



# Software verification with VeriFast: Industrial case studies<sup>☆</sup>



Pieter Philippaerts<sup>\*</sup>, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans,  
Bart Jacobs, Frank Piessens

*iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001, Leuven, Belgium*

## ARTICLE INFO

### Article history:

Received 5 May 2012

Received in revised form 5 November 2012

Accepted 26 January 2013

Available online 8 February 2013

### Keywords:

Software verification  
Industrial case studies  
VeriFast

## ABSTRACT

In this article, we present a series of four industrial case studies in software verification. We applied VeriFast, a sound and modular software verifier based on separation logic, to two Java Card smart card applets, a Linux device driver, and an embedded Linux network management component, the latter two written in C. Our case studies have been carefully selected so as to evaluate the industrial applicability of VeriFast. We focus on proving the absence of safety violations, e.g., that the programs do not perform illegal operations such as dividing by zero or illegal memory accesses. Yet, given the sensitive application environment of our case studies, these safety properties typically have security implications. In this article we give a detailed description of the VeriFast approach to software verification based on two of the above case studies, one in Java and one in C. Finally, we draw conclusions on the overall feasibility of using VeriFast to verify software components in industrial domains that have stringent requirements on reliability and security.

© 2013 Elsevier B.V. All rights reserved.

## 1. Introduction

A major goal in computer science is to create a platform on which a developer is able to develop software that is both stable and secure, is easy to write and modify, and is also fast. Unfortunately, a number of these goals seem to counter each other. For example, software that is easy to write will typically be slower. It is now up to the research community to find the optimal trade-off between these properties.

Software verification can help to reach this optimal trade-off by giving developers the means to formally prove the absence of a range of bugs and potential security loophole in their code. The developer must annotate the code to indicate what it *should* do, and a verifier will then check whether the code actually does what the programmer wants it to do. Wherever possible, proof steps are automatically generated by the verifier. However, it might sometimes be necessary to manually annotate the code with proof steps in order to guide the verifier.

The field of formal software verification has made great strides in the last decade [1]. Yet, especially when considering post-hoc software verification, many existing techniques and tools are tailor-made for rather specific application domains or case studies. In particular, programming constructs that involve concurrency and dynamic memory management are often ruled out.

<sup>☆</sup> An extended abstract of this article appeared in the proceedings of AVOCS 2011: Philippaerts et al., The Belgian electronic identity card: a verification case study, in: Electronic Communications of the EASST, Vol. 46. A paper on the verification of the Linux's USB BP keyboard driver appeared in the proceedings of NFM 2012: Penninckx et al., in: Sound formal verification of Linux's USB BP keyboard driver, in: Lecture Notes in Computer Science, vol. 7226, Springer, 2012, pp. 210–215.

<sup>\*</sup> Corresponding author.

E-mail address: [Pieter.Phillippaerts@cs.kuleuven.be](mailto:Pieter.Phillippaerts@cs.kuleuven.be) (P. Philippaerts).

In this article we present a series of four non-trivial case studies in software verification. We employ VeriFast, a sound and modular software verifier based on separation logic, to two Java Smart Card applets, a Linux device driver, and an embedded Linux network management component, which are written in C. Our case studies have been carefully selected so as to evaluate the industrial applicability of VeriFast. We aim to show that VeriFast is a general-purpose software verification tool that combines an interactive verification experience with a high degree of automation and features reasoning about programs that involve concurrency and dynamic memory allocation. We focus on proving safety properties, i.e. the absence of run-time errors; proving functional correctness is possible with VeriFast but not considered in these case studies.

VeriFast [2] is a verifier for single-threaded and multithreaded C and Java programs. The approach enables programmers to prove the absence of certain safety violations such as invalid memory accesses (including null pointer dereferences and out-of-bounds array accesses), as well as compliance with programmer-specified method preconditions and postconditions.

### 1.1. Case studies

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 justifiable.

However, there are a number of areas where software verification potentially does make sense. Smart card applications for example 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.

