Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the
10th International Workshop on
Automated Verification of Critical Systems
(AVoCS 2010)

The Belgian Electronic Identity Card: a Verification Case Study

Pieter Philippaerts, Frédéric Vogels, Jan Smans[1], Bart Jacobs, Frank Piessens

15 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

[1] Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

The Belgian Electronic Identity Card: a Verification Case Study

Pieter Philippaerts, Frédéric Vogels, Jan Smans[†], Bart Jacobs, Frank Piessens

IBBT-DistriNet, Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200a, B3001 Leuven, Belgium

Abstract: In the field of annotation-based source code level program verification for Java-like languages, separation-logic based verifiers offer a promising alternative to classic JML based verifiers such as ESC/Java2, the Mobius tool or Spec#. Researchers have demonstrated the advantages of separation logic based verification by showing that it is feasible to verify very challenging (though very small) sample code, such as design patterns, or highly concurrent code. However, there is little experience in using this new breed of verifiers on real code. In this paper we report on our experience of verifying several thousands of lines of Java Card code using VeriFast, one of the state-of-the-art separation logic based verifiers. We quantify annotation overhead, verification performance, and impact on code quality (number of bugs found). Finally, our experiments suggest a number of potential improvements to the VeriFast tool.

Keywords: verification, VeriFast, separation logic, Java Card

1 Introduction

Software verification is finally reaching a point where it is possible to verify relatively complex applications written in popular programming languages. Even though it is still often a significant effort to annotate applications in order to help them get verified automatically, the benefits outweigh the cost for a number of software markets. In particular, software with a very high cost of failure (for example, airplane controllers) or software for systems that are difficult to update after deployment (for example, smart cards) are perfect candidates for software verification.

VeriFast [JSP10] is a verifier for single-threaded and multithreaded C and Java programs annotated with separation logic specifications. The approach enables programmers to ascertain the absence of invalid memory accesses, including null pointer dereferences and out-of-bounds array accesses, as well as compliance with programmer-specified method preconditions and postconditions.

This paper will assess the applicability of verification of Java Card applets using the VeriFast approach. Two non-trivial applets are annotated and an analysis of the verification effort and results is made. Section 2 introduces the VeriFast tool and gives a short introduction to Java Card technology. Section 3 describes the applets that were used in this case study and gives a short overview of some of the solutions we used to annotate certain features. Section 4 evaluates the results of the case study and Section 5 summarizes the future work. Finally, Section 6 concludes the paper.

[†] Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

2 Background

The results in this paper build on two technologies that will be presented in this section. Section 2.1 presents a short overview of the verification technology used in this case study, and Section 2.2 introduces the features of the Java Card platform relevant to the applets being verified.

2.1 VeriFast

VeriFast [JSP10] is a verifier for Java and C programs annotated with separation logic [ORY01] specifications. The tool modularly checks via symbolic execution [BCO05] that each method in the program satisfies its specification. If VeriFast deems a Java program to be correct, then that program does contain neither null dereferences, array indexing errors, API usage violations nor data races. Moreover, all user-specified assertions are guaranteed to hold.

At the heart of the separation logic lies the concept of permissions. In particular, a method can only access a field if it has permission to do so. For example, consider the class `Interval` shown below. `o.low |-> v` denotes (1) the permission to access (read and write) the field `low` of the `Interval` object `o` and (2) that the current value of that field is `v`.

Listing 1: VeriFast annotations in Java code

```
/*@ predicate interval(Interval i, int l, int h) =
  i.low |-> l &*& i.high |-> h &*& l <= h;
@*/

public class Interval {
  int low, high;

  void shift(int amount)
    //@ requires interval(this, ?low, ?high);
    //@ ensures interval(this, low + amount, high + amount);
  {
    //@ open interval(this, low, high);
    this.low += amount;
    this.high += amount;
    //@ close interval(this, low + amount, high + amount);
  }
}
```

To distinguish full (read and write) from read-only access, permissions are qualified with fractions [Boy03] between 0 (exclusive) and 1 (inclusive), where `1` corresponds to full access and any other fraction represents read-only access. For example, `[f]o.low |-> v` denotes read-only access if `f` is less than `1` and full access if `f` equals `1`. We typically omit explicitly writing `[1]` for full permissions. Permissions can be split and merged as required during the proof. For example, two read-only permissions `[1/2]o.low |-> v` and `[1/2]o.low |-> v` can be combined into a single full permission `[1]o.low |-> v` and vice versa. In the context of

Java Card, we rely on fractional permissions to check that fields are not assigned to outside of transactions.

