A Semi-empirical Model of Test Quality in Symmetric Testing: Application to Testing Java Card APIs *

Arnaud Gotlieb
IRISA-INRIA
Campus Beaulieu
35042 Rennes Cedex, FRANCE
Arnaud.Gotlieb@irisa.fr

Patrick Bernard
OBERTHUR CARD SYSTEMS
rue Auguste Blanche
92800 PUTEAUX, FRANCE
p.bernard@oberthurcs.com

ABSTRACT

In the smart card quality assurance field, Software Testing is the privileged way of increasing the confidence level in the implementation correctness. When testing Java Card application programming interfaces (APIs), the tester has to deal with the classical oracle problem, i.e. to find a way to evaluate the correctness of the computed output. In this paper, we report on an experience in testing methods of the Oberthur Card Systems Cosmo 32 RSA Java Card APIs by using the Symmetric Testing paradigm. This paradigm exploits user-defined symmetry properties of Java methods as test oracles. We propose an experimental environment that combines random testing and symmetry checking for (on-card) cross testing of several Java Card API methods. We develop a semi-empirical model (a model fed by experimental data) to help deciding when to stop testing and to assess test quality.

1. INTRODUCTION

Although formal verification and software testing were viewed as opposites for a long time, with formal verification concentrating on proving program correctness while testing concentrating on finding faults in program implementation, they can now be considered as complementary techniques [1]. In the smart card field, software testing is required by the Common Criteria evaluation scheme [2] to increase the confidence level of the certifying authority in the implementation correctness of security functions. In this context, techniques and tools that permit to automate (even partially) the testing process are welcome. Research works in that field include the BZ-testing approach designed by Legeard et al. [3, 4] to generate automatically test cases from a formal B or Z specification. The corresponding tools suite has been employed to validate the Java Card transaction mechanism by generating test cases on the boundary states of the formal specification [5]. In [6], Pretschner et al. followed a similar approach by using the AUTOFOCUS tool for specifying the command/response mechanism of an inhouse smart card and generating test cases for validating the authentification protocol of the card. At the same time, Clarke et al. [7] developed symbolic test generation algorithms and applied them to generate on-the-fly test cases for a feature of the CEPS[1] e-purse application and Martin and Du Bousquet [8] proposed to use UML-based tools to generate test suites for testing Java Card applets.

All these approaches have in common to require first a formal model (Z or B specification, automata, input/output transition system or statecharts) to be constructed in order to generate test cases. When the time-to-market of a new product is critical, this effort appears as being too costly and cheaper (but still rigorous) approaches are needed. Techniques such as statistical testing [9–11], boundary testing [12], or local exhaustive testing [13] do not require a formal model to be developed. Statistical testing aims at selecting randomly the values inside the input domain of the application under test by using pseudo-random numbers generators, boundary testing relies on selecting the boundaries of an input space partition, whereas local exhaustive testing systematically explores a bounded part of the input domain. In these approaches, testing just depends on the availability of oracles, that is, some procedures for predicting the expected results of the applications under test. Unfortunately, as earlier pointed out by Weyuker [14], there are programs to be tested for which the design of oracles is a non-trivial task. Examples of such programs in the smart card field include standard and proprietary Java Card APIs as they are just usually described by their interfaces and a few lines of natural text[2]. For these APIs, current industrial practices rely on coding the oracle as the result of another program that will be confronted with the result of the API under test. This approach suffers from several drawbacks such as the high cost of the development of oracles and the existence of faults into the oracles.

Recently, we have proposed [15] to address this oracle problem for Java programs by using user-defined symmetries of programs to check the correctness of the computed output. Here, symmetries are input-output permutation relations over program executions that lead to partitioning the input space into equivalence classes and the equivalence between two executions serves as an oracle. We introduce a testing paradigm called Symmetric Testing, where automatic test data generation was coupled with symmetries checking and local exhaustive testing to uncover faults inside the programs.

In this paper, we report on an experience in applying Symmetric Testing to test methods of the Oberthur Card Systems Cosmo 32 RSA V3.4 Java Card API [33] by using random testing. Unlike our previous work [15], we develop here an original semi-empirical

*Supported by the Réseau National des Technologies Logicielles as part of the CASTLES project (www-sop.inria.fr/everest/projects/castles/)

[1]The Common Electronic Purse Specification is a standard for creating inter-operable multi-currency smart card e-purse systems.
[2]Although formalizations do exist [31].

model to help decide when to stop testing and to assess test quality in Symmetric Testing. This model is fed with an empirical parameter (based on symmetry checking) in a theoretical model of random testing, in order to obtain the minimum number of test data required to reach a given level of quality. From the Oberthur Card Systems Cosmo 32 RSA V3.4 Java Card API [33], we have selected the methods to test by studying their symmetry properties, as Symmetric Testing is only suitable for testing programs that possesses input-output symmetry relation. By using several tools, we have designed an experimental environment to build our semi-empirical model and to apply Symmetric Testing in situations as close as possible to the real situations. In contrast with other research works in testing Java card programs [6, 7], test execution and symmetries checking have been conducted by cross-testing on a smart card and not by using simulations.

The rest of the paper is organized as follows: section 2 presents the Symmetric Testing paradigm and gives examples of symmetry relations. Section 3 reports the symmetry analysis of a few methods of the Oberthur Card Systems Cosmo 32 RSA V3.4 Java Card API while section 4 details our semi-empirical model of random testing based on symmetries checking. Section 5 reports the first experimental results and discusses extension of the framework to handle non-symmetric methods of the Java Card APIs. Finally section 6 pinpoints several perspectives to this work.

2. SYMMETRIC TESTING

Exploiting symmetry in verification is not a new idea. Emerson and Sistla [17] and Ip and Dill [18] proposed early to exploit structural symmetries to address the problem of state explosion in model checking. This approach has been experienced and proved interesting in practice in several tools, such as VeriSoft [19] or SPIN [20]; its principle is based on basic results from group theory [17–19] and partial order techniques [21].