Another example are operating system drivers that have to be extremely stable because any bug can crash the entire system. In addition, drivers typically run with elevated privileges, so bugs in driver code may have very significant security implications. Moreover, multiple threads can execute concurrently inside a driver, making it difficult to find bugs through testing and to reliably reproduce them once they are found.

This article reports on four case studies, two of which are discussed in detail. We detail our experience with a large open source Java Card applet, and a commercial C implementation of a Policy Enforcement Point for embedded Linux gateways. We also briefly describe two other case studies, one on a Linux device driver and one on an industrial Java Card applet. The commercial Policy Enforcement Point and the industrial Java Card applet originate from industry partners of the SECURECHANGE<sup>1</sup> project.

### 1.2. Organization of the article

This article assesses the applicability of the VeriFast approach to industrial code. Four case studies have been selected, two of which are Java Card applets and two C programs. Section 2 introduces the VeriFast tool and provides some background on verifying Java Card and C programs. Sections 3 and 4 describe the open source Java Card applet and Policy Enforcement Point case studies in detail; 5 briefly discusses the two remaining ones. Finally, Section 6 discusses related software verification tools and case studies, and Section 7 concludes the article and summarizes our experience.

## 2. Background

This section briefly describes the verification technology used for the case studies. Section 2.1 presents a short overview of the VeriFast approach for verification of C and Java programs. Section 2.2 introduces Java Card specific topics. Section 2.3 shows how code can be annotated and debugged using the VeriFast symbolic debugger.

### 2.1. VeriFast

VeriFast<sup>2</sup> [2] is a verifier for C and Java programs annotated with separation logic specifications [3]. The tool modularly checks via symbolic execution [4] that each function or method in the program satisfies its specification. If VeriFast deems a program to be correct, then that program does not exhibit certain run-time errors.

VeriFast's underlying logic is based on an extension of separation logic. At the heart of separation logic lies the concept of permissions [5]. That is, each activation record holds a number of permissions and it can only access a memory location if it

<sup>1</sup> Background information on SECURECHANGE is available online at <http://www.securechange.eu/>.

<sup>2</sup> VeriFast can be downloaded from <http://distrinet.cs.kuleuven.be/software/VeriFast/>. The current VeriFast distribution ships with many examples and tutorials.

holds the corresponding permission to do so: at the beginning of each function or method call, the permissions required by the callee are transferred from the caller to the callee. When a function or method returns, the permissions expected by the caller are transferred from the callee back to the caller. The VeriFast tool reports an error if either the caller lacks a permission required by the callee or if the callee lacks a permission expected by the caller. VeriFast enforces the program wide invariant that (1) if an activation record holds the permission to access a particular memory location, then that memory location is allocated and (2) that if an activation record has permission to write a memory location, then no other activation record has permission to access that memory location at the same time. By enforcing this invariant, the tool can guarantee the absence of certain run-time errors. First of all, as an activation record can only access a memory location if it has permission to do so and because of (1), verified programs do not contain illegal memory accesses. Secondly, a data race occurs when two threads simultaneously access the same memory location and at least one of these accesses is a write operation. Verified programs do not contain data races because a thread can only access a memory location if one of its activation records holds the corresponding permission and because of (2). Finally, the permission policy allows the tool to deduce an upper bound on the set of memory locations that can be modified by a function or method call: if the caller holds the permission to access a memory location and does not transfer this permission to the callee, then the callee cannot modify that memory location.

In the remainder of this section, we explain particular aspects of the verifier relevant to the case studies discussed in this paper in more detail. We first discuss verification of C programs and then turn our attention to Java.

### 2.1.1. Verification of C programs

An activation record can only access a memory location if it has permission to do so. The permission to access (i.e. both read and write) the field  $f$  of an object  $o$  with value  $v$  is denoted  $o.f \mid \rightarrow v$ . For example, consider the struct `interval` in Listing 1. The permission to access to field `low` of an `interval` pointer  $i$  with value  $l$  is denoted  $i \rightarrow \text{low} \mid \rightarrow l$ .

