SCORE: a Scalable Concolic Testing Tool for Reliable Embedded Software

Yunho Kim and Moonzoo Kim
Computer Science Department
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

ABSTRACT

Current industrial testing practices often generate test cases in a manual manner, which degrades both the effectiveness and efficiency of testing. To alleviate this problem, concolic testing generates test cases that can achieve high coverage in an automated fashion. One main task of concolic testing is to extract symbolic information from a concrete execution of a target program at runtime. Thus, a design decision on how to extract symbolic information affects efficiency, effectiveness, and applicability of concolic testing. We have developed a Scalable COncolic testing tool for REliable embedded software (SCORE) that targets embedded C programs. SCORE instruments a target C program to extract symbolic information and applies concolic testing to a target program in a scalable manner by utilizing a large number of distributed computing nodes. In this paper, we describe our design decisions that are implemented in SCORE and demonstrate the performance of SCORE through the experiments on the SIR benchmarks.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

1. 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. An alternative approach, concolic (CONCrete + symbOLIC) [18] testing (also known as dynamic symbolic execution [20] or white-box fuzzing [8]) 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).

A concolic testing tool extracts symbolic information from a concrete execution of a target program at runtime. In other words, to build a corresponding symbolic path formula, a concolic testing tool monitors every update of symbolic variables and branching decisions in an execution of a target program. Thus, a design decision on how to extract symbolic information affects efficiency, effectiveness, and applicability of concolic testing.

We have developed a Scalable COncolic testing tool for REliabile embedded software (SCORE) that targets sequential embedded C programs. SCORE instruments a target C source code by inserting probes to extract symbolic information at runtime. This concolic testing approach has several advantages when applied to embedded C programs compared to a modified virtual machine approach (Section 2). In addition, to reduce a time cost to explore a large number of execution paths, SCORE utilizes a scalable distributed concolic algorithm [12] that involves a large number of computing nodes dynamically with high efficiency (Section 3). Currently, SCORE is implemented on the Amazon EC2 cloud computing platform [6]. In this paper, we describe our design decisions that are implemented in the SCORE tool for applying concolic testing in a practical and scalable manner to embedded C programs. In addition, we demonstrate the performance of SCORE through the experiments on the SIR benchmarks

2. INSTRUMENTATION-BASED CONCOLIC TESTING FOR EMBEDDED C PROGRAMS

A 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 [16] for a survey). Existing tools can be classified into two groups 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 (VM). The concolic testing tools that use this approach are implemented as modified VMs on which target programs execute. An advantage of this approach is that the tools can exploit all execution information at run-time, since a VM possesses all necessary information. PEX [20] targets C# programs that are compiled into Microsoft .Net bytecode, KLEE [4] targets LLVM [15] bytecode, and jFuzz [10] targets Java bytecode on top of Java PathFinder [21].

The second approach for extracting symbolic path formulas is to instrument the target source code to insert probes that ex-

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09 ...$10.00.

tract symbolic path formulas from concrete executions at runtime. Tools that use this approach include CUTE [18], DART [7], and CREST [3], which operate on C programs, and jCUTE [17], which operates on Java programs. This approach is more suitable for embedded C programs compared to the first approach, since embedded software often has different development/runtime environments from those of non-embedded software due to limited computational power and physical constraints. Advantages of the instrumentation-based concolic testing for embedded C programs are as follows:

1. Embedded software is often developed for a specific hardware/OS platform (or tested on a customized simulation environment such as Scratchbox [1]) (see [13][1]). Therefore, the first approach using a VM requires to port a modified VM that supports concolic testing to a specific hardware/simulator platform, which is expensive.

2. An embedded program often contains native binary code to access hardware devices directly, which cannot be interpreted using the VM approach. Most concolic testing tools of the first approach simply stop/ignore a current symbolic execution when they encounter such native binary code and try another execution path, which can miss bugs related with low-level I/O operations that are often the focus of embedded programs. Although the second approach cannot symbolically interpret native binary code either, it still executes such binary code and continues to analyze an execution path using concrete values.

3. A target program written in C can be compiled into a binary executable program that can run on hardware directly. A binary executable program that is compiled from a C source code can run order of magnitude faster than the corresponding bytecode runs on a VM. Thus, this second approach can test an embedded C program much faster than the first approach does, which is an important advantage, since concolic testing takes a large amount of time.