Based on similar ideas, we recently introduced Symmetric Testing [15] in the context of Software Testing. The flavour of our approach is explained here on a very basic example. Consider a program P intended to compute the greatest common divisor (gcd) of two non-negative integers u and v and suppose that P is tested with the following test datum ($u = 1309, v = 693$) automatically generated by a random test data generator. Although we all know how to compute the gcd of two integers[3], it is not so easy to predict the expected value of $gcd(1309, 693)$ without the help of a calculator. Fortunately, gcd satisfies a simple symmetry relation: $\forall u \forall v, gcd(u, v) = gcd(v, u)$. So, if $P(1309, 693) \neq P(693, 1309)$ then the testing process will succeed to uncover a fault in P without the help of a complete oracle of gcd. Note that such a symmetry relation is a necessary but not sufficient condition, for the correctness of P. Such user-specified relations between several program executions have been called metamorphic relations and thoroughly investigated by Chen et al. [22–24].

Identifying such symmetry relations for larger programs might appear to be difficult or useless to detect non-trivial fault. On the contrary, we argue that numerous programs have to satisfy symmetry relations and these relations are useful for detecting subtle faults. In fact, every program P that takes an unordered set as argument has to satisfy a symmetry relation: the expected outcome of P is invariant under any permutation of the elements of the set. Numerous programs take unordered sets as arguments: consider sorting or selection programs that are used in search engines, programs that operate over data buffers, or graph-based programs just to name a few. Note that experimental evidence are also available to support this argument in [23, 24] and [15].

[3]With the Euclidian algorithm for example.

2.1 Symmetry relations

We generalized the above idea to obtain a formal and generic definition of symmetry relation. This definition is based on basic results from Group theory that are briefly recalled here. A detailed but still accessible presentation can be found in [25].

The notion of **symmetric group** is the corner-stone of Symmetric Testing. The symmetric group S_n is the set of bijective mappings from $\{1, .., n\}$ to itself. It has exactly $n!$ elements, called permutations. A permutation in S_n is written: $\theta = \begin{pmatrix} 1 & .. & n \\ i(1) & .. & i(n) \end{pmatrix}$ where $i(1), .., i(n)$ denote the images of $1, .., n$ by the permutation θ. A group action of S_n on a set E is a mapping $(\theta, x) \mapsto \theta \cdot x$ such as: $id_{S_n} \cdot x = x$ and $(\theta_1 \circ \theta_2) \cdot x = \theta_1 \cdot (\theta_2 \cdot x)$ for all $x \in E$ and $\theta, \theta_1, \theta_2 \in S_n$ (we say that S_n acts on E and θ acts on x). Note that E is closed under the action of S_n.

It is well-known that any permutation can be expressed as the composition of certain simple permutations, called cycles. Consider for example the permutation

$\theta = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{pmatrix}$ of S_5, the same permutation can be written as $\theta = (1\ 3)(2\ 4\ 5)$ where each pair of brackets denotes a **cycle** $(a_1\ a_2\ ..\ a_r)$, that maps a_1 to a_2, .. a_{r-1} to a_r, a_r to a_1 and leave unchanged the other elements. A cycle written $(a_i\ a_j)$ is usually referred to as a **transposition**.

A subset X of elements of a symmetric group S_n is a set of generators iff every element of S_n can be written as a finite composition of the elements of X. For example, S_3 is generated by the two transpositions $\tau_1 = (1\ 2)$ and $\tau_2 = (2\ 3)$. More generally, S_n is generated by the transposition $\tau = (1\ 2)$ and the cycle $\sigma = (1\ 2\ ..\ n)$ and cannot be generated by less than two permutations [25]. Note that other two-generator sets can be found for S_n.

Symmetries of the function computed by a program P become interesting with regards to testing when they express general abstract properties. This leads to the notion of symmetry relations for a program.

DEFINITION 1 (**symmetry relation**). *Let P be a program that computes a function f over an input domain $dom(f)$ toward a range domain $ran(f)$, and let S_n act on $dom(f)$ with a group action $\cdot$ and S_m act on $ran(f)$ with a group action $\odot$. A symmetry relation $\Psi_{n,m}$ holds for P iff*

1. $\forall \theta \in S_n, \exists \eta \in S_m$ *such that* $\forall x \in dom(f)$, $f(\theta \cdot x) = \eta \odot f(x)$
2. $\Psi_{n,m} : \theta \longmapsto \eta$ *is a homomorphism from* S_n *to* S_m

The first item requires f to satisfy an invariant property for all θ in S_n and for all x in the input domain of f. Note that η, the image of θ by the symmetry relation $\Psi_{n,m}$, is independent of the choice of x. Most of the time the two group actions will be the same ($\cdot \equiv \odot$), however we will see below an example of distinct group actions in a symmetry relation. The second item requires the symmetry relation to be a homomorphism. A homomorphism is a map φ from G_1 to G_2 such that $\varphi(\theta \circ \theta') = \varphi(\theta) \circ \varphi(\theta')$ for all $\theta, \theta' \in G_1$. Informally speaking, this requirement guarantees the symmetric structure of $dom(f)$ to be preserved by application of f, allowing so nice composition properties of symmetric relations. In our framework, we make an extensive use of this property to optimize the symmetry testing process, as explained below.

2.2 Examples

As an example, consider the Java Card program `void max3(byte[] A, byte[] B)` which selects the three maximum values of the array A and sorts them into the array B. If n denotes the size of A ($n \geq 3$) and f denotes the function computed by `max3` from $\mathbb{B}^n$ to $\mathbb{B}^3$, where $\mathbb{B}$ is the finite set of all possible bytes whose values are 8-bit signed two's complement integers, then the program `max3` has to satisfy a $\Psi_{n,3}$ symmetry relation because the array `B` is invariant under any permutation of `A`. Here, the considered group action (in both cases) is defined by: $S_n \times \mathbb{B}^n \to \mathbb{B}^n$, $(\theta, A = [a_1, .., a_n]) \mapsto \theta \cdot A = [a_{\theta^{-1}(1)}, .., a_{\theta^{-1}(n)}]$
As `B` is required to be sorted, all permutations θ will map to the identity of S_3.