To abstract over the set of permissions required by a method, permissions can be grouped and hidden via predicates. For example, consider the predicate `interval` shown above. This predicate groups the permissions to access `low` and `high`, and additionally states that the value of `low` must be less or equal to the value of `high`. Just like basic permissions, predicates can be split and merged as required during the proof.

Each method in the program has a corresponding method contract consisting of a pre- and postcondition that respectively describe the permissions required and returned by the method. More specifically, the permissions described by the precondition conceptually transfer from the caller to the callee on entry to the method, and vice versa for the postcondition when the method returns. For example, consider the method `shift` in the class `Interval`. `shift`'s precondition states that the method may only be called if **`this`** points to a valid interval (where the meaning of "valid interval" is determined by the predicate `interval`). The precondition imposes no restriction on the interval's bounds, but binds the lower bound to the variable `low` (indicated by the question mark) and the upper bound to `high`. The postcondition ensures that **`this`** is still a valid interval, and its bounds have been shifted by `amount` with respect to the method pre-state. Note that our verification tool requires all annotations to be written inside special comments (`/*@ ... @*/`) which are ignored by the Java compiler but recognized by our verifier.

VeriFast does not automatically fold and unfold predicate definitions. Instead, folding and unfolding must be done explicitly by developers via ghost commands (unless the predicate is marked precise). For example, the open statement in the body of shift unfolds the definition of the predicate `interval`, and similarly the close statement folds the definition. Verification of the code snippet shown above fails if any of the ghost statements is removed.

In addition to static predicates (placed outside of a class), VeriFast also supports instance predicates (placed inside of a class). Just like instance methods, instance predicates are dynamically bound. That is, the variable **`this`** is an implicit argument to each instance predicate, and its dynamic type determines the exact meaning of the predicate. For example, the interface `Vehicle` shown below defines the instance predicate `valid`. The meaning of valid depends on the subclass at hand. For example, `o.valid()` denotes the permission to read the field `maxspeed` if the dynamic type of `o` is `Car`. In the context of Java Card, we use instance predicates to describe consistency conditions for applets (i.e. the invariant that must be preserved by each transaction).

Listing 2: Instance predicates.

```
interface Vehicle {
  //@ predicate valid();
}
class Car implements Vehicle {
  int maxspeed;

  //@ predicate valid() = [1/2]this.maxspeed |-> _;
}
```

The extended static checker for Java (Esc/Java) [FRL$^+$02] is another program verifier that has been used to verify Java Card programs [MP10, CH]. However, Esc/Java is unsound [LNS00, Appendix C.0]. This means that Esc/Java can fail to detect certain bugs. For example, the extended static checker reasons incorrectly about object invariants in the presence of reentrant calls. Unlike Esc/Java, the VeriFast methodology has been proven to be sound [JP11].

Gomes *et al.* [GMD05] have investigated using the B method to generate correct Java Card implementations from abstract models via refinement. Contrary to the B method, VeriFast does not start from an abstract model, but instead reasons directly about the applet's source code. The advantage of our approach is that we can retroactively prove correctness of existing implementations.

VeriFast performs *modular* verification. This means that the verifier analyzes each method in isolation using only method contracts (not the callees' implementations) to reason about method calls. An advantage of modular reasoning is that verification scales (verification times remain low) and that deep properties can be specified and checked. A disadvantage however is that the developer must write annotations at method boundaries. To avoid having to write annotations, Huisman *et al.* [HGSC04] have used model checking to find bugs in Java Card applications. Unlike VeriFast, Huisman *et al.* do not aim to prove the absence of all errors, but only of certain undesired applet interactions.

Mostowski [Mos07] has written a specification for the Java Card API in dynamic logic. In addition, he has used this specification to verify a number of applets using the Key verifier. A recurring problem encountered during these case studies was bad prover performance. For example, Mostowski states that "*it is not uncommon for the prover to run over an hour to finish the proof of one method*". Contrary to [Mos07], we use separation logic to specify the Java Card API. While separation logic has proven to be a powerful specification formalism for reasoning about complex (but small) examples such as design patterns and highly concurrent code, there is only limited experience in applying separation logic to larger, realistic programs. This paper reports on our experience in applying separation logic to verify realistic Java Card code. An explicit goal of VeriFast is to keep verification times low. For example, the time needed to verify full functional correctness of a single method is typically under one second.

