CUTE [1], DART [11], and CREST [12], which operate on C programs, and jCUTE [13], which operates on Java programs. SCORE uses this approach.

Because concolic testing has been studied for a relatively short period of time, there has been little research on employing distributed platforms to improve the scalability of concolic testing techniques. Staats et al. [14] propose a *static partitioning* technique for parallelizing symbolic execution that uses pre-conditions/prefixes of symbolic executions to partition the symbolic execution tree. They have implemented the technique on top of Java PathFinder [9] using the Symbolic PathFinder extension [10]. A limitation of this approach is that the resulting partitioned symbolic execution trees are *not well-balanced*, because the technique statically partitions a symbolic execution tree based only on its prefixes. Thus, some nodes may finish exploring symbolic execution paths quickly and become idle while other nodes take long times to complete exploration, which degrades overall performance.

In contrast, King [15], ParSym [16], and Cloud9 [17] utilize *dynamic partitioning* of target program executions. King's master's thesis [15] describes a distributed symbolic execution framework for Java [18]. King populates a queue of symbolic execution subtrees dynamically, but the resulting speedup decreases as the number of nodes increases beyond six. ParSym [16] uses a central server that collects test cases generated from nodes and distributes the test cases to the other nodes whose queues are empty. ParSym demonstrates speedup on `grep` 2.2 and a binary tree program on up to 512 nodes, but fails to achieve linear speedup (e.g., achieving a speedup of 13 times on 128 nodes but a speedup of only 6 times on 512 nodes on `grep`). Cloud9 [17] is a testing service framework based on parallel symbolic execution techniques implemented on KLEE. Cloud9 uses dynamic partitioning to ensure that the job queue lengths of all nodes stay within a given range and shows linear speed-up as the number of nodes increases up to 48. Similar to these techniques, SCORE distributes test cases among multiple nodes in a dynamic on-demand manner (Section III-B) and achieves linear speed-up on up to 256 nodes (Section IV-E). Note that Cloud9 and SCORE utilize different parallelization techniques. In Cloud9, when an execution meets a branch containing symbolic values, two parallel executions are *forked* with a corresponding clone of the program state and distributed to nodes to continue (Cloud9 can obtain parallel executions since Cloud9 operates as a virtual machine). In contrast, SCORE generates whole executions one by one by negating symbolic branch conditions in a systematic manner while preventing redundant test cases (Section III-A).

In our own preliminary work [19], we created a proof-of-concept implementation of a distributed concolic algorithm and conducted a case study on a single program, running on up to 16 nodes. In this work, we have redesigned the communication protocol for that approach and implemented a fully developed distributed concolic testing framework. We describe the results of this effort, and present results of a controlled experiment examining the effectiveness and scalability of the framework on up to 256 nodes.

III. The SCORE Framework

We now describe the SCORE framework, beginning with a description of the distributed concolic algorithm (Section III-A), followed by details on communications between nodes (Section III-B). Section III-C then discusses our implementation of the framework.

A. Distributed Concolic Algorithm

Algorithm 1 describes how to generate test cases on a single node in the SCORE framework. We assume that there exists one startup node that runs $DstrConcolic(startup)$ with $startup$ as true (line 1). This startup node running $DstrConcolic()$ generates a test case pair (tc_1,neg_limit_1) where tc_1 is a random value and $neg\_limit_1 = 1$ (lines 6-7). The other non-startup nodes running $DstrConcolic(false)$ wait until they receive test case pairs from other nodes (lines 9-11).

Next, a node begins generating further test case pairs from a test case pair (tc,neg_limit) in the queue q_{tc} (lines 14-33). First, the algorithm removes (tc,neg_limit) from q_{tc} (line 15) and obtains a symbolic path formula ϕ_i (line 19) from the concrete execution on tc (line 17). Then, the algorithm generates further symbolic path formulas ψ_is by negating path conditions (i.e., $c_1, c_2, ..., c_{|\phi_i|}$) of ϕ_i one by one in decreasing order through the while loop (lines 21-32). [1] These ψ_is are solved by a SMT solver (line 25) and a corresponding solution to ψ_i is stored in q_{tc} as a new test case pair $(tc_{i+1}, j + 1)$ (line 27).

If q_{tc} is empty (exiting the loop of lines 14-33) and the q_{tc}s of all distributed nodes are empty, the algorithm terminates (line 39). Otherwise (i.e., there is another node n'' that has test cases), a current node n requests test cases from n'' (line 35) and receives test cases from n'' (line 36). The received test cases are then added into q_{tc} (line 37) and the algorithm continues from line 13.

Note that Algorithm 1 traverses all possible execution paths and does *not* generate redundant test cases (test cases that cover the same path) with the assumption that ϕ_i truly reflects $path_i$ and $Solve(\psi_i)$ can solve ψ_i. [2] More detail on

[1]The use of neg_limit_i for tc_i prevents the algorithm from generating redundant test cases. neg_limit_i is an index to the path condition (PC) in ϕ_i beyond which PCs should not be negated (lines 21-23) (i.e., c_k should not be negated for $k < neg\_limit_i$), where ϕ_i is a symbolic path formula obtained from an execution path on tc_i (line 19).