As a more complicated example, consider the program `short getIndex0(short[] A)` that takes an array `A` of (non-negative) **distinct** values as argument and returns the index of the occurrence of 0 in the array or throws an exception if 0 is not present in `A`. The program `getIndex0` computes a function f from $\mathbb{S}^n$ to $\{0, 1, .., n-1\} \cup \{err\}$ where $\mathbb{S}$ denotes the finite set of 32-bit signed short integers, n denotes the size of `A` and err denotes an erroneous symbolic value. When 0 belongs to the array `A`, `getIndex0` has to satisfy a $\Psi_{n,n}$ symmetry relation because, 1) for any $\theta \in S_n$, $f(\theta \cdot A) = \theta \odot f(A)$ for all $A \in \mathbb{S}^n$ that contains an occurrence of 0, and 2) the identical map $\theta \longmapsto \theta$ is a group homomorphism. For instance, if
$A = [98, 4578, 1258, 654, 2589, 558, 12577, 0, 4876, 25541]$ and $\theta = (1\ 2\ ..\ 10)$ then $f(A)$ returns 7 and $f(\theta \cdot A)$ returns 8 which is the image of 7 by θ when it acts over $\{0, 1, .., 9\}$. Note that this example shows two distinct group actions: S_n acts over $\mathbb{S}^{10}$ when it is applied to the input sequence A of f whereas it acts over $\{0, 1, .., 9\}$ when applied to the outcome of f with the following group action:
$S_n \times \{0, 1, .., n-1\} \to \{0, 1, .., n-1\}$, $(\theta, x) \mapsto \theta \odot y = \theta(x)$.

2.3 Symmetric Testing

Symmetry relations can be used to seek for a subclass of faults within an implementation. Informally speaking, the Symmetric Testing principle aims at finding counter-examples (called symmetry violations) of the symmetry relation that a program has to satisfy.

DEFINITION 2. *(**Symmetry violation**) let P be a program over an input domain D and $\Psi_{k,l}$ be a symmetry relation that P has to satisfy, then a symmetry violation for P w.r.t. $\Psi_{k,l}$ is a couple (x, θ) such as $x \in D, \theta \in S_n$ and $P(\theta \cdot x) \neq \Psi_{k,l}(\theta) \odot P(x)$.*

The interesting point here is that symmetry violation can be checked by program executions whereas trying to prove formally that the function computed by a program satisfies a symmetry relation would be very difficult. Note that there is no way to distinguish among the two test data x and $\theta \cdot x$ the one that leads to an incorrect outcome for P. In the worst case, they can even be both faulty. So, given a set of test data and a symmetry relation, we get a naive procedure that can uncover a subclass of faults in P: it requires to compute P with all the permutations of the permutable inputs of each vector x in the test set and then to check whether the outcome vectors are equal to a permutation of the vector returned by P. The latter operation is called an *outcome comparison* in the rest of the paper.

However, the somehow naive procedure given above requires an outcome comparison for each possible permutation in the Symmetric Group S_n and, as S_n contains $n!$ permutations, the approach becomes impractical when n increases. The following result is exploited to reduce the number of outcome comparisons:

THEOREM 1. *Let P be a program that computes a function f and $\Psi_{k,l}$ be a symmetry relation for P, let $\tau = (1\ 2)$ and $\sigma = (1\ 2\ ..\ k)$, then we have*

$$\begin{cases} f \circ \tau = \Psi_{k,l}(\tau) \circ f \\ f \circ \sigma = \Psi_{k,l}(\sigma) \circ f \end{cases} \iff f \circ \theta = \Psi_{k,l}(\theta) \circ f \quad \forall \theta \in S_k$$

A proof of this theorem can be found in [15]; it is based on the fact that $\Psi_{k,l}$ is required to be a group homomorphism. Hence, by showing that $f(\tau \cdot x) = \Psi_{k,l}(\tau) \odot f(x)$ and $f(\sigma \cdot x) = \Psi_{k,l}(\sigma) \odot f(x)$, we get $f(\theta \cdot x) = \Psi_{k,l}(\theta) \odot f(x)$ for all $\theta \in S_k$, meaning that only two permutations are required to be checked. Moreover, by noticing that if (x, θ) is a symmetry violation then $(\theta \cdot x, \theta^{-1})$ is automatically another symmetry violation, the input domain to be explored can even be shrinked. These properties are exploited to design an efficient procedure for Symmetric Testing, that is fully described in [15].

The rest of the paper reports on our experience in applying Symmetric Testing combined with Random Testing to the testing of some Java Card API methods.

3. SYMMETRY IN JAVA CARD API

Unlike other smart cards, a Java Card includes a Java Virtual Machine and a set of API classes implemented in its read-only memory part. The Java Card Virtual Machine provides the interpretation of Java Card language constructs and the APIs are a set of classes and interfaces providing additional functionality that can be accessed by Java Card applets. A complete view of the development process of Java Card applets can be found in [29]. The OCS[4] Cosmo 32 RSA V3.4 (called Cosmo in the following) contains an implementation of the Java Card APIs.

3.1 The Cosmo Java Card APIs

The structure of the Cosmo Java Card platform is given in Fig.1. It consists of several components, such as an implementation of the Java Virtual Machine, the open platform applications, a set of packages implementing the standard SUN's Java Card API [16] and a set of proprietary packages. The four OCS proprietary packages consists of standard security services such as the VISA Open Platform Provider Security Domain, a set of base classes for implementing a Provider Security Domain, a complete range of classes for creating, maintaining and inspecting the card file-system and methods that are useful for JCRE related operations. Note that the Cosmo Java Card platform includes garbage collection facilities.