Therefore, SCORE utilizes the second approach for embedded C programs. As shown in Figure 1, a target C program is instrumented by the SCORE instrumentor (implemented in CIL with an extension module). To alleviate difficulty of extracting symbolic information from a concrete execution, SCORE transforms a target C program into an equivalent one that has no side-effect and a single atomic predicate at each conditional statement (i.e., every conditional statement with a compound predicate is transformed into combination of conditional statements with atomic predicates). Then, SCORE inserts probes into a target C program that invoke symbolic execution library to extract symbolic information to build a corresponding symbolic path formula at run-time. Lastly, the instrumented target program is compiled into an executable binary through gcc.

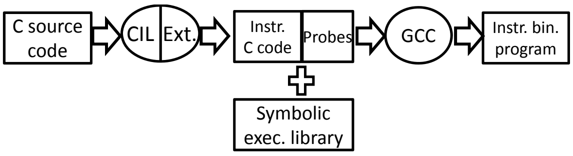

Figure 1: Instrumentation Process of SCORE

3. DISTRIBUTED CONCOLIC TESTING FRAMEWORK

3.1 Overview

SCORE consists of one server, multiple clients, and a user interface program through which a user can control SCORE. For efficient communication between computing nodes, SCORE adopts a hybrid architecture of a client-server architecture and a peer-to-peer architecture by utilizing a central server that handles control messages to keep track of the status of the clients and to direct load-balancing between the clients (i.e., test cases are transferred between clients in a well-balanced manner) (see Figure 2).

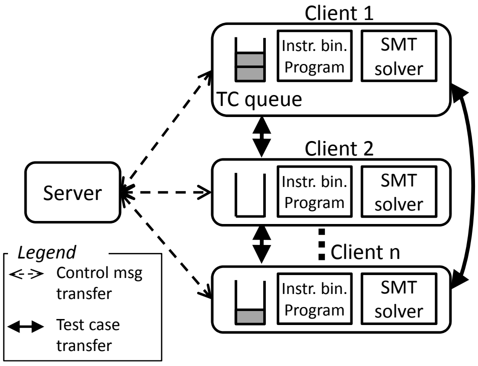

Figure 2: SCORE architecture

Each client has its own copy of an instrumented target binary program (possibly with a platform simulator) and applies concolic testing to the target program to generate test cases, which are stored in a test case queue. After a client pops a test case from the queue, it generates further test cases by solving symbolic path formulas that are obtained by negating branch conditions of the execution path on the test case. If a client has an empty test case queue, it requests new test cases from the server and the server commands another client that has test cases to send test cases to the requesting client. [2] Although SCORE utilizes a large number of computing nodes, a user can manage these nodes conveniently through the automated initialization (Section 3.2) and dynamic management of nodes (Section 3.3) features.

There are several concolic testing tools utilizing distributed platforms such as Staats et al. [19], King [14] and Cloud9 [2], all of which utilize a VM-based approach. However, these tools do not consider characteristics of embedded software and they are not adequate for embedded software due to the reasons explained in Section 2. In addition, except Cloud9, these tools do not achieve linear speed-up in terms of a number of computing nodes.

3.2 Initialization of Distributed Clients

Before starting distributed concolic testing, SCORE automatically instantiates virtual nodes and installs copies of an instrumented target binary program on the virtual nodes. This automated initialization process is desirable, since SCORE utilizes virtual nodes of 'spot-instance' type of Amazon EC2 whose storages are destroyed when virtual nodes are turned off. SCORE utilizes virtual nodes of spot-instance type, because a price for virtual node of spot-instance is cheaper (US$ 0.06/hour) than a price for a virtual node of persistent storage (US$ 0.17/hour). In addition, only

[1] We could not apply SCORE to the industrial project with Samsung Electronics, since we were not allowed to utilize Amazon EC2 due to the security policy of Samsung Electronics.

[2] The distributed concolic algorithm of SCORE, detailed communication protocol, and complete experimental results can be found in [11].

20 virtual nodes of persistent storage type can be instantiated in one user account while 100 virtual nodes of spot-instance type can be instantiated in one user account.