[2]In practice, a program P may contain complex arithmetic or binary library calls that cannot be reasoned about by SMT solvers. Thus, Algorithm 1 generates symbolic path formulas without such conditions, and in these cases this may result in redundant test cases.

```
Input:
$startup$: a flag to indicate whether a current node $n$ is a startup node or not.
Output:
$TC_n$: a set of test cases generated at a current node $n$ (i.e., $tc_{i+1}$s of line 25)
1  $DstrConcolic(startup)$ {
2  $q_{tc} = \emptyset$; // a queue containing ($tc$, $neg\_limit$)s
3  $TC_n = \emptyset$; // a set of generated test cases
4  $i = 1$;
5  if startup then
6      $tc_1$ = random value; // initial test case
7      Add $(tc_1, 1)$ to $q_{tc}$;
8  else
9      Send a request for test cases to $n'$;
10     Receive $(tc, neg\_limit)$s from $n'$;
11     Add $(tc, neg\_limit)$s to $q_{tc}$;
12 end
13 while true do
14     while $| q_{tc} | > 0$ do
15         Remove $(tc, neg\_limit)$ from $q_{tc}$;
16         // Concrete execution
17         $path_i$ = an execution path on $tc$;
18         // Obtain a symbolic path formula $\phi_i$
19         $\phi_i$ = a symbolic path formula of $path_i$ (i.e., $c_1 \wedge c_2 \wedge ... \wedge c_{|\phi_i|}$) ;
20         $j = | \phi_i |$;
21         while $j >= neg\_limit$ do
22             // Generate $\psi_i$ for the next input values
23             $\psi_i = c_1 \wedge ... \wedge c_{j-1} \wedge \neg c_j$ ;
24             // Select the next input values
25             $tc_{i+1} = Solve(\psi_i)$;
26             if $tc_{i+1}$ is not NULL then
27                 Add $(tc_{i+1}, j+1)$ to $q_{tc}$;
28                 $TC_n = TC_n \cup \{tc_{i+1}\}$;
29                 $i = i + 1$;
30             end
31             $j = j - 1$;
32         end
33     end
34     if there is a test case in another node $n''$ then
35         Send a request for test cases to $n''$;
36         Receive $(tc, neg\_limit)$s from $n''$;
37         Add $(tc, neg\_limit)$s to $q_{tc}$;
38     else
39         Halt; // no test cases exist in all nodes
40     end
41 end
42 }
```

Algorithm 1: Distributed concolic algorithm

Algorithm 1 including a comparison between the original concolic algorithm and Algorithm 1 and the proofs of the algorithm's traversal of all possible execution paths and its unique test case generation are described in our technical report [20].

B. Communication between Nodes

Figure 1 illustrates the communication among nodes in the SCORE framework. SCORE operates on distributed computing nodes where one node operates as a server and the other nodes operate as clients. In addition, one client is designated as a startup client, which initiates distributed concolic testing (lines 5-7 in Algorithm 1).

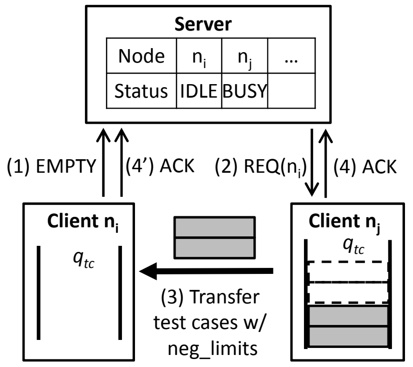

Figure 1. Communication among clients and the server

We assume that all clients are connected to the server properly and the network does not lose or corrupt its messages, neither change message delivery order. There are five different control packets sent between a server and clients to coordinate distributed clients that generate test cases.

- An `EMPTY` packet is sent to the server from a client n_i that has no test case pair in its q_{tc}.
- A `REQ` packet is sent from the server to a client n_j that has test case pairs and that is not currently serving any `REQ` packet.
- An `ACK` packet is sent from n_j that receives a `REQ` packet and has transferred test case pairs to n_i that sent an `EMPTY` packet to the server. In a similar manner, an `ACK` packet is sent from n_i that sent an `EMPTY` to the server after n_i receives test case pairs.
- A `NACK` packet is sent from n_j that has received a `REQ` packet and could not transfer test case pairs, since $|q_{tc}| \leq 1$.
- A `STOP` packet is sent from the server to clients when every client has no test case pair (i.e., the distributed concolic testing process is completed).

To identify which client has test case pairs to transfer, the server maintains a status table of all clients. From the viewpoint of the server, the status of a client must be one of the following:

- IDLE when $\mid q_{tc} \mid = 0$. IDLE indicates that the client has no test case pairs to execute.
- BUSY when $\mid q_{tc} \mid > 0$ and the client is not serving a request from the server. BUSY indicates that the client is ready to provide test case pairs to another client.
- SERVING_REQ when the client is serving a request from the server. In other words, the client received a REQ but has not yet sent an ACK or NACK to the server. SERVING_REQ indicates that the client is not currently able to send test case pairs to another client.

By tracking control messages between the server and the client, the server knows the status of clients. If the server receives EMPTY from a client, then the client has no test case pairs (IDLE). If the server receives an ACK from a client n_j that sent test case pairs to n_i, then the server considers the status of n_j to be BUSY. In a similar manner, if the server receives an ACK from a client n_i, then the server considers the status of n_i to be BUSY. If the server has sent REQ to a client and has not yet received an ACK or NACK, then the state of the client is SERVING_REQ.