3.2 Symmetry analysis of a selected Cosmo Java Card API class

Among several possibilities, we selected `com.oberthurcs.javacard.file.Utilfs` as a case study because it consists of several generic utility methods that present symmetries. All the methods operate on byte or object arrays and are useful for dealing with APDU[5] buffers. The `Utilfs` class is composed of 10 methods, shown in Tab.1 together with their symmetry relations. The first and second column are extracted from the Cosmo API informal specification [33]. The third column summarizes the results of our symmetry analysis. The set of all possible bytes is noted $\mathbb{B}$ and the set of available objects is noted $\mathbb{O}$. Note first that some methods that deal with multiple array elements are tagged as NonAtomic. Atomicity defines how the card handles the contents of persistent storage

[4] Stands for Oberthur Card Systems

[5] Application Protocol Data Unit is an ISO-normalized communication format between the card and the off-card applications.

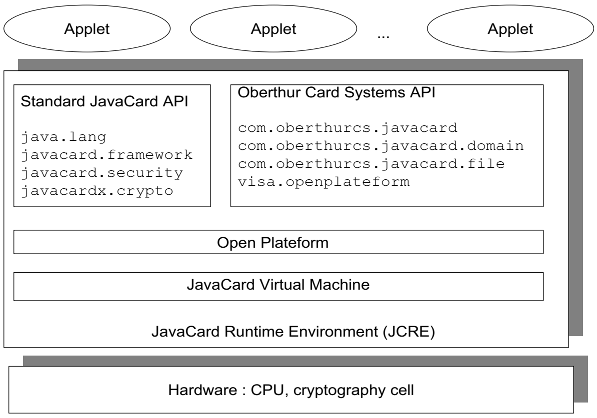

Figure 1: The OCS Cosmo 32 RSA V3.4 platform

after a failure or fatal exception during an update of a class field or an array component [16]. An applet might not require atomicity for array updates. The `Utilfs.arrayAndNonAtomic` method is an example: it shall not use the transaction commit buffer even when called with a transaction in progress.

Among the ten methods of this class, we found that seven have to satisfy a simple symmetry relation. We discuss a few of them; the other ones can easily be deduced from these. The method `short arrayAndNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)` can be abstracted by a function f from $\mathbb{B}^{len} \times \mathbb{B}^{len}$ to $\mathbb{B}^{len}$ as it modifies the input-output parameter `dest` by combining `src` and `dest` and by considering all other parameters as non-variable. `arrayAndNonAtomic` has to satisfy a $\Psi_{len,len}$ symmetry relation because of the following invariant property:

$$\forall \theta \in S_{len}, f(\theta \cdot dest, \theta \cdot src) = \theta \odot f(dest, src)$$

This is due to the fact that $dest$ and src represent two unordered sets of values for this method. Note that the two group actions are distinct as the first one holds over the set $\mathbb{B}^{len} \times \mathbb{B}^{len}$ (i.e. $\theta \cdot (x, y) = (\theta \cdot x, \theta \cdot y)$ where x and y are vectors of $\mathbb{B}^{len}$) whereas the second one $\odot$ holds just over $\mathbb{B}^{len}$. The methods `short arrayCompare( byte[] src, short srcOff, byte patByte, short length)` and `short arrayFindByte(byte[] src, short srcOff, short len, byte pattern)` have each to satisfy a $\Psi_{len,len}$ symmetry relation as the following invariant property holds: $\forall \theta \in S_{len}, f(\theta \cdot src) = \theta \odot f(src)$ where f denotes a map from $\mathbb{B}^{len}$ to $\{1, .., len\}$. In fact, these two programs are selection programs that are invariant to permutations of a subset of their input parameters. In the `Utilfs` class, some methods do not have to satisfy simple symmetry relations. For example, the method `arrayFindShort` has incompatible input types, that is to say the method looks for a short integer variable into an array of bytes. Although we have not realized a full study of the Cosmo Java Card APIs, we took a look at other classes to find symmetry relations. For example, the classes `visa.openplatform.OPSystem`, `javacard.security.MessageDigest` or `javacard.framework.Util` contain methods that have to satisfy symmetry relations. Note that the compositional definition of symmetry relations allows to combine several method calls. However, we also found numerous classes where the symmetry analysis does not reveal any symmetry relations. Examples of such classes include `com.oberthurcs.javacard.file.*` or `javacard.framework.JCSystem`. So, the Symmetric Testing approach remains limited in application to a restricted part of the Cosmo Java Card APIs. For these classes and methods, other input-output properties should be taken as partial oracles as discussed in section 5.3.

4. A SEMI-EMPIRICAL MODEL

In this paper, the Symmetric Testing principle combines random test data generation and automatic symmetry checking. Random testing has traditionally been viewed as a blind approach of program testing. However, results of actual random testing experiments confirmed its potential to reveal faults and as a validation tool [9]. Nevertheless, when the tester wants to exploit a random test data generator, he faces two main difficulties. The first is the classical oracle problem already discussed in the introduction of this paper: an automatic way of checking the output correctness is required. The second problem is to determine the test quality level reached by such a testing approach. In general, it is difficult to quantify how reliable is a program that has only been tested by randomly generated test data. Several works deal with this problem by using a purely theoretical framework based on probabilistic analysis [10, 30]. In this paper, we exploit a semi-empirical model (a model fed by experimental data) to help decide when to stop testing. This section is devoted to the presentation of this semi-empirical model.

4.1 Random testing

Let p_f be the probability that a randomly generated input test datum x exhibit a fault in the program P. A fault in P can be understood as a syntactical change in the source code that leads, for some input data, to a difference between $P(x)$ and the expected output of the function computed by P with x. By a simple probabilistic reasoning, a model of random testing based on p_f can be developed. It is a law between the number N of randomly generated test data and a probabilistic parameter that characterizes the fault-detecting effectiveness of the random testing strategy [9, 32]. The probability of detecting at least one failure is called the test quality[6] and it is noted Q_N [10]. Its value is given by the following definition:

DEFINITION 3. $Q_N = 1 - (1 - p_f)^N$

As an immediate consequence, we get an estimation of the minimum number of test data required to reach a certain value of Q_N:

THEOREM 2. $N \geq \lceil \frac{ln(1-Q_N)}{ln(1-p_f)} \rceil$

where $\lceil x \rceil$ denotes the ceiling function applied to a real number x.