2.2 Java Card

The Java Card platform [Ora11] was initially launched by Sun in 1996 and aimed to simplify the development of smart card applications. Until then, smart card code was largely written in C, which is difficult to write in the first place, and also has distinct disadvantages in terms of security and reliability.

The Java Card platform allowed developers to write smart card applets in a subset of the Java language that targets a specifically optimized Java framework for smart cards. The older (and most popular) platform, now called Java Card Classic Edition, does not support floating point operations, strings, multi-threading, garbage collection, stack inspection, multidimensional arrays, reflection, etc. The newest Java Card 3.0 Connected Edition supports more features but is still lacking compared to the full Java language and framework.

Java Card is now the dominant platform for smart cards, with applications for GSM, 3G, finance, PKI, e-commerce and e-government. Due to the absence of serious competition and the improvements of the latest incarnation of the Java Card platform, it can be expected that this will remain the case in the near future.

2.2.1 Applets

The entry point of each Java Card applet is a class that extends the built-in, abstract class `javacard.framework.Applet`. This class defines a number of methods that are called by the Java Card runtime to interact with the applet. In particular, the class `Applet` defines an abstract method `process` that must be overridden by the subclass. The implementation of `process` forms the core of the applet. More specifically, `process` takes an apdu (i.e. a wrapper around a byte array) as input, processes it, and possibly returns an updated apdu to the runtime. Typically, the apdu contains both information on the action that should be performed by the applet and data associated with that action.

A subclass of `javacard.framework.Applet` is a valid applet only if it declares a static method called `install`. The goal of this method is to create a new applet instance and to register this instance with the runtime. The class `MyApplet` shows the prototypical structure of a Java Card applet.

Listing 3: The prototypical structure of an applet.

```
class MyApplet extends Applet {
  public static void install(byte[] arr, short offset, byte length) {
    MyApplet applet = new MyApplet();
    // initialize the applet
    applet.register();
  }

  public void process(APDU apdu) {
    // process the apdu
  }
}
```

2.2.2 Transactions

Java Card applets use two types of memory to store data and intermediate results. Fields and objects are stored in persistent EEPROM memory, whereas the stack (and hence local variables) are stored in volatile RAM memory. In addition, the applet can also choose to allocate arrays in RAM memory, because this type of memory is faster and is harder for attackers to read. This complicates things because the smart card may lose power at any time during the computation, which results in the RAM memory being wiped, whereas the EEPROM memory retains the intermediate results.

To preserve consistency of the data stored in persistent memory, Java Card supports transactions. More specifically, the platform defines three methods to interact with the transaction mechanism: `beginTransaction`, `commitTransaction`, and `abortTransaction`. When `beginTransaction` is called, all changes to persistent memory are made conditionally. Only when a call to `commitTransaction` is executed, the changes to the persistent memory are committed atomically. If `abortTransaction` is called instead, or if the card suddenly loses power before calling `commitTransaction`, the persistent memory is restored to its original state (on card boot-up when power is restored). Note that the transaction mechanism does not impact values stored in RAM. Incorrect use of the transaction API, for example calling `beginTransaction` while a transaction is already in progress, results in an exception.

2.2.3 Java Card and VeriFast

VeriFast was originally developed for C and Java programs, but has been modified to also support Java Card applications. The Java language used for Java Card applications is a precise subset of the full Java language, thus adding Java Card support to VeriFast was easy.

Java Card does, however, use a very different class library optimized for smart cards. VeriFast needs to know for every function in the library the pre- and postconditions in order to reason about code. These specifications are placed in a separate file that defines all the classes and methods in the Java Card framework. The specifications are based on the descriptions of these methods in the official Java Card documentation. The actual implementation of these library functions is not checked.

Building the specification file is an incremental process. VeriFast only needs pre- and postconditions for the methods that are actually used by the applications you want to verify. Hence, only a subset of the full Java Card class library has been annotated in the specification file. It is critical that the specifications of library functions is correct; errors in their annotations could lead to errors in the verification process. Therefore, extreme diligence is used when adding new function definitions to the specification.