Listing 1. VeriFast annotations for C.

```

1  struct interval {
2      int low;
3      int high;
4  };
5
6  void shift(struct interval* i, int v)
7      /*@ requires i->low |-> ?l &*& i->high |-> ?h;
8         /*@ ensures i->low |-> l + v &*& i->high |-> h + v;
9  {
10     i->low += v;
11     i->high += v;
12 }
13
14 int get_low(struct interval* i)
15     /*@ requires [1/2]i->low |-> ?l;
16     /*@ ensures [1/2]i->low |-> l;
17 {
18     return i->low;
19 }
```

Each function in the program has a corresponding function contract consisting of a precondition (keyword `requires`) and a postcondition (keyword `ensures`). The precondition describes the permissions the function requires in order to execute successfully. For example, the function `shift` from Listing 1 requires the permissions to access  $i \rightarrow \text{low}$  and  $i \rightarrow \text{high}$ . Intuitively, these permissions are transferred from callers to `shift` when the function is called. The permissions granted by the precondition allow `shift`'s body to update  $i$ 's fields. The precondition uses  $?l$  and  $?h$  for the values of respectively  $i \rightarrow \text{low}$  and  $i \rightarrow \text{high}$ . This indicates that the precondition places no restriction on the values of both fields. Instead, the value of  $i \rightarrow \text{low}$  is bound to  $l$  and the value of  $i \rightarrow \text{high}$  to  $h$ . The postcondition describes the permissions that are transferred from the function to its caller when the function returns. For example, the function `shift` returns the permissions for accessing  $i$ 's fields. The postcondition additionally indicates that the values of both fields have been incremented by  $v$ . Note that our verification tool requires all annotations (such as pre- and postconditions) to be written inside special comments (`/*@ ... @*/`) which are ignored by the C and Java compilers but recognized by our verifier.

To distinguish full (read and write) from read-only access, permissions can be qualified with a fraction between 0 (exclusive) and 1 (inclusive), where 1 corresponds to full access and any other fraction represents read-only access. For example,  $[f]o.f \mid \rightarrow v$  denotes full access if  $f$  equals 1 and read-only access if  $f$  is less than 1. We typically omit  $[1]$  for full permissions. For example,  $i \rightarrow \text{low} \mid \rightarrow ?l$  in the precondition of `shift` is a shorthand for  $[1]i \rightarrow \text{low} \mid \rightarrow ?l$ . The function `get_low` of Listing 1 only reads  $i \rightarrow \text{low}$  and therefore its precondition only demands read-only access to  $i \rightarrow \text{low}$  instead of full access. By using the fractional permissions, multiple threads can concurrently call `get_low`. Permissions can be split and merged as required during the proof. For example, two read-only permissions  $[1/2]o.\text{low} \mid \rightarrow l$  and  $[1/2]o.\text{low} \mid \rightarrow l$  can be combined to a single full permission  $[1]o.\text{low} \mid \rightarrow l$  and the other way around. For each memory location, the sum of the fractional permissions over all activation records is at most one at all times.

**Listing 2.** Data abstraction via predicates.

```

1  /*@
2  predicate interval(struct interval* i, int l, int h) =
3    i->low /-> l &*& i->high /-> h &*& l <= h;
4  @*/
5
6  void shift(struct interval* i, int v)
7    /*@ requires interval(i, ?l, ?h);
8      /*@ ensures interval(i, l + v, h + v);
9  {
10   /*@ open interval(i, l, h);
11   i->low += v;
12   i->high += v;
13   /*@ close interval(i, l + v, h + v);
14 }
15
16 int get_low(struct interval* i)
17   /*@ requires [1/2]interval(i, ?l, ?h);
18   /*@ ensures [1/2]interval(i, l, h);
19 {
20   /*@ open [1/2]interval(i, l, h);
21   return i->low;
22   /*@ close [1/2]interval(i, l, h);
23 }

```

To abstract over the set of permissions required by a function, permissions can be grouped and hidden via predicates. For example, the predicate `interval` from Listing 2 groups and hides the permissions to access `i->low` and `i->high`. In addition, it imposes the constraint that `i->low` must be less than or equal to `i->high`. By using the predicate instead of the field permissions in the contracts of `shift` and `get_low`, the contracts are made implementation-independent. That is, the implementation of `interval` can be changed without having to modify the function contracts, and hence without having to worry about breaking or having to reverify client code. Just like basic permissions, predicates can be split and merged as required during the proof. As shown in Listing 2, we use `open` and `close` annotations to unfold and fold predicate chunks.