Figure 1 illustrates the following scenario. When the server receives EMPTY from a client n_i, it searches the client status table and finds a client n_j whose status is BUSY. Then, it sends REQ to n_j with destination n_i. If n_j has m test case pairs (i.e., $\mid q_{tc} \mid = m$), it sends $\lfloor m/2 \rfloor$ test case pairs to n_i and then sends ACK to the server. Client n_i also sends ACK to the server after it receives test case pairs. If client n_j has one or zero test case pairs, it sends NACK to the server. Then, the server tries another client whose status is BUSY in a round-robin manner. If no client is in BUSY state, the server waits until at least one client enters the BUSY state. If all clients are IDLE, the server sends a STOP message to the clients, and the distributed concolic algorithm terminates. When clients receive a STOP from the server, the clients transfer generated test cases, covered path/branch information, and statistics on testing activities (e.g., total time, number of test cases generated, number of symbolic formulas generated, etc) to the server.

Note that communication between clients n_i and n_j occurs only when q_{tc} of n_i is empty. Because q_{tc} is non-empty for most of the testing time, [3] the number of communications is small compared to the number of test cases generated. Furthermore, test case pairs are directly transferred between clients without causing heavy load on the server. Otherwise, the server would become a bottleneck.

C. The SCORE Implementation

The SCORE framework is implemented to operate on distributed computers connected through TCP/IP networks. Thus, SCORE can operate on a large number of computing nodes such as on a cloud computing platform or any computers connected through the internet. The SCORE framework uses CREST 0.1.1 [12] (an open-source concolic testing tool) to instrument a target C program and to obtain symbolic formulas from concrete execution paths at runtime. It uses Yices 1.0.24 [21] as an internal constraint solver for solving symbolic path formulas. However, the current SCORE implementation handles only linear-integer arithmetic statements symbolically (other statements are handled using concrete values), since CREST supports only linear-integer arithmetic statements. We plan to upgrade SCORE to support bit-vector theory to alleviate this limitation. The SCORE framework is written in C/C++ and contains 7000 lines of code with 18 classes and 215 functions.

For n clients, the server creates n threads, each of which communicates with one client. To minimize communication overhead, each client is implemented as two separate threads. One thread generates and stores test case pairs in q_{tc} through the distributed concolic algorithm while the other thread handles communication with other nodes such as receiving test case pairs into q_{tc} or sending test case pairs from q_{tc} to another node per request. All communications in the SCORE framework are implemented using TCP sockets, since the framework may be deployed on a large scale computing platform where communication might not be reliable. In addition to the communication between clients and the server described in Section III-B, as logs, clients periodically report to the server the current status of testing activities such as the number of test cases generated, the size of q_{tc}, the number of test cases received from another node, and so forth. Each client stores testing outcomes such as test cases generated, covered branches, and covered execution paths on the local hard disk. When the testing process terminates or a user sends a command to stop the concolic testing, these outcomes are collected by the server automatically.

IV. Empirical Study

There are two overall methodologies that we could use to empirically study the SCORE framework. The first methodology involves empirically comparing SCORE to other approaches for parallelizing dynamic symbolic execution such as those of Staats et al. [14], King [15], ParSym [16] and Cloud9 [17], discussed in Section II. At this time, however, such a comparison would be difficult, because Staats' approach is implemented only for Java, and the other approaches are not available in implemented form. Moreover, comparisons of implementations created in different contexts pose many threats to internal validity in terms of ensuring the comparability of techniques.

The second methodology we can use to assess SCORE involves comparing the framework to baseline approaches that allow direct assessment of the benefits of SCORE's

[3]A client generates and stores multiple new test case pairs in q_{tc} (lines 21-32 of Algorithm 1) by removing one test case pair from q_{tc} (line 15 of Algorithm 1), except when the client targets leaves of an execution tree.

distributed aspects. For example, we can compare SCORE to the original concolic testing approach applied on distributed nodes. If this comparison does *not* show that SCORE is effective and scalable, then the results obviate the need to perform expensive implementations of other techniques. We thus chose this approach.

The SCORE framework is meant to increase the *effectiveness* of concolic testing at generating potentially useful test inputs by distributing workload over large numbers of client nodes. However, the efficiency of distributed algorithms tends to decrease as the number of client nodes increases due to redundant computations, overhead related to increasing communications, and unbalanced workloads. These issues impact the *scalability* of these algorithms. The degree to which the framework can achieve these attributes on real workloads, however, must be assessed empirically.

To provide such an assessment, we designed an empirical study addressing the following research questions:

- **RQ1**: To what extent does the SCORE framework increase the effectiveness of test case generation?
- **RQ2**: To what extent does the SCORE framework achieve scalability?

A. Target Benchmark Programs

As objects of study, we selected six programs (see Table I) from the SIR repository [22], including three of the Siemens programs [23], and three non-trivial real-world programs (`grep` 2.0, `sed` 1.17, and `vim` 5.0). We selected these programs because they were written in C and thus can be processed by our tools, and because they do not utilize intensive numbers of non-linear integer arithmetic statements.