3 Case Study

Software verification is still a very time-consuming process. Existing or new source code must be annotated in order to express assumptions and invariants, and to let the verifier reason about the code. Minimizing these required annotations is an active field of research where a lot of work remains to be done. For current verification technologies the overhead of annotating code is far from negligible, so it is not (yet) economically profitable to try to annotate and verify every piece of code. Large, non-critical code bases are examples where the effort probably is not worth the hassle.

However, smart card applications have a number of properties that *do* make them ideal candidates for software verification. First of all, they are typically small, in the order of a few thousand lines of code. Secondly, they are critical, in the sense that they usually offer some kind of security service. And last but not least, it is extremely difficult to update the code once it has been deployed. If a serious bug is discovered in the code, it might be necessary to recall all the deployed smart cards and issue new ones, which could be a commercial disaster.

This paper reports on the verification of two Java Card applets: one large open source applet that implements a clone of the Belgian Electronic Identity Card, and another smaller commercial applet.

The remainder of this section focusses on the larger, open source applet. Unfortunately, due to contractual constraints we are not allowed to discuss the details of the commercial applet.

3.1 The Belgian Electronic Identity Card

The Belgian Electronic Identity Card (eID) was introduced in 2003 as a replacement for the existing non-electronic identity card. Its purpose is to enable e-government and e-business scenarios where strong authentication is necessary. The card has the size of a standard credit card and features an embedded chip. In addition to containing a machine readable version of the information printed on the card, the chip also contains the address of the owner and two RSA key pairs with the corresponding X509 certificates. One key pair is used for authentication, whereas the other key pair can be used to generate legally binding electronic signatures.

The card is implemented on top of the Java Card platform (Classic Edition) and implements the smart card commands as defined in the ISO7816 standard. Unfortunately, the actual code that runs on the eID cards is not publicly available. For our case study, we used an open source, cloned version of the eID applet that implements the same functionality as the real eID card[1]. It is aimed at developers who wish to interact with eID cards as an easy to use and customizable testing platform.

The eID implementation consists of one large class called `EidCard` and a few other small helper classes. The `EidCard` class inherits from the `Applet` class and encapsulates about 80% of the entire code base. It is a complex class of about 900 lines of code and no less than 38 fields.

3.2 Specification of Transaction Correctness

Java Card offers transactions to preserve consistency of the data stored in persistent memory. However, what does it mean for an applet to be consistent? In VeriFast, developers can explicitly write down what fields are part of the persistent state together with the desired consistency conditions. More specifically, the class `Applet` defines an instance predicate called `valid`. Each subclass must override this predicate. The implementation of the predicate given in the subclass defines the consistency conditions for the applet at hand. For example, consider the applet class `ExampleApplet` shown below. The predicate `valid` indicates that both the fields `arr` and `i`, and the array pointed to by `arr` are part of the persistent state (line 6). Moreover, the predicate imposes the consistency condition that `i` is a valid index in `arr` (line 7).

While reading fields is possible at any time, updates to persistent memory should be made inside of a transaction. The permission system used by VeriFast is the key to enforcing this property. More specifically, at the start of the `process` method, no transaction is in progress. As shown in Listing 5, the precondition of `process` contains $1/2$ of the `valid` predicate. This means that the method can read but not update fields included in `valid` (as the method only has one half of the permissions included in `valid`). The predicate `current_applet` is simply a token describing the currently active applet.

To update the fields of the applet, the method should somehow gain additional permissions (namely the other half of the `valid` predicate). These additional permissions can be acquired by calling `beginTransaction`. In particular, the postcondition of `beginTransaction`

[1] The code can be downloaded from http://code.google.com/p/eid-quick-key-toolset/

Listing 4: The contract of the `process` method, using fractional permissions.

```
class ExampleApplet extends Applet {
  short i;
  short[] arr;
  /*@
  predicate valid() =
    this.arr |-> ?arr &*& this.i |-> ?i &*&
    array_slice(arr, 0, ?len, _) &*&
    0 <= i &*& i < len;
  @*/
}
```

Listing 5: The contract of the `process` method, using fractional permissions.

```
public void process(...)
    //@ requires current_applet(this) &*& [1/2]valid() &*& ...;
    //@ ensures current_applet(this) &*& [1/2]valid() &*& ...;
{
    ...
}
```

shown in Figure 6 gives 1/2 of the `valid` predicate. The `process` method can then merge `[1/2]valid()` (gained from the precondition of `process`) and `[1/2]valid()` (gained from the postcondition of `beginTransaction`) into `[1]valid()`. The full permission to `valid` gives the applet the right to modify the applet's fields for the duration of the transaction. When calling `commitTransaction`, half of the permissions included in the `valid()` predicate return to the system again. Note that it is impossible to call `endTransaction` if the applet is in an invalid state (according to the conditions described by `valid`), as the precondition of `commitTransaction` requires the consistency conditions to hold.