By default, VeriFast does not automatically fold and unfold predicate definitions. Instead, folding and unfolding must be done explicitly by developers via ghost commands. 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.

Programs typically rely on function type definitions to define the signature of function pointer parameters. For example, the function type definition `print` from Listing 3 defines a signature for functions that implement pretty printing. The function `print_twice` takes a pointer `x` and a function pointer `f` as arguments and uses `f` to print `x` twice. In VeriFast, a function contract can be associated with each function type definition. The assertion `is_print(f)` in the precondition of `print_twice` requires `f` to be a pointer to a function that satisfies the contract of `print`. However, the set of permissions required by the pretty printing function depends on the data being printed. For example, the function `print_int` requires that `x` points to a valid integer (i.e. an integer(`x`, `_`) chunk), while `print_interval` requires `x` to point to a valid interval (i.e. interval(`x`, `_`, `_`)). `print`'s contract can be defined in an implementation independent manner via a predicate family. Contrary to the regular predicates described above, a predicate family can have multiple definitions, depending on an additional parameter called the predicate family index (typically the address of a function). For example, the function type definition `print` is specified in terms of the predicate family `print_data`. Because the definition of `print_data` depends on the function used as an index, which is referred to by `this` in `print`'s contract, any data structure can be pretty printed. For example, the function `print_int` satisfies the contract of `print` (indicated by the `/*@: print` annotation), because `print_data` is defined to hold in the context of this function only if `x` is a valid pointer to an integer. Similarly, `print_interval` satisfies the contract of `print` as `print_data` is defined to hold in the context of `print_interval` only if `x` points to a valid interval.

If VeriFast deems a C program to be correct, then that program does not exhibit undefined behavior as described by the C11 standard [6]. In particular, that program does not perform illegal memory access (such as buffer overflows), memory leaks, data races, assertion violations and specified API usage violations (such as double frees).

### 2.1.2. Verification of Java programs

Just as for C programs, permissions lie at the heart of VeriFast for Java. As such, the specifications for a C program and the equivalent Java program are largely similar. As example, consider the class `Interval` from Listing 4. The specifications of `Interval` are similar to the ones shown in the C program of Listing 2.

There are two main differences between VeriFast for Java and VeriFast for C. First of all, each method and each predicate has an implicit `this` parameter. Therefore, it is not necessary to explicitly include a reference to the interval. The second and most important difference is that each non-private instance method and each predicate is dynamically bound on the dynamic type of `this`. For example, `i.interval(l, h)` denotes that `i` is valid interval according to the definition of

**Listing 3.** Predicate families.

```

1  /// predicate_family print_data(void *index)(void *x);
2
3  typedef void print(void *x);
4  /// requires print_data(this)(x);
5  /// ensures print_data(this)(x);
6
7  void print_twice(void* x, print* f)
8  /// requires print_data(f)(x) &*& is_print(f) == true;
9  /// ensures print_data(f)(x) &*& is_print(f) == true;
10 {
11     f(x); f(x);
12 }
13
14 /*
15 predicate_family_instance print_data(print_int)(int* x) =
16 integer(x, _);
17 */
18
19 void print_int(int* x) ///: print
20 /// requires print_data(print_int)(x);
21 /// ensures print_data(print_int)(x);
22 {
23     /// open print_data(print_int)(x);
24     printf("%i", x);
25     /// close print_data(print_int)(x);
26 }
27
28 /*
29 predicate_family_instance print_data(print_interval)
30 (struct interval* x) =
31 interval(x,_, _);
32 */
33
34 void print_interval(struct interval* i) ///: print
35 /// requires print_data(print_interval)(i);
36 /// ensures print_data(print_interval)(i);
37 {
38     /// open print_data(print_interval)(i);
39     /// open interval(i, _, _);
40     printf("intervalfrom%ito%i", i->low, i->high);
41     /// close interval(i, _, _);
42     /// close print_data(print_interval)(i);
43 }