Table I
EXPERIMENT OBJECTS

Program	Functions	LOC	Branches	Test cases
`grep`	126	12562	3768	808
`ptok1`	19	725	284	4140
`ptok2`	24	569	168	4140
`repl`	2	563	210	5543
`sed`	70	8678	2690	389
`vim`	3049	111227	33486	975

To perform concolic testing on these programs, we needed to decide what sizes of inputs to utilize for symbolic variables. To make a realistic decision, we reviewed all of the test cases provided for each program in the SIR repository. We selected symbolic input sizes as the maximum size of the 90% of the smallest test cases in the SIR repository, since there is often a large gap between that size and the sizes of the remaining 10% of the test cases. Table II shows the sizes of symbolic inputs chosen for each program.

We provided two symbolic options for `grep`, because `grep` uses three different algorithms for pattern matching based on a given option. Also, 90% of the test cases for `grep` in the SIR repository contain two or fewer options; for an option with an argument, we assigned one symbolic

Table II
SIZE OF SYMBOLIC INPUTS (BYTES)

`grep`	`ptok1`	`ptok2`	`repl`			`sed`	`vim`
pattern	target text	target text	from	to	target text	com-mand	script
21	82	82	23	28	64	148	76

character as a corresponding argument. Finally, for `grep` and `sed`, we chose to use target text files provided in the SIR repository as concrete inputs rather than to generate file contents symbolically, because meaningful files are too large (greater than 100k) to treat as symbolic input.

B. Variables and Measures

To address our research questions, our experiment manipulated two independent variables:

IV1: Test case generation technique
To investigate RQ1 we study two techniques: the distributed algorithm implemented in the SCORE framework, and the non-distributed concolic testing algorithm [12]. Further, to facilitate comparisons we apply the non-distributed (baseline) algorithm in two ways: (1) running a single instance of the algorithm on a single node, and (2) running multiple instances, one on each client node, with different random seeds, and aggregating results from each node in the end. The former approach lets us assess the effectiveness of SCORE relative to current practice, while the latter lets us assess whether the particular parallelization solution used by SCORE is more effective than the naive approach of simply running the original algorithm on the same number of nodes.

IV2: Client number level
To investigate scalability issues such as those posed by RQ2, as well as to examine technique effectiveness over a wide range of distribution settings, we need to apply techniques using different numbers of client nodes. We chose to use client number levels 1, 64, 128, 192, and 256.

To measure effectiveness and scalability we selected five dependent variables. The first variable tracks effectiveness in terms of numbers of test cases that can be generated in a given time. An issue that arises in this context, however, concerns redundancy. As explained earlier, in the context of concolic testing, where test cases are generated to cover paths, test cases that cover the same paths are redundant. In theory, neither the SCORE algorithm nor the original concolic algorithm can generate redundant test cases. [20] However, limitations in concolic algorithms or limitations in symbolic execution engines can lead in practice to cases where redundancy does occur. Since the different techniques that we compare may differ in terms of the number of redundant test cases that they create, a fair comparison of their effectiveness must exclude redundant test cases. Thus, our first dependent variable focuses on creation of non-redundant test cases.

DV1: Number of non-redundant test cases generated in time τ
We chose $\tau = 5$ minutes, because exploratory studies with smaller and larger times suggested that increases beyond 5 minutes had negligible effects on results.

To track scalability we use four variables, each of which represents a different important aspect of performance in the context of distributed algorithms.

DV2: Number of communicated messages
We measured the total number of messages communicated between nodes (both server and client).

DV3: Communication overhead in terms of waiting time
We measured the elapsed waiting time between lines 34 and 37 of Algorithm 1 due to empty q_{tc}, for each client node.

DV4: Workload assigned to each client
We measured the number of test cases generated by each client node.

DV5: Efficiency of the SCORE framework
We measured the efficiency of the framework by calculating its effectiveness ratios (i.e., # of test cases generated by SCORE over # of test cases generated by the non-distributed concolic algorithm) over the number of clients, $\frac{\text{effectiveness ratio}}{\text{\# of clients nodes}}$.

C. Experiment Operation

For each target program, we executed the non-distributed CREST algorithm and the distributed SCORE algorithm on each of the five client number levels. To control for potential differences in runs due to the randomization inherent in the techniques, we repeated this process 30 times.

To count the non-redundant test cases in a generated test suite, we stored the sequence of branch IDs $B_i = [b_{i_1}, b_{i_2}, ...b_{i_n}]$ that were executed on each test case tc_i. Then, we used a SHA512(B_i) as a representative hash value for each B_i, because the total number and size of B_is are large (e.g., we had a total of 0.8 million B_is consuming 500 gigabytes for VIM on 256 nodes). In our experimental runs, the number of hash collisions was negligible. As a sample, we compared all B_is directly in one entire set of technique runs on 256 nodes, and found no hash value collisions. Finally, we compared SHA512(B_i) values with each other to remove redundant test cases, keeping exactly one test case from each set of test cases that execute identical paths.

All experiments were performed on the Amazon EC2 cloud computing platform [24]. The server of the SCORE framework ran on a virtual node that had seven gigabytes of memory and eight CPU cores of 20 ECU computing power in total (1 ECU is equivalent to a 1Ghz Xeon processor). Each client ran on a virtual node that was equipped with 1.7 of gigabytes memory and two CPU cores of 5 ECU in total. The server and clients ran on Fedora Core Linux 8. All virtual nodes are connected through a 1 gigabps Ethernet.