Listing 6: The declaration of the `beginTransaction` and `commitTransaction` methods

```
public static void beginTransaction();
    //@ requires current_applet(?a) &*& ...;
    //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;

public static void commitTransaction();
    //@ requires current_applet(?a) &*& a.valid() &*& ...;
    //@ ensures current_applet(a) &*& [1/2]a.valid() &*& ...;
```

3.3 Inheritance

The ISO7816 standard specifies a mechanism to access files that are stored on a smart card. Three types of files are defined:

1. **Master files** represent the root of the file system. Each smart card contains at most one master file.
2. **Elementary files** contain actual data.
3. **Dedicated files** behave like directories. They can contain other dedicated or elementary files.

To represent this structure, the eID implementation uses helper classes that form a class hierarchy. The root of the hierarchy is the abstract `File` class. This class has two subclasses: `DedicatedFile` and `ElementaryFile`. And finally, the `MasterFile` class inherits from `DedicatedFile`.

When a class is defined in the source code, it can be annotated with a predicate that represents an instance of that class. These predicates can then be used elsewhere to represent a fully initialized instance of that class. Listing 7 shows how a `File` predicate can be defined for the corresponding `File` class. The class consists of two fields, which are also represented in the predicate. The predicate can also contain other information about the class such as invariants.

Listing 7: A first definition of the *File* class and predicate.

```
public abstract class File {
   /*@ predicate File(short theFileID, boolean activeState) =
       this.fileID |-> theFileID &*&
       this.active |-> activeState; @*/

   private short fileID;
   protected boolean active;

   ...
}
```

The `ElementaryFile` class redefines the `File` predicate as shown in lines 2-4 of Listing 8. A `File` predicate that is associated with an `ElementaryFile` class is defined as an `ElementaryFile` predicate where three of the five parameters are undefined.

The definition of the `ElementaryFile` predicate (lines 5-13) consists of a link to the `File` predicate defined in Listing 7 and some extra fields and information that are specific to elementary files.

When an object is cast from the `File` to the `ElementaryFile` class (or vice versa), the corresponding predicate on the symbolic heap must be changed as well. We ‘annotated’ this by adding the methods that are defined in Listing 9 to the `ElementaryFile` class and calling these methods when required. Obviously, this solution is far from elegant because it requires adding calls to stub functions in the code of the applet. The most recent version of VeriFast supports annotating this behavior as lemma methods (i.e. methods defined inside an annotation), removing the requirement of modifying the applet’s code.

One problem that occurs with the methods presented in Listing 9 is that information is lost when an `ElementaryFile` is cast to a `File` and then back again to an `ElementaryFile`.

Listing 8: A first definition of the *ElementaryFile* class and predicate.

```
public final class ElementaryFile extends File {
   /*@ predicate File(short theFileID, boolean activeState) =
         ElementaryFile(theFileID, ?dedFile, ?dta,
           activeState, ?sz); @*/
   /*@ predicate ElementaryFile(short fileID,
           DedicatedFile parentFile, byte[] data,
           boolean activeState, short size) =
         this.File(File.class)(fileID, activeState) &*&
         this.parentFile |-> parentFile &*&
         this.data |-> data &*& data != null &*&
         this.size |-> size &*&
         array_slice(data, 0, data.length, _) &*&
         size >= 0 &*& size <= data.length; @*/

   private DedicatedFile parentFile;
   private byte[] data;
   private short size;

   ...
}
```

This loss of information happens in the `castFileToElementary` method where three parameters are left undefined.

There are some instances in the eID applet where this loss of information is problematic. The solution was to extend the `File` and `ElementaryFile` predicates to contain an extra parameter that can store any information. The result can be seen in Listing 10. Line 3 shows the definition of this extra parameter. In the case of the `File` class, no extra information is kept and the parameter is defined to be empty (denoted as 'unit' on line 5). Similarly, line 22 defines the parameter to be empty for the `ElementaryFile` predicate, because all state information that can be stored in the predicate is fully defined by the other parameters.