4.2 The empirical parameter p_s

The above model of random testing suffers from a major drawback: it is based on p_f which is almost impossible to evaluate without a precise knowledge of all the existing faults in the program P. We address this problem by using an empirical parameter in place of p_f to build our model. This parameter p_s is related to symmetry checking: p_s is the probability of detecting a symmetry violation (x, θ) when x is randomly generated over a subset of size s of the input domain. This parameter characterizes the probability for Symmetric Testing to reveal a fault in P when it makes use of a random test data generator to generate a single test datum.

In this paper, we propose to empirically evaluate p_s on a correct specimen program by making use of fault-injection techniques.

[6] This measure is also called the P-measure

Table 1: Symmetry in the OCS Utilfs methods

Java Card methods	Informal specifications	Symmetry relations
`short arrayAndNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	Copies the result of a bitwise AND on the first operand dest and the second operand src into dest	The program computes the following function: $f : \mathbb{B}^{len} \times \mathbb{B}^{len} \longrightarrow \mathbb{B}^{len}$ $(dest, src) \longmapsto dest'$ where $dest'$ stands for the value $dest$ computed after application of the `arrayAndNonAtomic` program. It has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayCompare( byte[] src, short srcOff, byte patByte, short len)`	Returns the index of the first byte in the specified part of src that does **not** match patByte, or 0xFFFF if every byte matches	We ask src to contain 1 occurrence of patByte and $srcOff = 0$ $f : \mathbb{B}^{len} \longrightarrow \{1, .., len\}$ $src \longmapsto ret$ `arrayCompare` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayFindByte( byte[] src, short srcOff, short len, byte pattern)`	Returns the index of the first byte in the part of src that matches the specified pattern.	We ask src to contain 1 occurrence of pattern $f : \mathbb{B}^{len} \longrightarrow \{1, .., len\}$ $src \longmapsto ret$ `arrayFindByte` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayFindPattern( byte[] src, short srcOff, short srcLen, byte[] patSrc, short patOff, short patLen)`	Returns the index of the first byte in the part of src that matches the specified pattern.	no simple symmetry
`short arrayFindShort( byte[] src, short srcOff, short len, short pattern)`	Returns the index of the first byte in the part of src that matches the specified pattern (a short is 2 bytes).	no simple symmetry
`short arrayOrNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	Copies the result of a bitwise OR on the first and second operands into dest.	$f : \mathbb{B}^{len} \times \mathbb{B}^{len} \longrightarrow \mathbb{B}^{len}$ $(dest, src) \longmapsto dest'$ `arrayOrNonAtomic` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short arrayXorNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	Copies the result of a bitwise XOR on the first and second operands into dest.	$f : \mathbb{B}^{len} \times \mathbb{B}^{len} \longrightarrow \mathbb{B}^{len}$ $(dest, src) \longmapsto dest'$ `arrayXorNonAtomic` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short getObjectIndex( java.lang.Object[] src, short srcOff, short n, java.lang.Object pattern)`	Returns the index of the nth occurrence in the part of src that matches pattern.	len denotes the length of src and $n = 1$, we ask src to contain 1 occurrence of pattern $f : \mathbb{O}^{len} \longrightarrow \{1, ..., len\}$ $src \longmapsto ret$ `getObjectIndex` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short getShortIndex( short[] src, short srcOff, short n, short pattern)`	Returns the index of the nth occurrence in the part of src that matches pattern.	len denotes the length of src and $n = 1$, we ask src to contain 1 occurrence of pattern $f : \mathbb{O}^{len} \longrightarrow \{1, ..., len\}$ $src \longmapsto ret$ `getShortIndex` has to satisfy a $\Psi_{len,len}$ symmetry relation.
`short getTaggedShort( byte[] src, short srcOff, short srcLen, short tag)`	Returns the index of the first TLV tag in the part of src that matches the specified tag.	no simple symmetry

Using a specimen program is advantageous as we can easily inject faults by modifying its source code. The key point of our approach resides in the knowledge of symmetry violations occurring when checking the output correctness of this program. Note that this approach is based on an uniform hypothesis: the inferred value for the specimen program applies to other programs as well. This hypothesis is debatable and relates to the difficulty of finding a representative sample in statistics. In our framework, we preferred to select a single representative program rather than a large set of non-representative programs. Of course, any other more representative program can be employed. For example, when testing an airborne flight-guidance software, one can employ a well-established correct program of the airborne software domain.

4.3 Our protocol to evaluate p_s

Our protocol to evaluate p_s is based on a set of faulty versions of the specimen program P that are automatically created by a mutation analysis scheme [34]. A mutant Q is a version of P where a single syntactical change has been introduced. Classically, a mutant is said to be killed by a test datum x when $Q(x) \neq P(x)$. In our framework, we will consider a mutant to be killed if there exists $\theta \in S_n$ such as (x, θ) is a symmetry violation for Q w.r.t. $\Psi_{n,m}$. The value of p_s depends on s, the size of the subdomain of this input space that is considered for the random test data generation. Given a size s, the empirical protocol is as follows:

1. built $\{Q_i\}_{i=1..K}$ a set of K mutants of a specimen program P that has to satisfy a symmetry relation $\Psi_{n,m}$;
2. for each Q_i, compute the booleans $b_{i,x} = (Q_i(\tau.x) \neq \Psi_{n,m}(\tau).Q_i(x)$ or $Q_i(\sigma.x) \neq \Psi_{n,m}(\sigma).Q_i(x))$ for each test datum x of the input domain of P;
3. returns $p_s = \frac{1}{N*K}.\sum b_{i,x}$ which is just the median value of the probability for the K mutants.

