Experimental Comparison of Concolic and Random Testing for Java Card Applets*

Kari Kähkönen, Roland Kindermann, Keijo Heljanko, and Ilkka Niemelä

Aalto University, Department of Information and Computer Science
P.O. Box 15400, FI-00076 AALTO, Finland
{Kari.Kahkonen, Roland.Kindermann, Keijo.Heljanko,
Ilkka.Niemela}@tkk.fi

Abstract. Concolic testing is a method for test input generation where a given program is executed both concretely and symbolically at the same time. This paper introduces the LIME Concolic Tester (LCT), an open source concolic testing tool for sequential Java programs. It discusses the design choices behind LCT as well as its use in automated unit test generation for the JUnit testing framework. As the main experimental contribution we report on an empirical evaluation of LCT for testing smart card Java applets. In particular, we focus on the problem of differential testing, where a Java class implementation is tested against a reference implementation. Two different concolic unit test generation approaches are presented and their effectiveness is compared with random testing. The experiments show that concolic testing is able to find significantly more bugs than random testing in the testing domain at hand.

1 Introduction

This paper discusses the use of concolic testing [1–6] to generate tests for Java applets written for the Sun Java Card platform [7, 8]. In particular, we consider a new open source concolic testing tool we have developed called the LIME Concolic Tester (LCT) which is included in the LIME test bench toolset (`http://www.tcs.hut.fi/Research/Logic/LIME2/`). The tool can automatically generate unit test data and also stub code needed for unit testing in the JUnit testing framework. The main improvements in LCT over existing Java concolic testing systems such as jCUTE [2] are the following: (i) the use of state-of-the art bitvector SMT solvers such as Boolector [9] make the symbolic execution of Java more precise, (ii) the twin class hierarchy instrumentation approach of LCT allows the Java base classes to be instrumented unlike in previous approaches such as jCUTE, (iii) the tool architecture supports distributed testing where the constraint solving is done for several tests in parallel in a distributed manner, (iv) the tool is integrated with runtime monitoring of interface specifications [10] and the runtime monitors are used to guide the test generation order; and (v) the tool is freely available as open source. Distributed constraint solving has been previously employed by the Microsoft Whitebox fuzzing tool SAGE [11, 12] that uses a distributed

* Work financially supported by Tekes - Finnish Funding Agency for Technology and Innovation, Conformiq Software, Elektrobit, Nokia, Space Systems Finland, and Academy of Finland (projects 126860 and 128050).

constraint solver Disolver while LCT uses a non-distributed constraint solver but can work on several branches of the symbolic execution tree in parallel.

We are interested in testing embedded software and, in particular, Java Card applets. In this paper we focus on the following *differential testing* scenario. An applet is being developed and a set of modifications has been made to it that should not change its class behavior, i.e., method calls to classes should behave as in the original version. Such modifications could be optimizations, changes in the internal data structures, refactoring of the code, clean ups removing redundant code etc. Hence, we have two versions of the same class which we call the *reference implementation* (the original version) and the implementation under test IUT (the modified version). Now the problem is to test whether the reference implementation and IUT have the same class behavior. In practice, the testing process starts by identifying modified classes and then it boils down to treating each modified class as an IUT and testing it against the corresponding original class taken as the reference implementation. Random testing techniques are often quite successful in such unit testing settings. In this paper we study how concolic testing and, in particular, the LCT tool can be used for this testing scenario. We develop two approaches to using concolic testing for checking whether an IUT of a class has the same class behavior as its reference implementation. Then we study experimentally how well the test sets generated by LCT using concolic testing techniques are able to detect differences in class behavior when compared to random testing.

The main contribution of this paper is the experimental work comparing the concolic testing approach to random automated testing. The context is Java Card applets designed for the Java Card smart card platform. In the experiments we compare the bug detection capabilities of both concolic and random testing by using a smart card application called the Logical Channel Demo [8] as the reference implementation. In order to provide a sufficient number of buggy implementations to serve as IUTs in the experiments, a Java source code mutant generator tool is used to provide mutated versions of the reference implementation. The experimental setup consists of a large number of experiments where a buggy mutated implementation is tested against the reference implementation. The results clearly show that the concolic testing approach is more effective in finding bugs than random testing.

There has been significant earlier work in experimentally evaluating concolic testing [1–6]. Hybrid concolic testing [13] interleaves random testing with concolic execution, and the experiments in [13] report on four times higher branch coverage with hybrid concolic testing compared to random testing on red-black trees and on the text editor vim. Experimental research on Whitebox fuzz testing efficiency [14, 11, 12] have used quite a different experimental test setup than we have in this paper and, thus, the results are hard to directly compare. However, also in the Whitebox fuzz testing context techniques based on symbolic execution techniques seem to be able to find many bugs missed by fully randomized testing.

The structure of the rest of this paper is as follows. Section 2 introduces concolic testing while the design choices done in the design of the LCT concolic testing tool are discussed in Sect. 3. Section 4 introduces differential testing, a method for comparing the behavior of two implementations of a class, and describes how LCT can be used for

differential testing. Section 5 discusses our experimental setup for applying differential testing to Java Card applets. Finally, Sect. 6 sums up the paper.

2 Concolic Testing

Concolic testing [1–6] (also known as dynamic symbolic execution) is a method for test input generation where a given program is executed both concretely and symbolically at the same time. In other words, the test inputs are generated from a real executable program instead of a model of it. The main idea behind this approach is to at runtime collect symbolic constraints on inputs to the system that specify the possible input values that force the program to follow a specific execution path. Symbolic execution of programs is made possible by instrumenting the system under test with additional code that collects the constraints without disrupting the concrete execution.