D. Threats to Validity

The primary threat to external validity for our study involves the representativeness of our object programs, since we have examined only six C programs (although three of them are real-world applications). Furthermore, we have chosen programs that are amenable to concolic testing, and thus, do not reveal cases in which program characteristics might hinder that approach. As a second threat, we have employed only the CREST tool as an example of an original concolic algorithm implementation; results obtained with other implementations may differ. A third threat to validity is the limited power of the underlying symbolic execution engine used in SCORE. Currently, SCORE handles only linear-integer arithmetic (LIA) statements. A symbolic execution engine using a more powerful solver (e.g., a bit-vector solver) could cover execution paths that cannot be covered using an LIA solver, but with decreased test case generation speed due to the presence of more complex symbolic path formulas. Thus, results could differ in such cases. We believe, however, that the use of a more powerful solver would not affect assessments of the scalability of SCORE, because SCORE's scalability is primarily affected by communication cost, and SCORE's communication protocol is independent of the SMT solver.

The primary threat to internal validity is possible faults in the implementation of our algorithms and in tools we use to collect metrics. We controlled for this threat through extensive functional testing of our tools. A second threat pertains to differences in the implementations compared; we limited this threat by using the same underlying concolic test case generation tool in all cases.

Where construct validity is concerned, there are other metrics that could be pertinent to the effects studied. In particular, as an effectiveness measure we consider only numbers of non-redundant test cases generated (which correlates with path coverage achieved). In contrast, studies of Cloud9 [17] relied on statement coverage as an effectiveness measure. We believe, however, that our effectiveness measure is more appropriate for assessing concolic testing, since concolic testing targets path coverage, not statement or branch coverage. As another potential metric for assessing effectiveness, numbers of faults detected could also be considered. To measure scalability, the time required to reach a fixed level of statement/branch coverage or the time consumed to explore an execution subtree in a k-depth bound completely can also be considered. These metrics, however, have weaknesses. First, concolic testing may not be able to generate test cases to reach a given fixed level of coverage. In addition, it is difficult to accurately estimate the time required to reach a fixed level of coverage in advance, causing difficulty for experiment design. Yet another metric for scalability in terms of workload distribution, also employed in [17], is to measure the number of target program instructions executed per node. This might be a valid performance indicator for parallel algorithms in general, but when the goal of such algorithms is test case generation, we believe that it is not the best measure.

E. Results and Analysis

We now present and analyze our results, per research question.

1) RQ1: To what extent does SCORE increase the effectiveness of concolic testing?: We begin by comparing SCORE to the non-distributed algorithm run on a single node. Table III displays the mean total numbers of non-redundant test cases generated by both the non-distributed and distributed algorithms, for each of the object programs, per client number level, across all 30 runs of the algorithm at that level. As the table shows, the number of test cases generated by SCORE increased substantially as the number of client nodes increased.

Table III
TOTAL # OF NON-REDUNDANT TESTS GENERATED PER CLIENT NUMBER LEVEL, NON-DISTRIBUTED AND DISTRIBUTED ALGORITHMS

	CREST	SCORE				
	1	1	64	128	192	256
`grep`	961	997	76612	134503	260244	322345
`ptok1`	2315	2385	135842	289335	429770	559250
`ptok2`	29340	29118	1184348	2310901	3490055	4553864
`repl`	25784	26181	723430	1416901	2242355	2978568
`sed`	16550	16679	592491	1137879	1701740	2262199
`vim`	2378	2390	50914	98964	140141	189038

Figure 2 illustrates the effectiveness (i.e., number of non-redundant test cases generated) increase achieved by SCORE as the client number level increased, in a manner that compares the two algorithms. The figure compares effectiveness results obtained by the distributed algorithm at all five client number levels to the results obtained by the non-distributed algorithm on a single node, per program, in terms of the ratio of numbers of test cases generated by each. Effectiveness appears to increase *linearly* with client number level, but the rate of increase does vary per program.

To assess whether the observed differences in performance were statistically significantly different, we applied Wilcoxon tests [25] to the effectiveness data achieved at each client number level, per program, by SCORE and the non-distributed algorithm, with $\alpha = 0.05$ as confidence level. In every case above client number level 1 the differences were statistically significant.

It is worth noting that the effectiveness ratio for `grep` at client number level 256 is greater (i.e., 335.4 = 322345/961) than the number of client nodes. We believe that the reason for this is that the average length of the symbolic path formulas analyzed to generate 961 test cases by one client

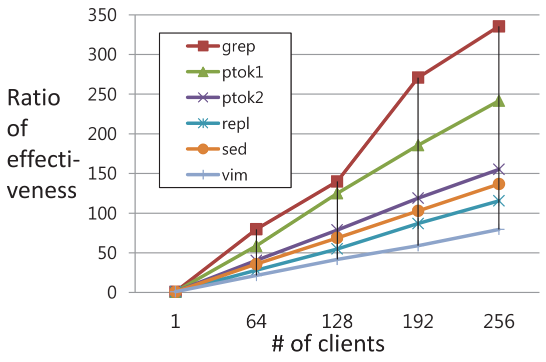

Figure 2. Ratio of effectiveness between non-distributed and distributed algorithms, per client number level

node using the non-distributed algorithm was larger than the average length of the symbolic path formulas analyzed to generate the 322345 test cases by 256 client nodes using the distributed algorithm. The relatively low effectiveness ratio (i.e., 79.5 = 189038/2378) for `vim` at client number level 256 can be explained similarly.