Note that the programs Q_i are executed on a large part of their input domain, hence it is important to select a specimen program having an input space of reasonable size. Note also that only two permutations are required to be checked in this protocol ($\tau = (1\ 2)$ and $\sigma = (1\ 2\ ..\ n)$). This is a direct consequence of Theorem 1.

We selected the well–known triangle classification program `trityp` [27], that belongs to the Software Testing folklore. It takes three non-negative bytes as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene, isocele, equilateral or illegal. The results of `trityp` must be invariant to every permutation of its three input values, leading to a $\Psi_{3,1}$ symmetry relation. This program appears to be an interesting specimen candidate as it contains a lot of decisions and the probability of a symmetry violation to occur is highly related to the flow of control. Hence, this probability highly depends on the input subdomain that is being explored. This property has recently been investigated from an experimental point of view in [24]. Of course, any more representative program can be employed but we

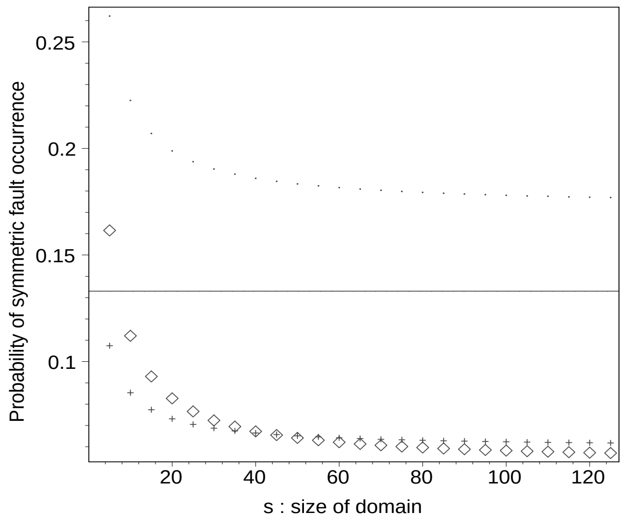

Figure 2: Empirical evaluation of p_s

would like just to study the feasibility of the approach rather than designing a fully acceptance testing methodology.

In our empirical protocol, application of the tool MuJava [34] led to build automatically 36 mutants where an arithmetic operator was replaced (AOR), 85 mutants where a relational operator was replaced (ROR), and 14 mutants where a Logical connector was replaced (LCR). In the current MuJava framework, equivalents mutants[7] are not removed from the set of mutants, although they cannot be revealed by the means of testing [27]. So, 135 mutants of the `trityp` programs were built by the tool and the input domain was restricted to contain at most $s = 126^3 = 2000376$ input values. Among the 135 mutants, 21 were not killed by Symmetric Testing but we kept them in the experiments to avoid introducting a bias in the study.

For each mutant, we compute the number of symmetry violations found when exploring exhaustively a subdomain of the input domain. The average number of symmetry violations that were detected when exploring a subdomain of size s allows for calculating the probability of a symmetry violation to occur by using a uniform random test data generator (p_s). Fig.2 contains the results we got for several increasing values of s ($1^3, 6^3, 11^3, ..., 126^3$) by distinguishing the class of considered mutants (AOR,ROR,LCR). We compute p_s as the center of mass of the 3 bottom values obtained for the greatest size s ($s = 126^3$). Hence, $p_s = (36 * 0.057 + 85 * 0.177 + 14 * 0.062)/135 = 0.133$.

4.4 Test quality based on symmetry violations

Based on definition 3, we get that $q_N = 1-(1-p_s)^N$ for random testing based on symmetry checking. The test quality q_N differs from Q_N as q_N is only based on symmetry checking. In fact, q_N measures the probability of N randomly generated test data in a subdomain of size s to reveal at least one symmetry violation in P. When the property is enforced (P has been tested with a test quality q_N), we get that the symmetry relation is satisfied by the program P with a probability q_N. So, by using this model it becomes possible

[7]programs which compute the same outcome as the original program although a mutation operator is applied

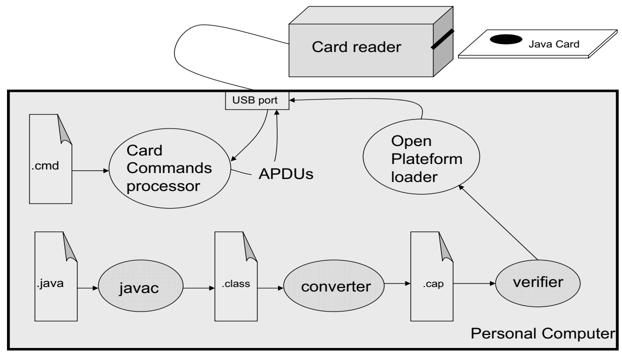

Figure 3: Experimental environment

to assess the symmetry-based test quality for P.

The test quality was required to be equal to 0.9995 as is usually the case in experimental frameworks [10]. By using the empirical value of $p_s = 0.133$ and the theorem $N \geq \lceil \frac{ln(1-q_N)}{ln(1-p_s)} \rceil$, we get that $N = 54$, meanning that at least 54 test cases must be generated. Note that we have just argued that this (arbitrary) value is suitable for feeding our semi-empirical model.

5. EXPERIMENTAL ENVIRONMENT

The goal of the experiments was to study the applicability of Symmetric Testing to reveal faults within Java Card APIs. The validation process of Java Card APIs is usually made of three distinct phases: firstly, Java card test applets are developed on a host machine by using simulation libraries; secondly, the tests are applied to an emulation code that runs on a card emulator; and finally, the test execution is conducted by cross-testing on the Java card. Our experiments were performed in situations as close as possible to the real usage. Hence, test execution and symmetries checking have been conducted by cross-testing on the Java card with the help of a card reader. In this respect, we differ from other smart cards testing research approaches that focus only on test cases generation [6, 7]. In fact, we would like to check whether Symmetric Testing can be combined with Random Testing in a cross-testing environment, which was a challenging question as lots of limitations to memory resources arise in such situations. Moreover, this approach required to develop carrefully our prototype implementation to masterize the memory and time consumption. Fig.3 contains a view of our experimental environment. It is composed of five components: the java compiler (SUN SDK 1.4), the OCS converter that produces standard Java Card byte code (converted applet file), the OCS verifier which statically determines whether a cap file complies with the Java Card specifications, the Open Platform loader which downloads and manages the applets onto the card and a Card Command Processor that sends commands to the smart card via a card-reader interface. Note that the bytecode verification process is done off-card by the OCS verifier. The Card Command Processor is a command interpreter that accepts several language constructs such as conditional and loop.