In concolic testing, each variable that has a value depending on inputs to the program has also a symbolic value associated to it. When a sequential program is executed, the same execution path is followed regardless of the input values until a branching statement is encountered that selects the true or false branch based on some variable that has a symbolic value. Given the symbolic value of this variable, it is possible to reason about the outcome of the statement symbolically by constructing a symbolic constraint. This constraint describes what the possible input values are that cause the program to take the true or false branch at the statement in question. A *path constraint* is a conjunction of symbolic constraints that describes the input values that cause the concrete execution to follow a specific execution path.

In concolic testing the program under test is first executed with concrete random input values. During this test run, symbolic execution is used to collect the path constraints expressed in theory T for each of the branching statements along the execution. These collected constraints are used to compute new test inputs to the program by using off-the-shelf constraint solvers. Typical solvers used in concolic testing are SMT (Satisfiability-Modulo-Theories) solvers such as Yices [15], Boolector [9] and Z3 [16] and typical theories include linear integer arithmetic and bit-vectors. The new test inputs will steer the future test runs to explore previously untested execution paths. This means that concolic testing can be seen as a method that systematically tests all the distinct execution paths of a program. These execution paths can be expressed as a *symbolic execution tree* which is a structure where each path from root to a leaf node represents an execution path and each leaf node has a path constraint describing the input values that force the program to follow that specific path.

The concrete execution in concolic testing brings the benefit that it makes available accurate information about the program state which might not be easily accessible when using only static analysis. It is possible to under-approximate the set of possible execution paths by using concrete values instead of symbolic values in cases where symbolic execution is not possible (e.g., with calls to libraries to which no source code is available). Furthermore, as each test is run concretely, concolic testing does not report spurious defects.

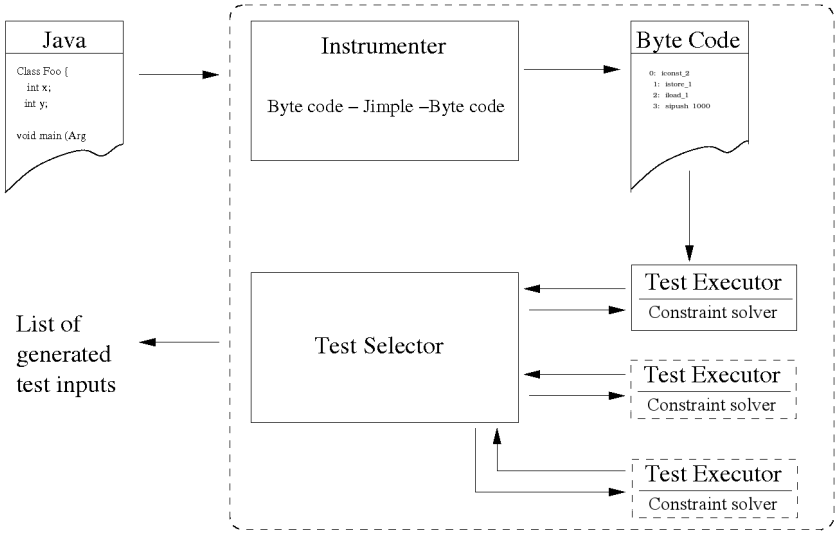

Fig. 1. The architecture of LCT

3 LIME Concolic Tester

The LIME Concolic Tester (LCT) is an open source test input generator for Java programs that is based on concolic testing. It takes a sequential Java program that has been compiled into bytecode as input and generates tests that attempt to cover all the execution paths of the program. LCT supports multiple search strategies which affect the order in which the execution paths are explored. During the testing process uncaught exceptions are reported as defects.

LCT is a part of the LIME Interface Test Bench developed in the LIME project (`http://www.tcs.hut.fi/Research/Logic/LIME2`). In the LIME project an interface specification language and the supporting LIME Interface Monitoring Tool (LIMT) [17] have also been developed and LCT allows the monitoring tool to guide the testing process in order to cover the specifications quickly. More details of this is given in Sect. 3.4.

As different test runs do not depend on each other, the problem of generating test inputs is easily parallelized. LCT takes advantage of this fact by having a separate test server that receives the symbolic constraints from the instrumented programs and selects which unexplored execution paths are tested next. This allows the tool to take advantage of multi-core processors and networks of computers.

The rest of this section discusses the implementation of LCT in more detail.

3.1 Architectural Overview

The architecture of LCT is shown in Figure 1 and it can be seen as consisting of three main parts: the instrumenter, the test selector and the test executors which are also used

to run the constraint solvers in a distributed fashion. The instrumenter is based on a tool called Soot [18] which can be used to analyze and transform Java byte code. Before a program is given to the instrumenter, the input locations in the source code are marked so that the instrumenter knows how to transform the code. LCT provides a static class that is used for this in the following fashion:

- `int x = LCT.getInteger()` gets an int type input value for `x`, and
- `List l = LCT.getObject("List")` indicates that `l` is an input object.

After the input variables have been marked in the source code, the program is given to the instrumenter that transforms the code into an intermediate representation called Jimple and adds the statements necessary for symbolic execution into it. When the instrumentation is finished, the code is transformed into byte code that can be run over a standard Java Virtual Machine. This modified version of the program is called test executor. To guarantee that every test execution terminates, the length of the concrete execution is limited by a user configurable depth limit.