For this automated initialization, we have built an Amazon AMI (Amazon Machine Instance) for SCORE, which is used to instantiate virtual node. Amazon AMI for SCORE is based on a 32 bit minimal version of Fedora Core 8 and contains basic development libraries, a ftp client (to download an instrumented target program), a SMT solver (Z3), and a monitoring tool (to measure CPU usage and network traffic).

Distributed concolic testing is initiated by invoking a SCORE server with an instrumented binary target program. The server instantiates clients and establishes network connections to the clients. Then, the clients download the target program from the server and start distributed concolic testing. Although the clients can share the target program on Network File System (NFS) without downloading the target program from the server, it will cause significant overhead due to a large number of concurrent network accesses to the target program on NFS.

3.3 Dynamic Management of Nodes

SCORE can add and remove clients dynamically at run-time. This dynamic resource management feature is useful, because SCORE utilizes a large number of computing nodes. This dynamic management feature supports flexible management of multiple testing tasks and thus assigns computing resources to multiple testing tasks in a globally efficient manner. In addition, flexible management of multiple testing tasks is beneficial in an economic sense. For example, the price for using computing nodes at night may be cheaper than the price during the day (this is true for Amazon EC2's spot-instance computing nodes, which are purchased by bidding).

For example, suppose that a user applies concolic testing to a target program using SCORE. The user commands a server to assign n clients to the testing task initially. In addition, during the testing process, SCORE allows a user to both increase or decrease n at runtime. To add a new client, the server instantiates a new computing node through the underlying Amazon EC2 API and remotely starts a client on the node. The client begins to participate in the testing task by sending a request packet to the server. To remove a client, a user can specify a number of clients to remove, or conditions by which to select a client such as the number of test cases in the client, execution time spent so far, and so on. For example, suppose a user requests removal of all clients that have fewer than five test cases. The server searches the status table of all clients and selects those that satisfy the condition. For each such client, the server sends a termination packet to the client. Then, the client transfers test cases generated so far, covered path/branch information, and statistics on testing activities to another designated client.

3.4 Non-redundant Test Case Generation

SCORE uses a depth-first search (DFS) strategy to explore a symbolic execution tree. In theory, the DFS strategy explores one execution path exactly once; it would generate redundant test cases (i.e., test cases running a same execution path) otherwise. However, when a target program contains external library calls and floating-point arithmetics that cannot be reasoned by SMT solvers, the DFS strategy may execute the same path multiple times in practice.

For example, suppose that a target program has `if(x == abs(x)) exit(); else {...}` where `abs()` is an external C library function. The program will terminate if a test case TC1 (`x=0`) is given. Then, SCORE negates this branch condition to generate TC2 to explore the `else` branch. Since the external library invocation `abs(x)` is *concretized* to its current concrete return value (i.e., 0), the negated symbolic path formula is `x != 0`. Although a new test case TC2 (`x=1`) is generated by solving the negated formula, TC2 executes the same execution path explored by TC1. Note that if a newly generated test case is redundant, all test cases that are derived from it can be also redundant. Thus, SCORE detects such redundant test cases and does not generate further test cases from these redundant test cases.

To detect such redundant test cases, SCORE checks whether a newly generated test case executes a *predicted* execution path $c_1 \wedge c_2 ... \wedge c_{k-1} \wedge \neg c_k$ (c_i is a branch condition and c_i is executed immediately before c_{i+1}) that has common prefix (i.e., $c_1 \wedge ... c_{k-1}$) with the previous execution path until the negated branch condition (i.e., $\neg c_k$). For this purpose, SCORE keeps information on the previous execution path, since this prediction test can be done by comparing a current execution path and the previous execution path. [3] This prediction test applies to the distributed clients by transferring test cases with common prefixes of their previous execution paths (i.e., $c_1 \wedge c_2 ... \wedge c_{k-1}$) and negated branch conditions (i.e., c_k).

3.5 Implementation

SCORE is written in C/C++ and contains 7600 lines of code with 24 classes and 265 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 test cases through concolic testing. The other thread communicates with other computing nodes to receive/send test cases or control messages. 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 requests to stop the concolic testing, these outcomes are collected by the server automatically.