Line 14 shows the case where the predicate needs the extra parameter to store additional information about the object. In this case, the `info` parameter stores a quad-tuple of extra information that can be used to correctly initialize the embedded `ElementaryFile` predicate without losing information.

4 Evaluation

The main goal of this case study was to see how practical it is to use VeriFast to annotate a Java Card applet that is more than a toy project. It gives us an idea of how much the annotation overhead is, where we can improve the tool, and whether we can actually find bugs in existing code using this approach.

Listing 9: Functions to cast predicates.

```
public void castFileToElementary()
    //@ requires [?f]File(?fid, ?state);
    //@ ensures [f]ElementaryFile(fid, _, _, state, _);
{
    //@ open [f]File(fid, state);
}

public void castElementaryToFile()
    //@ requires [?f]ElementaryFile(?fid, ?dedFile, ?dta, ?state, ?sz);
    //@ ensures [f]File(fid, state);
{
    //@ close [f]File(fid, state);
}
```

4.1 Annotation Overhead

The more information the developer gives in the annotations about how the applet should behave, the more VeriFast can prove about it. It is up to the developer to choose whether he wants to use VeriFast as a tool to only detect certain kinds of errors (unexpected exceptions and incorrect use of the API), or whether he wants to prove full functional correctness of the applet. Both modi operandi are supported by the tool. For the Java Card applets, we used the annotations to prove that the applet does not contain transaction errors, performs no out of bounds operations on buffers, and never dereferences null pointers. While we have used VeriFast to verify full functional correctness of sequential and fine-grained concurrent data structures [JP11], specifying and verifying full functional correctness of JavaCard applets is future work.

The eID applet and helper classes consist of 1,004 lines of Java Card code. In order to verify the project, we added 802 lines of VeriFast annotations (or about four lines of annotations for every five lines of code). The majority of these annotations were **requires**/**ensures** pairs (88 pairs, one for each method) and **open** and **close** statements (99 and 112 instances respectively). Remarkably, only 8 predicates are defined throughout the entire code base, reflecting the design decision of the authors of the applet to write most of it as one huge class file.

The commercial applet consists of 251 lines of Java Card code, which we annotated with 205 lines of VeriFast annotations. There were 13 **requires**/**ensures** pairs, 25 **open** statements and 29 **close** statements.

Another type of annotation overhead is the time it took to actually write the annotations. The verification of the eID applet was performed by a senior software engineer without prior experience with the VeriFast tool, but with regular opportunities to consult VeriFast expert users during the verification effort. We did not keep detailed effort logs, but a rough estimate of the effort that was required is 20 man-days. This includes time spent learning the VeriFast tool and the Java Card API specifications. The commercial applet was annotated by a VeriFast specialist and took about 5 man-days, excluding the time it took to add some new required features to the tool.

Listing 10: A more complete definition of the *File* and *ElementaryFile* predicates that supports downcasting.

```
public abstract class File {
    /*@ predicate File(short theFileID, boolean activeState,
            any info) =
        this.fileID |-> theFileID &*&
        this.active |-> activeState &*& info == unit; @*/

    ...
}

public final class ElementaryFile extends File {
    /*@ predicate File(short theFileID, boolean activeState,
            quad<DedicatedFile, byte[], short, any> info) =
        ElementaryFile(theFileID, ?dedFile, ?dta, activeState,
        ?sz, ?ifo) &*& info == quad(dedFile, dta, sz, ifo); @*/
    /*@ predicate ElementaryFile(short fileID,
            DedicatedFile parentFile, byte[] data, boolean activeState,
            short size, any info) =
        this.File(File.class)(fileID, activeState, _) &*&
        this.parentFile |-> parentFile &*&
        this.data |-> data &*& data != null &*& this.size |-> size
        &*& array_slice(data, 0, data.length, _) &*&
        size >= 0 &*& size <= data.length &*& info == unit; @*/

    ...
}
```

4.2 Bugs and Other Problems

Because the eID applet in our case study is aimed at developers, the authors did not spend a lot of time worrying about card tearing. This is demonstrated by the fact that they did not use the Java Card transaction system at all. Using VeriFast, we found 25 locations where a card tear could cause the persistent memory to enter an inconsistent state.