The test selector is responsible for constructing a symbolic execution tree based on the constraints collected by the test executors and selecting which path in the symbolic execution tree is explored next. The communication between the test selector and test executors has been implemented using TCP sockets. This way the test selector and test executors can run on different computers, and it is easy to run new test executors concurrently with others. The test selector can use various search strategies for selecting the next execution path to be explored. It is also possible to use LCT in a random mode, where no symbolic execution tree is constructed and the input values are generated completely randomly.

LCT provides the option to use Yices [15] or Boolector [9] as its constraint solver. In case of Yices, LCT uses linear integer arithmetic to encode the constraints and in the case of Boolector, bit-vectors are used. LCT has support for all primitive data types in Java as symbolic inputs with the exception of float and double data types as there is no native support for floating point variables in the used constraint solvers. We are currently targeting embedded software with limited data type support and better handling of complex data is part of future research. At the moment LCT can generate input objects that have their fields initialized as new input values in a similar fashion to [2].

3.2 Instrumentation

After the input locations have been marked in the source code, adding the code for symbolic execution to a given program can be made in a fully automatic fashion. To unit test a method, the user can, for example, write a test driver that calls the method to be tested with symbolic input values. LCT also supports generating such test drivers automatically not only for unit testing of methods but also for testing interfaces of classes through sequences of method calls.

To execute a program symbolically every statement that can read or update a variable with its value depending on the inputs must be instrumented. The approach taken in LCT is to instrument all statements that could operate on symbolic inputs regardless of whether a statement operates only with concrete values during test runs or not.

This means that a majority of the lines in the code will be instrumented. LCT uses the Soot framework [18] to first translate the program under test to an intermediate language which is then modified and transformed back into bytecode. A similar approach is taken in jCUTE [3].

To make symbolic execution possible, it is necessary to know for each variable the associated symbolic expression during execution. For this reason we use a symbolic memory *S* that maps primitive type variables and object references to symbolic expressions. We also need to construct the path constraints for unexplored execution paths. The approach taken in LCT to construct the memory map and path constraints follows closely that described in [1, 2]. To update the symbolic memory map, every assignment statement in the program is instrumented. At assignment of type $m = e$, where e is an expression, the symbolic value $S(e)$ is constructed and the mapping $S(m)$ is updated with this value. In case of new input values, $m = \texttt{INPUT}$, a new symbolic value is assigned to $S(m)$. Every branching statement, e.g., $\texttt{if}(p)$, must also be instrumented. Symbolic evaluation of p and its negation are the symbolic constraints that are used to construct the necessary path constraints. A more detailed description of the instrumentation process can be found in [19].

3.3 Search Strategies

The LCT tool contains a number of search strategies to control the exploration order of the different branches of the symbolic execution tree. The techniques are traditional depth-first and breadth-first search, priority based search using heuristic values obtained from the runtime monitors written in the LIME interface specification language, as well as randomized search. As this paper focuses on test generation methods that explore the full symbolic execution tree, the order in which branches are explored makes little difference; hence, we do not discuss these strategies here. Further details can be found in [19].

3.4 Test Generation for Programs with Specifications

The described concolic testing method reports uncaught exceptions as errors (i.e., it generates tests to see if the program can crash). This testing approach can be greatly enhanced if it is combined with runtime monitoring to check if given specifications hold during the test runs. In the LIME project a specification language has been developed together with a runtime monitoring tool [10] that allows the user to use propositional linear temporal logic (PLTL) and regular expressions to specify both external usage and internal behavior of a software component [10, 17].

The LIME interface specification language allows the user to write specifications that are not complete models of the system and to target the specifications to those parts of the systems that are seen as important to be tested. Also, the specifications provide additional information about the system that could be used to indicate when the program is close to a state that violates the specifications. We have extended the concolic testing method to take the LIME interface specifications into account so that the testing can be guided towards those execution paths that cause specifications to be monitored

and especially towards those paths that can potentially cause the specifications to be violated.

To guide the concolic testing, the LIME Interface Monitoring Tool (LIMT) [10, 17] has been extended to compute a heuristic value to indicate how close the current execution is to violating the monitored specifications. In LCT the instrumentation described earlier has been augmented with a call to LIMT to obtain the heuristic value at every branching statement where a symbolic constraint is constructed. The test input selector is then notified about the heuristic value and it can use the value to assign a priority to the unvisited node resulting from executing the branching statement. Further details can be found in [19].

3.5 Generating JUnit Tests

LCT also provides the possibility to generate JUnit tests based on the input values generated during concolic testing. The support for JUnit tests is limited to unit testing methods that take argument values that can be generated by LCT. To generate JUnit tests for a selected method, LCT first creates automatically a test driver that calls the method with input values computed by LCT. The generated input values are then stored and used to generate JUnit tests that can be executed even without LCT. It is also possible to generate a test driver for an interface. In this case LCT generates a test driver that calls nondeterministically methods of that interface with symbolic input arguments. The number of calls is limited by an user specified call sequence bound.