5.1 Tests generation and execution

In our experimental environment, a special attention has been paid to minimize the communications between the reader and the card. Recalling that our goal was to realize the test execution and the symmetry checking processes on-card, passing large sets of random numbers through the APDU mechanism would have been too time-consuming. Hence, we have designed a Java Card applet (to be loaded on-card) that generates test data and that checks the symmetry relations. This applet serves as a test harness and its size is around 1.2 kbytes. It defines a single command `TEST_API` that launches three executions of the API method under test ($P(x), P(\tau \cdot x), P(\sigma \cdot x)$) and checks the computed output with regards to a given symmetry relation. The applet makes use of a uniform random test data generator provided by the Cosmo API implementation of the `javacard.security.RandomData` class to generate a single test datum x. In case of symmetry violation, a boolean is returned through the APDU mechanism to inform the tester. After having compiled, converted and verified the applet, it is loaded on-card by the OP loader. Then, the command `TEST_API` is launched $N = 54$ times with the help of a command script, interpreted by the Card Command processor.

5.2 Experimental results

For our experiments, we selected the seven methods from the Cosmo Java Card API `Utilfs` that have a symmetry relation to satisfy. In the industrial validation process of the Cosmo kit, these methods are systematically tested by using a few values. For instance, the `arrayAndNonAtomic` method is tested with two randomly generated byte arrays by varying the values of destOff, srcOff and len. By using the approach presented in the paper, we tested each method of the API `Utilfs` (that has to satisfy a symmetry relation) with 54 randomly generated test data[8]. Tab. 2 contains the time elapsed to pass all the 54 tests for each method. This time value corresponds to the absolute user time (including garbage collections, operating system calls, etc.) elapsed on the 8-bit CPU Cosmo processor. It is just given here to illustrate the interest of using symmetry relations as automatic (partial) test oracles. This time should be compared to the time required by the tester to predict the expected results of the methods with each of the 54 randomly generated test data.

The test quality achieved by these tests is equals to 0.9995, that is to say each of these methods satisfies its symmetry relation with a test quality of 0.9995. We did not find any symmetry violations during this testing process but this does not prove the absence of symmetry violations as our approach is only probabilistic.

5.3 Discussion and further work

The main limitation of the Symmetric Testing paradigm arises when one tries to apply it to non-symmetric methods [15]. To address this problem, we plan to explore other properties to check the output correctness of Java Card APIs. Recently, Chen et al. proposed in [22] to use existing relations over the input data and the computed outcomes to eliminate faulty programs. Formally speaking, let $\{I_1, .., I_n\}_{n>1}$ be n distinct test data for a program intended to compute a function f and suppose that given a relation r over $\{I_1, .., I_n\}$, the results $f(I_1), .., f(I_n)$ must satisfy a property r_f, then we have: $r(I_1, .., I_n) \implies r_f(f(I_1), .., f(I_n))$. These relations, called metamorphic relations, are more general than symmetry relations. In a previous work we did [28], we proposed to automate the generation of input data that violate a given metamorphic relation, by using Constraint Logic Programming techniques. Specifying such metamorphic relations over the Java Card APIs would be interesting as they could serve as (partial) test oracles for non-symmetric methods. A similar approach would be to consider formally specified postconditions as a way to check the output correctness. For example, the formal specification of Java Card APIs written in JML (Java Modeling Language) by Poll et al. [31] could be an interesting way of getting formulas that can serve as (partial) test oracle. However, combining these formal postconditions with a random test data generator remain a non-trivial task as they make use of specific constructs that limit the possibility to asses test quality. For example, the formal JML postcondition of the `arrayCopy` Java Card API method extracted from [26] and shown in Fig.4 makes use of array accesses and a loop construct for which a fault occurrence probability seems to be difficult to establish.

```
ensures (\forall int i; (i<=0 & i<dest.length)
==>  (destOff <=i & i<destOff+length) ?
       dest[i] == src[srcOff + (i-destOff)] :
       dest[i] == \old(dest[i]) );
ensures \result == destOff+length ;
```

Figure 4: JML postconditions for `arrayCopy`

[8]All the randomly generated arrays are of size `0x7F` which is the greatest byte value

6. CONCLUSION

In this paper, we have introduced a software testing framework for on-card testing of symmetric Java Card API methods. The framework contains a semi-empirical model to help deciding when to stop testing and how to assess test quality. We have reported on a first experience on testing a few methods of the OCS Cosmo 32 RSA V3.4 Java Card API by using the Symmetric Testing paradigm. Further work will be dedicated to the exploitation of non-symmetric properties to check the output correctness of Java Card methods, such as metamorphic relations or postconditions extracted from a formal specification. Another perspective will consist in exploring how Symmetric Testing can be tuned to deal with the resources consumption problem. Due to its limited memory and execution features, Java cards and Java Card APIs must be thoroughly tested w.r.t. memory and time consumption. Symmetry relations combined with random testing could be an interesting candidate to find counter-examples of statically estimated consumption bounds but this remains to be shown.

7. REFERENCES

[1] J. P. Bowen, K. Bogdanov, J. Clark, M. Harman, R. Hierons, and P. Krause. FORTEST: Formal methods and testing. In *COMPSAC 02: 26th IEEE Annual Int. Computer Software and Applications Conf., Oxford, UK*, pages 91–101. Computer Society Press, Aug. 2002.