```

**Listing 4.** VeriFast annotations for Java.

```

1  class Interval {
2      private int low, high;
3
4      /*
5      predicate interval(int l, int h) =
6      this.low |-> l &*& this.high |-> h &*& l <= h;
7      */
8
9      void shift(int v)
10         /// requires interval(?l, ?h);
11         /// ensures interval(l + v, h + v);
12     {
13         /// open interval(l, h);
14         this.low += v;
15         this.high += v;
16         /// close interval(l + v, h + v);
17     }
18 }

```

the predicate defined in `i.getClass()`. This means that is not only possible to override methods but also predicates. For example, the class `CountingInterval` of Listing 5 overrides both the predicate `interval` and the method `shift` in order count the number of calls to `shift`.

Note that non-final static fields are treated as instance fields of the corresponding class object. For example, a thread can only modify the static field `f` of a class `C` if it holds full permission to `C.f`. To simplify verification, final static fields with

constant initializers (such as `Integer.MAX_VALUE`) are considered to be constants during verification (meaning that no permission is required to read them).

**Listing 5.** VeriFast annotations for Java.

```

1  class CountingInterval extends Interval {
2      private int counter;
3
4      /*@
5      predicate interval(int l, int h) =
6          this.interval(Interval.class)(l, h) &*&
7          this.counter /-> ?n &*& 0 <= n;
8      @*/
9
10     void shift(int v)
11         /*@ requires interval(?l, ?h);
12          /*@ ensures interval(l + v, h + v);
13     {
14         /*@ open interval(l, h);
15         super.shift(v);
16         counter++;
17         /*@ close interval(l + v, h + v);
18     }
19 }
```

Each predicate (defined inside a class or interface) represents a predicate family instance. However, the index is not the address of a function, but the dynamic type of `this`.

In both C and Java, predicates play the role of object invariants. For example, the predicate `valid` in the class `Interval` states amongst others that the lower bound must be less than or equal to the upper bound for the interval to be in a consistent state. As advocated by Parkinson [7], separation logic does not impose any built-in rules that describe when invariants must be established, where they can be assumed and when they can temporarily be broken. Instead, if an object is required or guaranteed to be consistent, then the developer should explicitly say so in the method contract by stating that the corresponding predicate holds.

If VeriFast deems a Java program to be correct, then that program does not contain null dereferences, array indexing errors, data races, assertion violations and API usage violations.

## 2.2. The Java Card platform

The Java Card platform [8] 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.

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 subset of the full Java language. As a consequence, supporting Java Card in VeriFast comes down to defining appropriate method contracts for the Java Card API.

### 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 *Application Protocol Data Unit* (APDU) received from the card terminal as input, processes it, and possibly returns an updated APDU back to the terminal. An APDU is a byte sequence, accessed through a byte array called the *APDU buffer*, whose meaning is partially standardized and partially application-specific.

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` of Listing 6 shows the prototypical structure of a Java Card applet.

**Listing 6.** The prototypical structure of an applet.

```

1  class MyApplet extends Applet {
2      public static void install(byte[] arr, short offset, byte length) {
3          MyApplet applet = new MyApplet();
4          // initialize the applet
5          applet.register();
6      }
7
8      public void process(APDU apdu) {
9          // process the apdu
10     }
11 }

```

### 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. Specifically, the platform defines three methods to interact with the transaction mechanism: `beginTransaction`, `commitTransaction`, and `abortTransaction`. When `beginTransaction` is called, all subsequent 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.

### 2.2.3. Integration with VeriFast

VeriFast needs to know for every library method the pre- and postconditions in order to reason about client code. In the case of the Java Card API, these specifications are placed in a separate file that gathers all specifications for classes and methods of the Java Card API. 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 be correct; errors in their annotations threaten the soundness of the verification process. Therefore, extra care is taken so that the specification of library functions fully corresponds with the documentation of said functions.

In the context of Java Card, instance predicates are used to describe consistency conditions for applets. Every applet has an invariant that must be preserved by each transaction. This invariant guarantees that the applet is in a consistent state.