Note also that the numbers of test cases generated by the non-distributed algorithm and SCORE did differ when the algorithms were each run on one node (see the second and third columns of Table III), because the test cases generated by the algorithms on any given run can differ. However, the magnitude of the difference in performance is relatively small, ranging from 0.5% (on `vim`) to 1.5% (on `repl`).

We next compare SCORE to the use of the non-distributed algorithm on the same numbers of client nodes. Table IV shows the mean numbers of non-redundant test cases generated by SCORE (S) and multiple runs of CREST (MC) across the 30 runs performed in each case, as well as the ratios between those means.

Table IV
NUMBERS OF NON-REDUNDANT TEST CASES

# of nodes		`grep`	`ptok1`	`ptok2`	`repl`	`sed`	`vim`
64	S	76612	135842	1184348	723430	592491	50914
	MC	22718	47484	656797	339735	442816	2539
	S/MC	**3.37**	**2.86**	**1.80**	**2.13**	**1.34**	**20.05**
128	S	134503	289335	2310901	1416901	1137879	98964
	MC	42875	86125	1193297	614645	829558	3164
	S/MC	**3.14**	**3.36**	**1.94**	**2.31**	**1.37**	**31.28**
192	S	260244	429770	3490055	2242355	1701740	140141
	MC	58247	122410	1654794	794858	1195363	3758
	S/MC	**4.47**	**3.51**	**2.11**	**2.82**	**1.42**	**37.29**
256	S	322345	559250	4553864	2978568	2262199	189038
	MC	75449	147003	2085353	906482	1455585	3886
	S/MC	**4.27**	**3.80**	**2.18**	**3.29**	**1.55**	**48.65**

As the data shows, the number of test cases generated by SCORE ranged from 34% (`sed` on 64 nodes) to 4765% (`vim` on 256 nodes) more than the number generated by multiple CREST runs. In addition, the S/MC ratios have a tendency to increase over the number of nodes utilized. For example, for `ptok1`, the ratios increase from 2.86 on

64 nodes to 3.80 on 256 nodes. (The only cases in which the ratios do not increase involve `grep`, going from 64 to 128 and 192 to 256 nodes). We conjecture that the reason for this S/MC increase over increasing number of client nodes is that the diversity effect obtained from initial random input becomes, compared to SCORE, relatively weaker as the number of client nodes increases. Again, Wilcoxon tests applied to the effectiveness data achieved at each client number level, per program, by SCORE and by the non-distributed algorithm, using $\alpha = 0.05$ as the confidence level, showed that in every case the differences were statistically significant. SCORE thus increases effectiveness more than the simple approach of running multiple instances of the non-distributed concolic algorithm.

2) RQ2: To what extent does the SCORE framework achieve scalability?: Table V shows the number of communicated messages (left) and the communication overhead (right) for the distributed algorithm, for each object program, for client number levels greater than 1. The number of communications is small compared to the number of generated non-redundant test cases. For example, using 64 client nodes on `grep` resulted in 3391 messages being communicated between client nodes and server, which is equivalent to 53 (3391/64) messages per client node. In other words, each client node communicates 53 messages while generating 1197 (76612/64) test cases. `Vim` communicates more messages per generated non-redundant test case (e.g., 12.36% on 64 nodes) than the other programs; this is because `vim` contains more complex statements and external binary libraries than the other programs do. Consequently, SCORE often fails to generate further test cases for `vim` and obtains an empty q_{tc} more frequently than on the other programs.

Table V
STATISTICS ON COMMUNICATION

	# of total messages and the ratios of # of msgs/# of TCs				Comm. overhead (%)			
	64	128	192	256	64	128	192	256
`grep`	3391 (4.43%)	5363 (3.99%)	6897 (2.65%)	6900 (2.14%)	0.10	0.21	0.24	0.21
`ptok1`	1604 (1.18%)	2467 (0.85%)	3874 (0.90%)	4009 (0.72%)	0.06	0.11	0.15	0.13
`ptok2`	1598 (0.13%)	1836 (0.08%)	3118 (0.09%)	3098 (0.07%)	0.07	0.14	0.13	0.15
`repl`	1569 (0.22%)	1780 (0.13%)	3601 (0.16%)	2980 (0.10%)	0.07	0.14	0.18	0.13
`sed`	2655 (0.45%)	3317 (0.29%)	4449 (0.26%)	4607 (0.20%)	0.12	0.21	0.23	0.18
`vim`	6295 (12.36%)	11238 (11.36%)	14183 (10.12%)	14742 (7.80%)	0.52	0.73	0.79	0.74

Note that the ratio of number of messages over generated non-redundant test cases usually decreases as the number of client nodes increases. For example, the ratios over the four client number levels on `grep` are 4.43%, 3.99%, 2.65%, and 2.14%, respectively (see the numbers in parentheses in Table V). This is because except for the startup client, clients begin with an empty q_{tc} and need to communicate to obtain initial test cases, but following the initial communication much fewer subsequent communications are needed. We designed a simple communication protocol (at the cost of boot-up time overhead). Suppose that there are 64 clients, $n_1, ...n_{64}$, and n_1 has one test case but the other clients have no test cases. Initially, n_2 sends `EMPTY` to the server. Then, a corresponding server thread sends `REQ` to n_1 and n_1 sends back `NACK` to the server thread. Consequently, the server thread sends `REQ` to n_1 again and creates many `REQ`-`NACK` messages until n_1 has more than one test case or no test case (this worst-case scenario occurs only during the short boot-up time – less than 1 second – since $|q_{tc}| > 1$ most of the time). At the same time, the other 62 clients $(n_3, ...n_{64})$ send `EMPTY` packets to the server. However, corresponding server threads (thus, also the 62 clients) will block, since n_1 is in `SERVING_REQ` state to serve n_2. For 256 clients, more clients (i.e., 254 clients) will block in the same manner, thus creating relatively fewer `REQ`-`NACK` messages per clients.