[2] ISO International Standard 15408. *Common Criteria for Information Technology Security Evaluation*, Aug. 1999. CCIMB-99-033, Part 3: Security assur. req.

[3] B. Legeard and F. Peureux. Generation of functional test sequences from b formal specification : presentation and industrial case study. In *In Proc. of ASE'01*, IEEE Computer Society Press, pages 377–381, San Diego, USA, Nov. 2001.

[4] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. Bz-testing-tools: A tool-set for test generation from z and b using constraint logic programming. In *In Proc. of FATES'02, Formal App. to Testing of Software, Workshop of CONCUR'02*, pages 105–120, Brn, Czech Republic, Aug. 2002.

[5] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from z and b. In *In Proc. of FME'02, Formal Methods Europe*, Springer Verlag LNCS 2391, pages 21–40, Copenhaguen, Denmark, Jul. 2002.

[6] A. Pretschner, O. Slotosch, H. Ltzbeyer, E. Aiglstorfer, and S. Kriebel. Model based testing for real: The inhouse card case study. In *In Proc. 6th Intl. Workshop on Formal*

Table 2: Symmetry in the OCS Utilfs methods

Java Card methods	User time elapsed
`short arrayAndNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short len)`	13 min 59 sec
`short arrayCompare( byte[] src, short srcOff, byte patByte, short length)`	6 min 15 sec
`short arrayFindByte( byte[] src, short srcOff, short len, byte pattern)`	4 min 24 sec
`short arrayOrNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short length)`	13 min 57 sec
`short arrayXorNonAtomic( byte[] dest, short destOff, byte[] src, short srcOff, short length)`	13 min 55 sec
`short getObjectIndex( java.lang.Object[] src, short srcOff, short n, java.lang.Object pattern)`	2 min 01 sec
`short getShortIndex( short[] src, short srcOff, short n, short pattern)`	2 min 05 sec

Methods for Industrial Critical Systems (FMICS'01), pages 79–94, Paris, Jul. 2001.

[7] D. Clarke, T. Jeron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *In Int. Conf. on Research in Smart Cards (e-Smart'01)*, Springer Verlag, LNCS 2140, pages 58–70, 2001.

[8] H. Martin and L.d. Bousquet. Automatic test generation for java-card applets. In Isabelle Attali and Thomas P. Jensen, editors, *Java on Smart Cards: Programming and Security, First International Workshop, JavaCard 2000, Cannes, France, September 14, 2000, Revised Papers*, volume 2041 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2001.

[9] J.W. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Trans. on Soft. Eng.*, 10(4):438–444, Jul. 1984.

[10] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Journal of Sotware Testing, Verification and Reliability*, 1(2):5–25, July 1991.

[11] T. Y. Chen, T. H. Tse, and Y. T. Yu. Proportional sampling strategy: a compendium and some insights. *The Journal of Systems and Software*, 58 (2001), pages 65–81. Elsevier 2001.

[12] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation*. The Benjamin/Cummings Publishing Company, INC., Menlo Park, CA, 1987.

[13] T. Wood, K. Miller, and R. E. Noonan. Local exhaustive testing: a software reliability tool. In *Proc. of the Southeast regional conf.*, pages 77–84. ACM Press, 1992.

[14] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4), 1982.

[15] A. Gotlieb. Exploiting symmetries to test programs. In *IEEE International Symposium on Software Reliability and Enginering (ISSRE)*, pages 365–374, Denver, CO, USA, Nov. 2003.

[16] SUN Microsystems. *Java Card 2.1.1 Application Programming Interface*, May 2000.

[17] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[18] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design: An International Journal*, 9(1/2):41–75, August 1996.

[19] P. Godefroid. Exploiting symmetry when model-checking software. *FORTE'99*, pp 257-275.

[20] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *SPIN*, pp 1–19, 2000.

[21] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *Soft. Tools for Tech. Transfer*, 2, 1998.

[22] T.Y. Chen, T.H. Tse, and Zhiquan Zhou. Fault-based testing in the absence of an oracle. In *IEEE Int. Comp. Soft. and App. Conf. (COMPSAC)*, pages 172–178, 2001.

[23] T.Y. Chen, T.H. Tse, and Zhiquan Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA)*, pages 191–195, 2002.

[24] T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *4th Ibero-American Symp. on Software Engineering and Knowledge Engineering (JIISIC 04)*, pages 569–583, 2004.

[25] M. A. Armstrong. *Groups and Symmetry, UTM*. Springer Verlag, 2nd ed., 1988.

[26] C.-B. Breunesse, N. Catao, M. Huisman, and B.P.F. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55(1-3):53–80, 2005.

[27] R.A. DeMillo and J.A. Offut. Constraint-based automatic test data generation. *IEEE Trans. on Soft. Eng.*, 17(9):900–910, Sep. 1991.

[28] A. Gotlieb and B. Botella. Automated metamorphic testing. In *27th IEEE COMPSAC'03*, Dallas, TX, USA, November 2003.

[29] U. Hansmann, M.S. Nicklous, T. Schack, and F. Seliger. *Smart Card Application Development using Java*. Springer Verlag, 2000.

[30] Y. Malaiya, M.N. Li, J.M. Bieman, and R. Karcich. Software reliability growth with test coverage. *Trans. on Reliability*, 51(4):420–426, Dec. 2002.

[31] H. Meijer and E. Poll. Towards a full formal specification of the java card api. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001*, LNCS 2140, pp 165–178. Springer Verlag, 2001.

[32] S. Ntafos. On Random and Partition Testing. *Soft. Eng. Notes*, 23(2):42–48, 1998.

[33] Oberthur Card Systems. *Cosmo 32 RSA V3.4 API Reference Guide*, 2003.

[34] J. Offutt, Y. Ma, and Y. Kwon. An experimental mutation system for java. *Softw. Eng. Notes*, 29(5):1–4, 2004.