## 2.3. The VeriFast symbolic debugger

A feature that proved to be crucial in understanding failed verification attempts is VeriFast's symbolic debugger. As shown in Fig. 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.

The symbolic debugger consists of a number of smaller windows that help the programmer determine where the verification failed. The main sub-window contains the source code of the program that is being verified. VeriFast indicates where the verification of the application failed by selecting and coloring the relevant parts of the source code. All the steps that were required to reach the result are shown in the *Steps* window. The programmer can use this window to select prior verification steps, and thus step back into the symbolic history of the verification algorithm. The *assumptions* window lists assumptions that hold at the selected verification step. The *heap chunks* window offers a view on the symbolic heap; it contains all the elements that are allocated in symbolic memory at the selected verification step. Finally, there is the *locals* window that shows all the local variables and their symbolic values.

## 3. 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

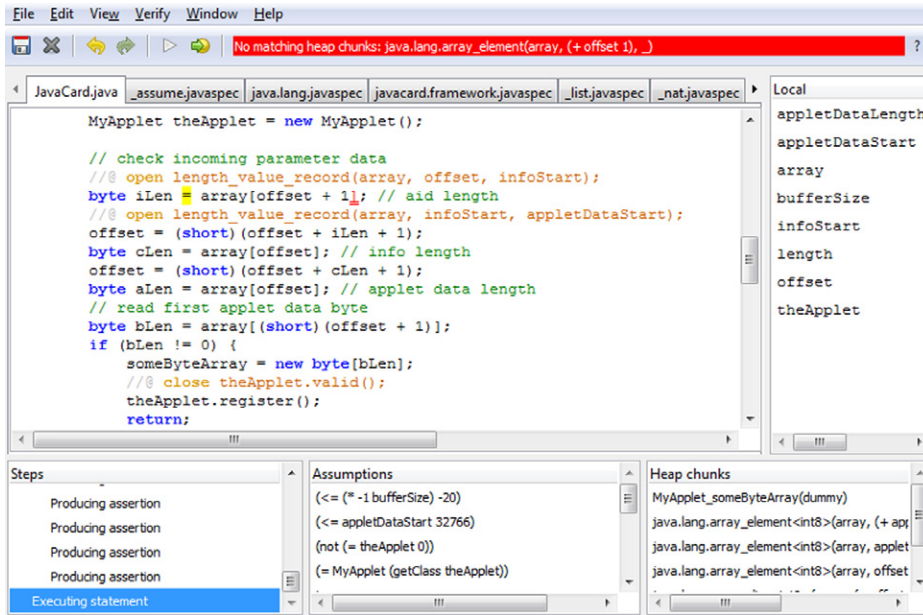


Fig. 1. The symbolic debugger of VeriFast.

corresponding X509 certificates. One key pair is used for authentication, whereas the other key pair can be used to generate legally binding electronic signatures [9].

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 official open source testbed version of the eID applet that implements the same functionality as the real eID card.<sup>3</sup> 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<sup>4</sup> and no less than 38 fields.

### 3.1. Specification of transaction correctness

Java Card offers transactions to preserve consistency of the data stored in persistent memory. In VeriFast, developers can explicitly write down the desired consistency conditions. More specifically, the class Applet defines an instance predicate called `valid`. Each subclass must override this predicate; `o.valid()` means that `valid` must hold as defined in the dynamic type of `o`. The implementation of the predicate given in the subclass defines the consistency conditions for the applet subclass at hand. For example, consider the applet class `ExampleApplet` shown below. The predicate `valid` binds the values of fields `arr` and `i` to logical variables of the same name (line 6), and specifies that `arr` points to a non-null array (line 7). Moreover, the predicate imposes the consistency condition that `i` is a valid index in `arr` (line 8).

Listing 7. The contract of the process method, using fractional permissions.

```

1 class ExampleApplet extends Applet {
2   short i;
3   short[] arr;
4   /*@
5   predicate valid() =
6     this.arr /-> ?arr &*& this.i /-> ?i &*&
7     array_slice(arr, 0, arr.length, _) &*&
8     0 <= i &*& i < arr.length;
9   @*/
10 }
```

<sup>3</sup> The source code of the eID applet can be downloaded from <http://code.google.com/p/eid-quick-key-toolset/>. An annotated version of this source code is included in the examples section of the VeriFast distribution.