3.6 Limitations

The current version of LCT has been designed for sequential Java programs and multi-threading support is currently under development. Sometimes LCT can not obtain full path coverage for supported Java programs. This can happen in the following situations: (i) *Non-instrumented code*: LCT is not able to do any symbolic reasoning if the control flow of the program goes to a library that has not been instrumented. This is possible, for example, when libraries implemented in a different programming language are called. To be able to instrument Java core classes, we have implemented custom versions of some of the most often required core classes to alleviate this problem. The program under test is then automatically modified to use the custom versions of these classes instead of the original counterpart. This approach can be seen as an instance of the twin class hierarchy approach presented in [20]. (ii) *Imprecise Symbolic reasoning*: Symbolic execution is limited by the ability of constraint solvers to compute new input values. LCT does not currently collect constraints over floating point numbers. LCT also does an identical non-aliasing assumption which the jCUTE tool does as well. The code "`a[i] = 0; a[j] = 1; if (a[i] == 0) ERROR;`" is an example of this. LCT assumes that the two writes do not alias and, thus, does not generate constraint $i = j$ required to reach the `ERROR` label; (iii) *Nondeterminism*: LCT assumes that the program under test and the libraries it uses are deterministic.

4 Differential Testing

Often, a software developer modifies source code in situations where the behavior of the code should not change, e.g., when cleaning up or refactoring code or when implementing a more optimized version of a given piece of code. Differential testing (terminology strongly influenced by [21]) is a technique that searches for differences in the behavior between the original code and the code after such modifications.

The basic idea behind differential testing is to compare the externally visible behavior of two implementations of the same class. Usually one of the implementations, called the reference implementation, is trusted to be correct. The task is then to test, whether the other implementation, called implementation under test (*IUT*), is correct as well by searching for differences in the behaviors of the two implementations.

For comparing the behaviors of two implementations, a black-box notion of class equivalence is used. Only the externally observable behavior of method calls is considered. Thus, two method calls are defined to have equivalent behavior if the following conditions are fulfilled:

1. Either both calls throw an exception or neither of them throws an exception.
2. If neither call throws an exception, they return equivalent values.
3. If both methods throw an exception, they throw equivalent exceptions.

The exact definition of equivalence of return values and exceptions may depend on the context in which differential testing is used. For example, return values in Java could be compared using the `equals` method. Sometimes, however, this notion of equivalence might be insufficient. Floating point values, for instance, could be considered equivalent even if they are not exactly the same as long as their difference is small enough. The definition of equivalent behavior used in this work only takes return values and thrown exceptions into account. In a more general setting, also other effects of methods calls that can be observed from the outside, like modified static variables, fields of the class that are declared public or method arguments that are modified, could be compared as well for a more refined notion of behavioral equivalence.

Two sequences of method calls to a class are considered to show equivalent class behavior if the behavior of the nth call in one sequence is equivalent to the behavior of the nth call in the other sequence. Two implementations of a class are considered to be *class behavior equivalent* if every sequence of method calls on a fresh instance of one of the implementations shows equivalent class behavior as the same sequence of method calls on a fresh instance of the other implementation. Determining whether two classes are class behavior equivalent is quite challenging as very long sequences of method calls might be needed to show non-equivalent behavior of the classes. In order to circumvent this difficulty, the notion of *k-bounded class behavior equivalence* is introduced. Two implementations of a class are considered to be k-boundedly class behavior equivalent if every sequence of at most k calls on a fresh instance of one of the implementations shows equivalent class behavior as the same sequence of calls on a fresh instance of the other implementation.

In the following, two concolic-testing-based and one random-testing-based technique for checking for bounded class behavior equivalence are introduced. All three

approaches are symmetric in the sense that they treat the IUT and the reference implementation in the same way.

Decoupled differential testing The basic idea behind using LCT for checking a pair of an IUT and an reference implementation for k-bounded class behavior equivalence is to let LCT generate sequences of method calls of length k and compare the behavior of the IUT and the reference implementation based on those sequences of method calls. The most obvious way to do this is to let LCT generate a test set for the IUT and another one for the reference implementation independently. Each test in the test sets consists of a sequence of method calls that can then be used for comparing the behavior of the IUT and the reference implementation. As LCT in this approach generates test sets for the IUT and the reference implementation individually, this approach is referred to as *decoupled differential testing*.

LCT is in the first step of the decoupled differential testing approach used to generate two test sets with call sequence bound k – one for the IUT and one for the reference implementation. Each test generated in this way consists of a sequence of methods calls to one of the implementations. The tests, however, do not evaluate the behavior of the method calls in any way. In the second step, the tests are therefore modified. Each test in the test set for the IUT is modified in a way such that each method that is executed on the IUT is executed on the reference implementation as well and an exception is thrown if and only if the behaviors of such a pair of method calls are non-equivalent. Calls to methods of the IUT are added to the test set of the reference implementation in the same way. In the last step of the decoupled differential testing approach, these modified tests are executed and an error is reported if any test signals non-equivalent behaviors.

In the decoupled differential testing approach, test sets for both the IUT and the reference implementation are generated. It would also be possible to only generate and use tests for either the IUT or the reference implementation. This approach, however, would make it quite likely that some classes of errors are missed. The developer of the IUT might, for instance, have forgotten that some input values need special treatment. In such a case, the `if` condition testing for these special values would be missing in the IUT and the LCT could achieve full path coverage without ever using any of the special input values, which would result in the described bug not being found by the generated test set. Therefore, a bug can easily be missed when only the test set for the IUT is used while it is found if the test set for the reference implementation is included, assuming the LCT reaches full path coverage when generating the test set for the reference implementation. A bug introduced by a developer who adds optimizations specific to a limited range of special input values could for similar reasons be missed if only the test set for the reference implementation but not the one for the IUT is used.