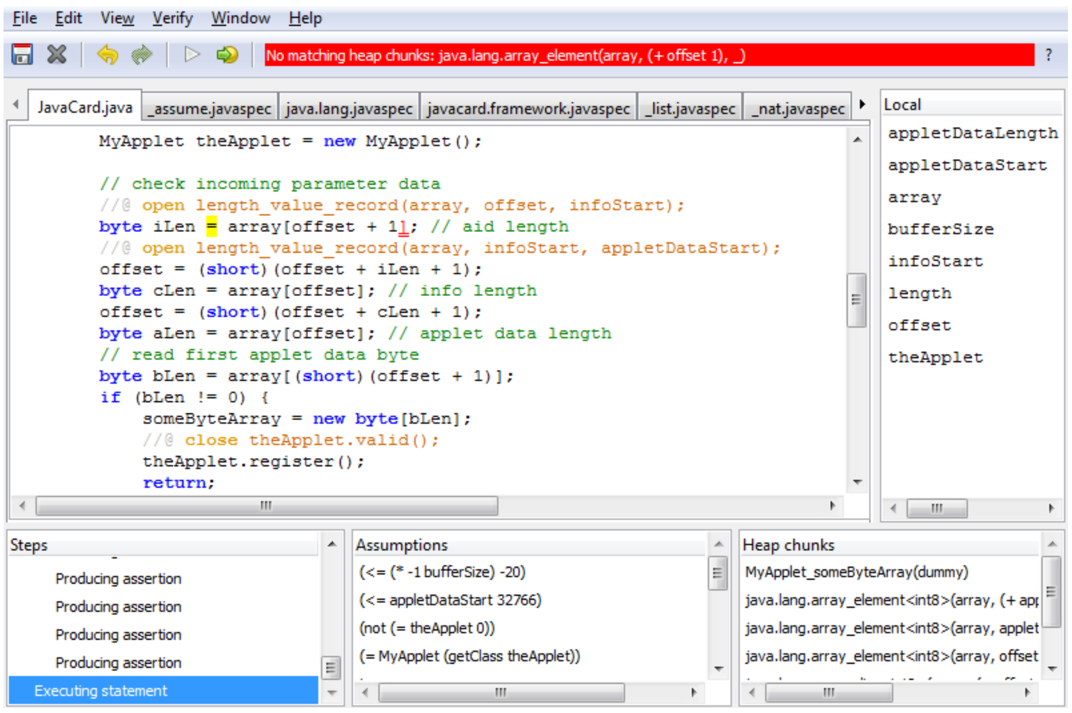
Three locations were found where a null pointer dereference could occur. An additional three class casting problems were found, where a variable holding a reference to the selected file (of type `File`) was cast to an `ElementaryFile` instance. These bugs could be triggered by sending messages with invalid file identifiers to the smart card. Seven potential out of bounds operations were also found in the code. These bugs could be triggered by sending illegal messages to the smart card.

We also found a number of bugs in the commercial applet, even though it had already been verified with another verification technology previously. We found an unsafe API call, a handful of unchecked assumptions about incoming apdus, and four locations where transactions were not used properly. Clearly, the tool used earlier was not sound or was not used in a sound way.

4.3 VeriFast Strengths

Compared to other program verifiers that target Java Card [Mos07, FRL+02], VeriFast has two advantages: speed and soundness. That is, VeriFast usually reports in only a couple of seconds (usually less) whether the applet is correct or whether it contains a potential bug. Secondly, if VeriFast deems a program to be correct, then that program is guaranteed to be free from null pointer and array index out of bounds exceptions, and API usage and assertion violations.

A feature that proved to be crucial in understanding failed verification attempts is VeriFast's symbolic debugger. As shown in Figure 1, the symbolic debugger can be used to diagnose verification errors by inspecting the symbolic states encountered on the path to the error. For example, if the tool reports an array indexing error, one can look at the symbolic states to find out why the index is incorrect. This stands in stark contrast to most verification condition generation-based tools that simply report an error, but do not provide any help to understand the cause of the error.



```
No matching heap chunks: java.lang.array_element(array, (+ offset 1), _)
```

```
        MyApplet theApplet = new MyApplet();

        // check incoming parameter data
        //@ open length_value_record(array, offset, infoStart);
        byte iLen = array[offset + 1]; // aid length
        //@ open length_value_record(array, infoStart, appletDataStart);
        offset = (short)(offset + iLen + 1);
        byte cLen = array[offset]; // info length
        offset = (short)(offset + cLen + 1);
        byte aLen = array[offset]; // applet data length
        // read first applet data byte
        byte bLen = array[(short)(offset + 1)];
        if (bLen != 0) {
            someByteArray = new byte[bLen];
            //@ close theApplet.valid();
            theApplet.register();
            return;
```

```
appletDataLength
appletDataStart
array
bufferSize
infoStart
length
offset
theApplet
```

```
Producing assertion
Producing assertion
Producing assertion
Producing assertion
Executing statement
```

```
(<= (* -1 bufferSize) -20)
(<= appletDataStart 32766)
(not (= theApplet 0))
(= MyApplet (getClass theApplet))
```

```
MyApplet_someByteArray(dummy)
java.lang.array_element<int8>(array, (+ app
java.lang.array_element<int8>(array, applet
java.lang.array_element<int8>(array, offset
```