SCORE is implemented to operate on distributed computers connected through TCP/IP networks, since the framework may be deployed on a large scale computing platform such as cloud computing platforms or P2P networks where communication might not be reliable. SCORE uses CREST 0.1.1 [3] to instrument a target C program and to obtain symbolic formulas from concrete execution paths at run-time. However, we have extended the instrumentation tool and the symbolic execution engine of CREST to support symbolic path formulas in bit-vector theory while the original CREST supports only linear-integer arithmetic symbolic path formulas. In addition, SCORE uses Z3 2.15 as a underlying SMT solver instead of Yices (a SMT solver used by CREST), since Z3 provides richer C APIs for bit-vector theory than Yices does (for example, Z3 provides C API for bit-vector division and modular operators, which are not directly supported by C API of Yices) and user community are more active.

3.6 Experimental Results

To demonstrate efficiency and effectiveness of SCORE, we applied SCORE to well-known benchmark programs. As objects of experiments, we selected six programs from the SIR repository [5], including three of the Siemens programs [9], and three non-trivial real-world programs (`grep` 2.0, `sed` 1.17, and `vim` 5.0). [4]

All experiments were performed on the Amazon EC2 cloud computing platform. The server of the SCORE framework ran on a virtual node that had 7GB of memory and 8 CPU cores of 20 ECU computing power in total (1 ECU is equivalent to a 1Ghz Xeon pro-

[3] A test case that exercises an execution path that does not match a predicted execution path may not be a redundant test case. However, since such a test case can be a redundant one, we conservatively consider the test case as a redundant one.

[4] The experimental results are taken from [11]

cessor). Each client ran on a virtual node that was equipped with 1.7 GB memory and 2 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.

For each target program, we executed CREST on a node and SCORE on each of the five client number levels for 5 minutes. To control for potential differences in runs due to the randomization inherent in the techniques, we repeated the experiments 30 times and reported the mean value of 30 runs.

We begin by comparing SCORE to CREST. In these experiments, we set SCORE to use linear-integer arithmetic symbolic path formulas for fair comparison to CREST. Figure 3 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 tools. The figure compares effectiveness results obtained by SCORE at all five client number levels to the results obtained by CREST, 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.

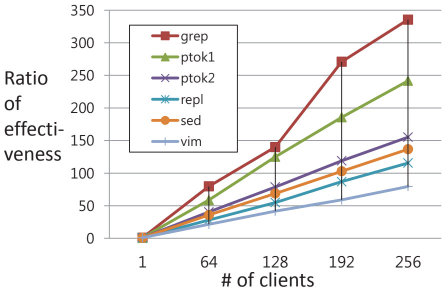

Figure 3: Ratio of effectiveness between CREST and SCORE, per client number level

4. CONCLUSION AND FUTURE WORK

We have developed the SCORE framework that targets embedded C programs to decrease the time 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. In addition, SCORE utilizes instrumentation-based approach that is suitable for testing embedded C programs. We demonstrated linear speedup of SCORE on distributed systems through the experiments.

As future work, we intend to apply SCORE to industrial embedded C programs, to analyze advantages and weakness of the framework in practice. In addition, we plan to develop additional distributed concolic algorithms that aim higher branch coverage in a fast manner instead of path coverage pursued by DFS, which may facilitate its earlier adoption in industrial software development. Finally, we plan to release SCORE as an open-source project for researchers and testing practitioners to utilize SCORE in their research platforms and projects freely and to improve SCORE with support from open source community.

Acknowledgements

This research was partially supported by the ERC of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea) (Grant 2011-0000978), and Basic Science Research Program through the NRF funded by the MEST (2010-0005498). We would like to thank Matt Staats for his feedback and comments.

5. REFERENCES

[1] Scratchbox - cross-compilation toolkit. `http://www.scratchbox.org/`.

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

[3] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.

[4] 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.

[5] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering Journal*, 10(4):405–435, 2005.

[6] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

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

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

[9] 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*, pages 191–200, 1994.

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

[11] M. Kim, Y. Kim, and G. Rothermel. A scalable distributed concolic testing approach. In *Automated Software Engineering*, 2011. under review.

[12] 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.

[13] Y. Kim, M. Kim, and Y. Jang. Concolic testing on embedded software - case studies on mobile platform programs. In *Foundations of Software Engineering (FSE)*, 2011.

[14] A. King. Distributed parallel symbolic execution. Technical report, Kansas State University, 2009. MS thesis.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation, 2004.

[16] C. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Software Tools for Tech. Transfer*, 11(4):339–353, 2009.

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

[18] 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.

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

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

[21] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Eng.*, 2000.