As shown on the right side of the table, total communication overhead was less than 0.3% of total execution time, except on `vim` where it ranged up to 0.8%. Since generating a new test case (including concolic execution and solving symbolic path formulas) takes much more time than communication, the overhead of waiting time including communication time is negligible.

Figure 3 presents boxplots showing workload distributions for all six object programs. The horizontal axes indicate client number levels, and the vertical axes indicate workload (number of test cases generated per client node). The central 50% of the data points (those denoted by the box) exhibit relatively small variance, and results do not vary widely as client number levels increase. This provides further evidence of scalability.

Finally, Table VI depicts the efficiency (i.e., $\frac{\text{effectiveness ratio}}{\text{\# of clients}}$) of the SCORE framework across different client number levels, per program. The efficiency of the distributed algorithm does not decrease, but remains almost constant over different client number levels. For example, the efficiencies for `ptok1` are 0.92, 0.98, 0.97, and 0.94 for 64, 128, 192, and 256 clients, respectively.

Table VI
EFFICIENCY OF CLIENTS AT GENERATING TEST CASES

$\frac{\text{effectiveness ratio}}{\text{\# of clients}}$	64	128	192	256
`grep`	1.25	1.09	1.41	1.31
`ptok1`	0.92	0.98	0.97	0.94
`ptok2`	0.63	0.62	0.62	0.61
`repl`	0.44	0.43	0.45	0.45
`sed`	0.56	0.54	0.54	0.53
`vim`	0.33	0.33	0.31	0.31

V. DISCUSSION

Fault Detection: We have not yet evaluated the fault-detection capabilities of our distributed approach to concolic test generation. However through our testing, we did detect

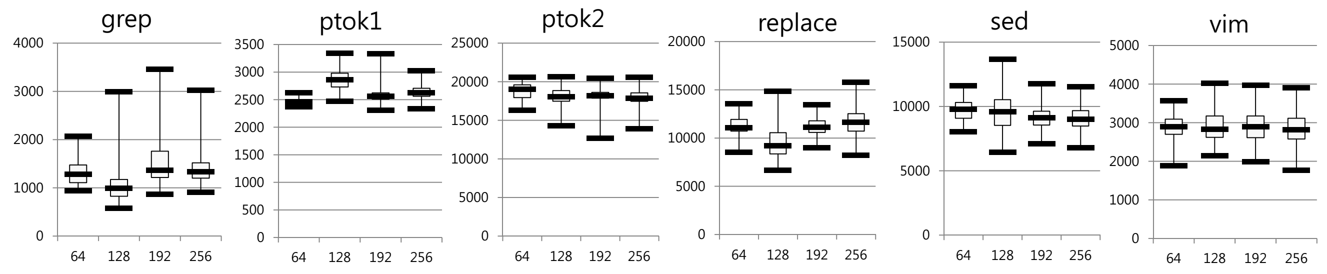

Figure 3. Distribution of numbers of test cases generated at each client node for the six object programs

a fault in grep 2.0 that made the program run in an infinite loop without generating output. This problem occurred when SCORE executed grep with a symbolic pattern containing '\n' as the first character and the -F option (i.e., using the 'fixed string matcher'). This fault was not detected by the SIR test cases. Also this fault is difficult to detect using manually created test cases, because it causes a failure only when such exceptional inputs are used. As a consequence, the fault had gone uncorrected for several years (until grep 2.4 was released).

Concolic Testing for Branch Coverage : In our experiment we generated test cases for path coverage, because path coverage is the target coverage of concolic testing. However, since branch coverage is more common in industrial practice, we also measured the percentages of branches covered in our study; we report these in Table VII. The branch coverages achieved did not increase much (and in two cases on ptok2, they decreased) as the number of clients used increased. We believe that this is a result of the concolic testing approach's focus on path coverage, which allow it to continue to explore new paths even though they do not cover new branches.

Table VII
BRANCH COVERAGE ACHIEVED (%)

	1	64	128	192	256	SIR
grep	24.1	44.1	44.0	45.0	46.3	50.3
ptok1	72.7	77.1	77.4	76.4	77.7	93.6
ptok2	78.7	81.3	80.4	81.8	81.5	98.2
repl	40.5	74.4	79.5	77.0	82.1	93.9
sed	21.7	25.4	27.0	27.9	27.8	47.3
vim	9.1	15.4	17.3	18.5	18.9	35.8

Note also that the branch coverages achieved in our experiment are lower than those achieved by the manually generated test cases in the SIR repository (rightmost column of the table). We believe that this too is related to the focus of the concolic approach on path coverage, though it may also be due to difficulties in solving constraints related to paths that reach particular branches. Nonetheless, given that the test cases for grep, sed, and vim found in the SIR repository were built over several weeks by students of the third author, and that the test cases for the other subjects were built from large pools of tests provided by the authors of [23], the mechanically achieved branch coverage attained by running SCORE for five minutes has value.