The main disadvantage of decoupled differential testing is that there may be situations in which non-equivalent behavior remains undetected even if full path coverage is reached on both the IUT and the reference implementation. A simple example of such a situation is a method that takes one integer argument and returns that argument in the reference implementation but returns the negation of the argument in the IUT. This difference in class behavior does not show if the method is called with zero as argument. Still, LCT can reach full path coverage on both implementations without using any other

argument than zero. Thus, the non-equivalent behavior may remain undetected even if LCT reaches full path coverage on both implementations. This motivates coupled differential testing, which circumvents the described issue by checking for non-equivalent behaviors in the code that is instrumented and run by LCT.

Coupled differential testing In the coupled differential testing approach, LCT is directly run on a class that compares the behavior of the IUT and the reference implementation. This comparison class has one instance each of the IUT and the reference implementation and has one method for every externally visible method of the IUT and the reference implementation. Each method in the comparison class calls the corresponding methods in the IUT, and the reference implementation, compares their behaviors and throws an exception if and only if they are not equivalent. Therefore, a sequence of method calls on the comparison class can be executed without an exception being thrown if and only if the IUT and the reference implementation show equivalent class behavior for this sequence of calls.

The comparison class is generated in the first step of the coupled differential testing approach. LCT is then used in the second step to generate a test set for the comparison class. Finally, the tests are executed and non-equivalent behavior is reported if any of the tests in the test set throws an exception. Thus, the main difference between coupled and decoupled differential testing is, that the behavior comparison code is added before LCT is run in the coupled differential testing approach while it is added after LCT is run in the decoupled differential testing approach.

An advantage of coupled differential testing is that non-equivalent behavior of the IUT and the reference implementation is reflected in the control flow of the code instrumented and run by LCT. If the reference implementation and the IUT show non-equivalent behavior, then the control flow eventually reaches the location in the comparison class at which the "non-equivalent behavior"-exception is thrown. If there is any sequence of method calls of length k that leads to this location, then LCT generates a test that reaches the location if full path coverage is reached and the call sequence bound used in the test generation is at least k. This implies that, any sequence of calls leading to non-equivalent behavior is guaranteed to be found by coupled differential testing as long as LCT reaches full path coverage and the call sequence bound is sufficiently large. As said before, such a guarantee can not be given for decoupled differential testing.

Random differential testing LCT can be used in random mode to explore random execution paths. In the random differential testing approach LCT is used to generate a set of random tests, i.e., JUnit tests that execute random sequences of methods with random arguments, for the reference implementation. These random tests are then modified to compare behaviors of calls in the same way as the tests in decoupled differential testing. Alternatively, random tests could be generated for the IUT or even for the comparison class used in coupled differential testing. As both of theses approaches, however, compare the IUT and the reference implementation based on random call sequences, they lead to comparable results.

5 Experiments

LCT was experimentally evaluated by running LCT-based differential testing on Java Card example code called Logical Channels Demo [8] and a number of slightly mutated versions of that code. A short introduction to the Java Card technology and a description of the Logical Channels Demo is given in Sect. 5.1. LCT was used to check different implementations of the Logical Channels Demo for bounded class equivalence using the methods described in Sect. 4. The original version of the Logical Channels Demo was used as the reference implementation and mutated versions of the Logical Channels Demo that show different class behavior were used as IUTs. A mutation generator called μJava [22] was used to generate the mutated versions. μJava and its use are described in Sect. 5.2. The numbers of IUTs for which the non-equivalent class behavior was detected by the individual testing approaches were used to compare effectiveness of concolic-testing-based differential testing to that of random differential testing. Section 5.3 describes the exact test set up while Sect. 5.4 discusses the results of the experiments.

5.1 Java Card and the Logical Channels Demo

The Java Card Technology [7, 8] enables Java programs to execute on smart cards. Java smart card programs, called applets, are an interesting class of Java programs. As they tend to be smaller than "real" Java programs, they are well suited for being used as test data in controlled experiments. For the experimental evaluation of LCT, smart card example code called Logical Channels Demo was used. The idea behind the Logical Channels Demo is to use a smart card to charge a user for connecting to a data communication network. Although the Logical Channels Demo is designed for demonstration purposes and lacks some features like proper user authentication, it is a non-trivial application that is similar to other Java Card applications. The Logical Channels Demo has previously been used for two LIME related case studies [23, 24].

Java Card applets Java Card applets are Java programs that can be run on smart cards. When executed, Java Card applets communicate with an off-card application. There are some differences between Java Card applets and normal Java applications. Most notably, Java Card applets can only use a very limited subset of the Java features due to the limitations of the hardware they run on. For example, the basic data types `long`, `float`, `double`, `char` and `String` are not supported and the support of `int` is optional. Also, multidimensional arrays and most standard Java classes are not supported. In addition, Java Card applets use special methods for throwing exceptions that are intended to be caught by the off-card application.

The Logical Channels Demo The Logical Channels Demo is one of several demos that are part of the Java Card Development Kit [8]. The Logical Channels Demo allows to use a smart card in a device that provides access to a network for a certain fee. The network is divided into several areas and the user has a home area, in which the fee is

lower than in the rest of the network. The smart card on which the Logical Channels Demo is installed keeps track of the user's account's balance.