Figure 1: The symbolic debugger of VeriFast

5 Future Work

This case study has led to a number of useful insights and showed us some of the rougher edges of the tool that need to be polished some more. Most of the issues were small and were either bugs in the tool (for instance, Java parsing errors) or functionality that was easy to implement but hadn't been done yet due to time constraints.

An important, missing feature that would greatly reduce the annotation overhead (and hence reduce the cost of verification) is automatic inference of open and close statements and of lemma applications. For example, the eID applet contains 211 open and close statements. While VeriFast already infers some ghost statements, we believe one of the most important steps to improve the verification experience is extending this inference mechanism.

Currently, only a subset of the Java Card API is supported by VeriFast. For example, we do not support multi-applet applications that communicate via the shareable interface mechanism yet. We intend to support these additional features and write specifications for all library functions in the Java Card API.

6 Conclusion

This paper reported on a case study for the VeriFast program verifier. Two non-trivial Java Card applets were annotated and verified for correctness with respect to certain common programming errors. In particular, the verification proved that the applet does not contain transaction errors, performs no out of bounds operations on buffers, and never dereferences null pointers.

The results of the case study are encouraging: with an annotation overhead of about four lines of annotations per five lines of code we found a total of 13 bugs in the eID applet, and 25 locations where transactions were not properly used.

This case study has been invaluable for us to improve the tool. A number of bugs were fixed and small additions were made in order to support the verification of the applets. A longer term plan has also been established to further add improvements and optimizations to the tool. In particular, the automatic generation of `open` and `close` statements will become an important part of the future work, as well as language and technology-specific extensions to the tool.

Acknowledgements: This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by IWT, by the Research Fund K.U.Leuven, and by the EU FP7 projects SecureChange and NESSoS.

Bibliography

[BCO05] J. Berdine, C. Calcagno, P. O'Hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 2005.

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In *Static Analysis: 10th International Symposium*. 2003.

[CH] N. Cataño, M. Huisman. Formal specification of Gemplus' electronic purse case study using ESC/Java. In *Formal Methods Europe*. Pp. 272–289.

[FRL$^+$02] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended Static Checking for Java. In *Programming Language Design and Implementation (PLDI)*. 2002.

[GMD05] B. E. G. Gomes, A. M. Moreira, D. Déharbe. Developing Java Card Applications with B. In *Brazilian Symposium on Formal Methods (SBMF)*. 2005.

[HGSC04] M. Huisman, D. Gurov, C. Sprenger, G. Chugunov. Checking Absence of Illicit Applet Interactions: A Case Study. In *Formal Aspects of Software Engineering*. 2004.

[JP11] B. Jacobs, F. Piessens. Expressive modular fine-grained concurrency specification. *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)* 46(1):271–282, 2011.

[JSP10] B. Jacobs, J. Smans, F. Piessens. A Quick Tour of the VeriFast Program Verifier. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 2010.

[LNS00] K. R. M. Leino, G. Nelson, J. B. Saxe. ESC/Java User's Manual. Technical report, Compaq Systems Research Center, 2000.

[Mos07] W. Mostowski. Fully Verified Java Card API Reference Implementation. In *International Verification Workshop (VERIFY)*. 2007.

[MP10] W. Mostowski, E. Poll. Midlet Navigation Graphs in JML. In *Brazilian Symposium on Formal Methods (SBMF)*. 2010.

[Ora11] Oracle. Java Card Technology. 2011.
http://www.oracle.com/technetwork/java/javacard/overview/

[ORY01] P. O'Hearn, J. Reynolds, H. Yang. Local Reasoning About Programs that Alter Data Structures. In *International Workshop on Computer Science Logic (CSL)*. 2001.