Limitations of Concolic Testing in Practice: As discussed earlier, concolic testing tools suffer from the limitations related to unavailability of library code, and the limitations of symbolic execution engines. Thus, in practice, concolic testing may not achieve full path coverage and may generate redundant test cases. To help examine this issue, Figure 4 shows the ratios of non-redundant test cases to total test cases generated for SCORE (S) (left) and multiple CREST runs (MC) (right).

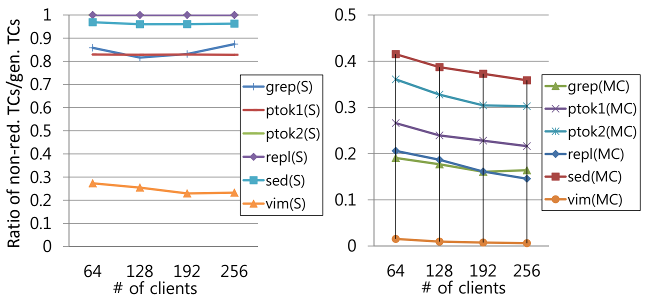

Figure 4. Ratios of non-redundant TCs over generated TCs

The ratios for SCORE are stable over increasing numbers of nodes but vary on different programs, since redundant test cases are generated when execution paths contain complex statements or external library calls. For example, ptok2 and replace (lines overlap in Figure 4) do not exhibit redundant test cases, but around 75% of the test cases generated for vim are redundant. This is because vim contains many uses of complex pointer arithmetic and uses external libraries to manipulate text strings. In contrast, the ratios for multiple CREST runs are lower than those for SCORE. In addition, the ratios decrease over increasing numbers of nodes. For example, on sed the ratios decrease from 0.42 (64 nodes) to 0.36 (256 nodes).

An additional limitation involves test oracles. Generating enormous numbers of test cases carries with it the complication that oracles are required for these test cases. In cases where oracles must be generated on a per test-case basis (e.g., creating "expected outputs" for each test case), this may involve excessive effort. Note that this problem arises with any large-scale automated test case generation effort. Therefore, to utilize automated test case generation frameworks to generate large numbers of test cases, we need to use oracle approaches whose cost is *not* proportional to the number of test cases. For example, we can utilize oracles that can be automated such as detecting system crashes and exceptional behavior or monitoring behavior for violations of user-given `assert` statements, which succeed in detecting many faults [3], [6], [11].

VI. Conclusion and Future Work

We have developed the SCORE framework to decrease the cost of concolic testing by utilizing a large number of distributed computing nodes. The framework enables distributed nodes to generate test cases independently, and in so doing it achieves scalability. We demonstrated the increased effectiveness of the framework, as well as its scalability, through an empirical study of several programs from the SIR repository. As future work, we intend to apply SCORE to additional applications, to analyze advantages and weakness of the framework in practice. Also, we plan to add fault-tolerant capability to SCORE.

References

[1] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *European Software Engineering Conference/Foundations of Software Engineering*, 2005.

[2] N. Tillmann and W. Schulte, "Parameterized unit tests," in *European Software Engineering Conference/Foundations of Software Engineering*, 2005.

[3] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Network and Distributed Systems Security*, 2008.

[4] M. Kim and Y. Kim, "Concolic testing of the multi-sector read operation for flash memory file system," in *Brazilian Symposium on Formal Methods*, 2009.

[5] C. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[6] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Operating System Design and Implementation*, 2008.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Intl. Symp. on Code Generation and Optimization*, 2004.

[8] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for Java," in *NASA Formal Methods Symposium*, 2009.

[9] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Automated Software Engineering*, Sep. 2000.

[10] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *International Symposium on Software Testing and Analysis*, 2008.

[11] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Programming Language Design and Implementation*, 2005.

[12] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-123, Sep 2008.

[13] K. Sen and G. Agha, "CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*, 2006.

[14] M. Staats and C. Pasareanu, "Parallel symbolic execution for structural test generation," in *International Symposium on Software Testing and Analysis*, 2010.

[15] A. King, "Distributed parallel symbolic execution," Kansas State University, Tech. Rep., 2009, MS thesis.

[16] J. H. Siddiqui and S. Khurshid, "ParSym: Parallel Symbolic Execution," in *International Conference on Software Technology and Engineering*, 2010.

[17] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *6th ACM SIGOPS/EuroSys*, 2011.

[18] X. Deng, J. Lee, and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *Automated Software Engineering*, 2006.

[19] Y. Kim, M. Kim, and N. Dang, "Scalable distributed concolic testing: a case study on a flash storage platform," in *Intl. Conf. on Theoretical Aspects of Computing*, 2010.

[20] M. Kim, Y. Kim, and G. Rothermel, "Uniqueness of the paths explored by the distributed concolic algorithm," KAIST, Tech. Rep., 2011, http://pswlab.kaist.ac.kr/publications/unique-proof.pdf.

[21] B. Dutertre and L. Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Computer Aided Verification*, 2006.

[22] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[23] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *International Conference on Software Engineering*, 1994, pp. 191–200.

[24] "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2/.

[25] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.