The Logical Channels Demo consists of two applets: one manages the user's account, and the other receives the state of the network connection from the off-card application and debits the account accordingly. The main purpose of the Logical Channels Demo is to illustrate how these two applets can be active and communicate with the off-card application simultaneously.

The Logical Channels Demo has been used in a previous case study to illustrate the use of the LIME interface specification language [23]. In course of that case study, the Java Card specific packet based argument passing and value returning mechanisms were replaced with standard Java arguments and return values in the Logical Channels Demo. The resulting modified version of the Logical Channels Demo was used for the evaluation of LCT as well.

Usually, Java Card applets need a Java Card simulator in order to be executed on a normal PC. As it would be challenging to use the LCT test generation in conjunction with a Java Card simulator, the Logical Channels Demo was modified in a way that allows the applets to run without a simulator. This was achieved by replacing Java Card API methods with stub code. The stub code behaves in the same way the Java Card API does in all respects that were of importance in the used testing setup.

The different testing approaches described in Sect. 4 compare the behavior of two implementations of one class. The Logical Channels Demo, however, consists of two applets, i.e., the behavior of two classes has to be compared simultaneously if one wants to compared two implementations of the Logical Channels Demo. In order to make it still possible to use the behavior comparison methods, a simple wrapper class that has one instance of each applet and provides methods that call the methods of the applets was added. Then, two implementations of the Logical Channels Demo could be compared by comparing the behavior of their wrapper classes.

The modified version of the Logical Channels Demo contains 328 lines of code. The comparison class used in coupled testing adds additional code that stores and compares return values and exceptions. The Logical Channels Demo does not contain any loops and therefore has only a finite number of execution paths.

5.2 The Mutations

In order to evaluate LCT experimentally, the differential testing methods described in Sect. 4 were used to compare pairs of implementations of the Logical Channels Demo. Mutated versions of the Logical Channels Demo, i.e., versions that contain small errors, were used to simulate faulty IUTs, and the bug-free version of the Logical Channels Demo was used as the reference implementation.

For the generation of the mutated classes, μJava [22] version 3 was employed. μJava generates mutations of Java programs by introducing small changes, e.g., by replacing an operator with another operator. While μJava ensures that the mutated programs can be compiled without errors, it does not guarantee that they really affect the program behavior. μJava may, for instance, generate a mutation that alters the value of a variable that is never used again.

μJava can generate two types of mutations: method-level and class-level mutations. A method-level mutation makes a small modification to the code inside one method, e.g., changes the sign of an operand in an expression. A class-level mutation modifies properties of and access to the class's fields and methods, e.g., turns a non-static field into a static field. Class-level mutations often only change the behavior of programs that use at least two instances of the mutated class. A static field, for instance, behaves just like a non-static field as long as only one instance of the class it belongs to is created. Therefore, a test setup that creates multiple instances of each class would be required to find class mutations. The used test setup, however, only creates one instance of each tested class and therefore would be unable to detect class methods. Hence, only method-level mutations were generated.

μJava generated 287 mutations of the Logical Channels Demo. For the experiments, only mutations that alter the behavior of the applets in a way that can theoretically be detected using the described class comparison methodology, i.e., ones that change the behavior of the applets w.r.t. the notion of equivalent behavior introduced in Sect. 4, were of interest. All mutations that do not change the behavior in such a way were classified and removed. Random differential testing was used to determine which mutations obviously changed the class behavior. The mutations for which no non-equivalences of behavior were discovered in this way were evaluated manually in order to determine whether or not they change the class behavior. Out of the 287 mutations, 65 did not affect the class behavior and were removed. The remaining 222 mutations were used for the experimental evaluation of LCT.

5.3 Test Setup

For experimental evaluation of LCT, the three differential testing approaches described in Sect. 4 were applied. The mutated classes were used as IUTs and the original Logical Channels Demo as the reference implementation. Bounds from one to three were used, i.e., the behaviors of pairs the original Logical Channels Demo and a mutant were checked for 1-bounded, 2-bounded and 3-bounded class equivalence.

The definition of equivalent method behavior introduced in Sect. 4 does not give an exact definition of when values returned or exceptions thrown by methods are equivalent. All methods in the Logical Channels Demo either return nothing or values of primitive types. Therefore, whether or not return values are equivalent was determined using the Java "==" operator. Java Card applets throw a special type of exception which contains a two-byte error code that indicates the exceptions cause. Such exceptions were considered equivalent if they contain the same error code. All other (native Java) exceptions were considered equivalent, if they were of the same class.

During experimentation, LCT was configured to use the SMT-solver Boolector [9]. LCT's depth limit was set high enough to allow LCT to always explore the full symbolic execution tree. The number of generated tests for random differential testing was set to 10000. All other LCT options were set to their default values.

Table 1. The number of correctly detected mutations for the different approaches and depths.

Approach	1-bounded	2-bounded	3-bounded
Decoupled	121 (54.50%)	185 (83.33%)	221 (99.95%)
Coupled	123 (55.41%)	187 (84.23%)	221 (99.95%)
Random	95 (42.79%)	151 (68.02%)	184 (82.88%)

Table 2. A more detailed listing of the numbers of mutations caught. The numbers for combinations not listed (e.g., only caught by random differential testing) are zero.

Caught by approach(es)	1-bounded	2-bounded	3-bounded
All	95	150	184
Coupled and decoupled	26	35	37
Coupled and random	0	1	0
Only coupled	2	1	0
None	99	35	1

5.4 Results and Discussion

The differential testing approaches introduced in Sect. 4 were run on every pair of the original Logical Channels Demo and one of the mutants described in Sect. 5.2 with bounds ranging from one to three. Then, the number of pairs for which non-equivalent class behavior was reported was used to compare the effectiveness of the different testing approaches. Also, the number of tests generated and the times needed for test generation and execution by the two concolic-testing-based approaches were compared.

Table 1 shows for each approach and each bound the number of mutations caught, i.e., the number of mutants for which behavior that differs from the original was correctly reported. At bound one, random differential testing was able to catch 95 out of the 222 mutations. Decoupled differential testing caught 121 and coupled differential testing caught 123 mutations. At bound two, these numbers increased to 151 for random, 185 for decoupled and 187 for coupled differential testing. At bound three, decoupled and coupled differential testing both caught all but one mutation while random differential testing caught only 184 mutations. These results illustrate that random differential testing is suited to catch many of the μJava mutations. Using the concolic-testing-based approaches, however, pays off in a significantly higher mutation detection rate.

While Table 1 shows the individual detection rates for the approaches, it does not provide more detailed information about the results, e.g., whether there were mutations caught by random differential testing that were not caught by the concolic-testing-based approaches. This more detailed information can be found in Table 2.

Independently of the bound, coupled differential testing caught every mutation that was caught by random or decoupled differential testing. At bounds one and two, coupled differential testing caught two mutations that were not caught by decoupled differential testing. The reason is that some mutations alter the class behavior only for very few input values. If a mutation, for instance, replaces the condition `if(a < 42)` where `a` is a method argument with `if(a <= 42)`, then the mutation is only caught if the

corresponding method is called with argument 42. Decoupled differential testing can, however, reach full path coverage without using 42 as argument. Therefore, there is a chance that decoupled differential testing misses the described mutation. For coupled differential testing in contrast using 42 as argument is the only way to reach the location in the code where the "different behavior" exception is thrown. Therefore, coupled differential testing can not reach full path coverage without catching the mutation. The mutations that were caught by coupled but not by decoupled differential testing during the experiments were similar to the described mutation. It was, however, observed that many similar mutations were caught by decoupled differential testing even though they alter the behavior only for a very limited number of values. The reason is that the used SMT-solver tended to assign values that occur as constants in the given constraints to variables during the experiments, e.g., `42` in the given example.

At bound three, coupled and decoupled differential testing both caught all but one mutation. Manual inspection of the one mutant that was never caught revealed that a sequence of at least four method calls is needed to make the mutant show behavior that differs from the behavior of the original Logical Channels Demo. Decoupled, coupled and random differential testing were executed with bound four for the mutation and all three methods were able to then catch the mutation. Generating the tests, however, took almost 33 minutes for decoupled and almost one hour for coupled differential testing. Therefore, no bound four tests were generated for the other mutations.

Table 3 shows the average numbers of tests generated per mutant and the average time required for generating and executing the tests for decoupled and coupled differential testing. Like the tests generated by decoupled and coupled differential testing, the random tests were generated using LCT. As random tests generation is not part of the LCT's core functionality, LCT generates random tests not as efficiently as dedicated random test generation tools. Thus, no times for random test generation are listed.

Generally, the number of tests generated by decoupled and coupled differential testing can be expected to grow exponentially in the bound used. At bound one, decoupled differential testing generated 22.13 tests on average and coupled differential testing generated 11.32 tests on average. At bound two, the average numbers of tests generated increased to 169.56 for decoupled and 147.36 for coupled differential testing and at bound three to 1306.22 and 1398.18, respectively. In order to give random differential testing a fair chance, the number of tests generated by random differential testing was chosen to be significantly higher than the average number of tests for the concolic-testing based approaches, namely 10000.

In coupled differential testing, every method call in the comparison class executes the same method once in each implementation. Thus, every method call in the comparison class executes exactly the same method twice if the IUT and the reference implementation are exactly identically. Therefore, there is exactly the same number of paths in the comparison class as in one of the implementations alone. In such a situation, LCT generates every test twice in decoupled differential testing, once for the IUT and once for the reference implementation. In coupled differential testing in contrast, LCT generates every test only once for the comparison class. Therefore, the number of tests generated in decoupled differential testing is twice the number of tests generated in coupled differential testing if the IUT and the reference implementation are identical.

Table 3. The average numbers of tests generated per mutant and the average times spent per mutant for generating tests, executing tests and in total.

	1-bounded		2-bounded		3-bounded	
	Decoupled	Coupled	Decoupled	Coupled	Decoupled	Coupled
Tests generated	22.1	11.3	169.6	147.4	1306.2	1398.2
Generation time	81.90 s	48.21 s	109.23 s	67.30 s	288.05 s	293.18 s
Execution time	3.12 s	1.56 s	4.68 s	1.92 s	13.08 s	4.63 s
Total time	85.02 s	49.77 s	113.91 s	69.22 s	301.13 s	297.81 s

This observation suggests that the average number of tests generated should be larger in decoupled differential testing. Indeed, the number of tests generated by decoupled differential testing was almost twice as high compared to coupled differential testing at bound one. At bound two, however, decoupled differential testing generated only about 15% more tests and at bound three decoupled differential testing even generated less tests than coupled differential testing.

The reason why the number of tests generated by coupled differential testing increased faster than the number generated by decoupled differential testing is that the mutants used as IUTs and reference implementation in the experiments are not exactly the same. Every method that is implemented differently in the IUT and the reference implementation contributes to an increase of the number of paths in the comparison class. Consider, e.g., a method that is implemented in a way that there are three paths through the method in the IUT and two paths in the reference implementation. Assume that the implementations differ in a way such that there are combinations of input values that allow to execute every combination of a path in the IUT and a path in the reference implementation. Then, there are six possible paths through the corresponding method in the comparison class. Thus, coupled differential testing will try six paths through the corresponding method in the comparison class, while decoupled differential testing will only try five paths, three for the IUT and two for the reference implementation. This effect increases the number of tests generated in coupled differential testing. The effect is even stronger for paths on which the method is called more than once. In the example, there are $6^2 = 36$ paths that consist of calling the described method twice in the comparison class. In the IUT in contrast there are only 3^2 paths that consist of two calls of the method and in the reference implementation there are only 2^2. Assuming that there are no other methods in the tested class this means that coupled differential testing will generate 36 tests at bound two while decoupled differential testing will only generate $3^2 + 2^2 = 13$ tests. This illustrates that the effect is stronger at higher depths which explains that the number of tests generated by coupled differential increased faster than the number of tests generated by decoupled differential testing. Also, the effect can be expected to be stronger if the differences between the IUT and the reference implementation are more extensive than those caused by the mutations used for the experiments.

Table 3 also shows the time spent generating and executing tests by decoupled and coupled differential testing. The test were executed on a Linux computer with 4 GB memory and an Intel Core 2 Duo E6550 processor running at 2.33 GHz. The times for

test generation and execution grow less quickly than the number of tests generated due to the fact that some steps like the instrumentation during test generation and starting the Java virtual machine and the JUnit test runner during test execution take roughly the same amount of time independently of the bound.

6 Conclusions

This paper introduces the LIME concolic tester (LCT), a new open source concolic testing tool for Java programs, and discusses the main design choices behind LCT. The paper focuses, in particular, on differential testing of Java Card applets using LCT. A setting for differential testing of applets is defined and two alternative approaches to generating test sets for differential testing using concolic testing and LCT are devised. The two approaches are compared experimentally to random testing in the Java Card application domain. The experiments show that the proposed concolic testing approaches compare favorably to random testing and the test sets generated by LCT can find more bugs than considerably bigger sets of randomly generated tests.

Acknowledgements The authors would like to warmly thank the anonymous referees for very detailed feedback to improve on this paper and also on interesting suggestions for further research.

References

1. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005), ACM (2005) 213–223
2. Sen, K.: Scalable automated methods for dynamic program analysis. Doctoral thesis, University of Illinois (2006)
3. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006). Volume 4144 of Lecture Notes in Computer Science., Springer (2006) 419–423 (Tool Paper).
4. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM conference on Computer and communications security (CCS 2006), ACM (2006) 322–335
5. Tillmann, N., de Halleux, J.: Pex – White box test generation for .NET. In: Proceedings of the Second International Conference on Tests and Proofs (TAP 2008). Volume 4966 of Lecture Notes in Computer Science., Springer (2008) 134–153
6. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), USENIX Association (2008) 209–224
7. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Prentice Hall (2000)
8. Sun Microsystems: Java Card Development Kit 2.2.2 (2009) `http://java.sun.com/javacard/devkit`.

9. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009). Volume 5505 of Lecture Notes in Computer Science., Springer (2009) 174–177
10. Kähkönen, K., Lampinen, J., Heljanko, K., Niemelä, I.: The LIME Interface Specification Language and Runtime Monitoring Tool. In: Proceedings of the 9th International Workshop on Runtime Verification (RV 2009). Volume 5779 of Lecture Notes in Computer Science., Springer (2009) 93–100
11. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, The Internet Society (2008) 151–166
12. Godefroid, P., Levin, M.Y., Molnar, D.A.: Active property checking. In: Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, ACM (2008) 207–216
13. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), IEEE Computer Society (2007) 416–426
14. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary Linux programs. In: Proceedings of the 18th USENIX Security Symposium (USENIX Security 2009), USENIX Association (2009) 67–81
15. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006). Volume 4144 of Lecture Notes in Computer Science., Springer (2006) 81–94
16. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337–340
17. Lampinen, J., Liedes, S., Kähkönen, K., Kauttio, J., Heljanko, K.: Interface specification methods for software components. Technical Report TKK-ICS-R25, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland (Dec 2009)
18. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999), IBM (1999) 13
19. Kähkönen, K.: Automated test generation for software components. Technical Report TKK-ICS-R26, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland (Dec 2009)
20. Factor, M., Schuster, A., Shagin, K.: Instrumentation of standard libraries in object-oriented languages: The twin class hierarchy approach. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), ACM (2004) 288–300
21. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE 2008), ACM (2008) 226–237
22. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: An automated class mutation system. Software Testing, Verification and Reliability **15**(2) (2005) 97–133
23. Kindermann, R.: Testing a Java Card applet using the LIME Interface Test Bench: A case study. Technical Report TKK-ICS-R18, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland (Sept 2009)
24. Holmström, P., Höglund, S., Sirén, L., Porres, I.: Evaluation of Specification-based Testing Approaches. Technical report, Åbo Akademi University, Department of Information Technologies (Sept 2009) `https://poseidon.cs.abo.fi/trac/gaudi/lime/raw-attachment/wiki/MainResults/t34-report.pdf`.