<sup>4</sup> Throughout this article we report lines of code (LOC) as physical lines of code excluding comments and blank lines.

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 8, 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 indicating the currently active applet.

**Listing 8.** The contract of the process method, using fractional permissions.

```

1  public void process(...)
2      //@ requires current_applet(this) @*@ [1/2]valid() @*@ ...;
3      //@ ensures current_applet(this) @*@ [1/2]valid() @*@ ...;
4  {
5      ...
6  }
```

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` shown in Listing 9 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 `commitTransaction` 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 9.** The declaration of the `beginTransaction` and `commitTransaction` methods.

```

1  public static void beginTransaction();
2      //@ requires current_applet(?a) @*@ ...;
3      //@ ensures current_applet(a) @*@ [1/2]a.valid() @*@ ...;
4
5  public static void commitTransaction();
6      //@ requires current_applet(?a) @*@ a.valid() @*@ ...;
7      //@ ensures current_applet(a) @*@ [1/2]a.valid() @*@ ...;
```

### 3.2. 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** serve as 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 10 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 10.** A first definition of the `File` class and predicate.

```

1  public abstract class File {
2      //@ predicate File(short theFileID, boolean activeState) =
3          this.fileID -> theFileID @*@
4          this.active -> activeState; @*/
5
6      private short fileID;
7      protected boolean active;
8
9      ...
10 }
```

The `ElementaryFile` class redefines the `File` predicate as shown in lines 2–4 of Listing 11. A `File` predicate that is associated with an `ElementaryFile` class is defined as an `ElementaryFile` predicate where three of the five parameters are unspecified.

The definition of the `ElementaryFile` predicate (lines 5–13) consists of a link to the `File` predicate defined in Listing 10 and some extra fields and information that are specific to elementary files.

**Listing 11.** A first definition of the *ElementaryFile* class and predicate.

```

1  public final class ElementaryFile extends File {
2      /*@ predicate File(short theFileID, boolean activeState) =
3          ElementaryFile(theFileID, ?dedFile, ?dta,
4              activeState, ?sz); @*/
5      /*@ predicate ElementaryFile(short fileID,
6          DedicatedFile parentFile, byte[] data,
7              boolean activeState, short size) =
8          this.File(File.class)(fileID, activeState) &* &
9          this.parentFile |-> parentFile &* &
10         this.data |-> data &* & data != null &* &
11         this.size |-> size &* &
12         array_slice(data, 0, data.length, _) &* &
13         size >= 0 &* & size <= data.length; @*/
14
15     private DedicatedFile parentFile;
16     private byte[] data;
17     private short size;
18
19     ...
20 }

```

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 12 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.

**Listing 12.** Functions to cast predicates.

```

1  public void castFileToElementary()
2      /*@ requires [?f]File(?fid, ?state);
3      /*@ ensures [f]ElementaryFile(fid, _, _, state, _);
4  {
5      /*@ open [f]File(fid, state);
6  }
7
8  public void castElementaryToFile()
9      /*@ requires [?f]ElementaryFile(?fid, ?dedFile, ?dta, ?state, ?sz);
10     /*@ ensures [f]File(fid, state);
11  {
12     /*@ close [f]File(fid, state);
13 }

```

One problem that occurs with the methods presented in Listing 12 is that information is lost when an `ElementaryFile` is cast to a `File` and then back again to an `ElementaryFile` instance. 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 13. 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.

### 3.3. 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 13.** A more complete definition of the *File* and *ElementaryFile* predicates that supports downcasting.

```

1  public abstract class File {
2      /*@ predicate File(short theFileID, boolean activeState,
3          any info) =
4          this.fileID /-> theFileID &*%
5          this.active /-> activeState &*% info == unit; @*/
6
7      ...
8  }
9
10 public final class ElementaryFile extends File {
11     /*@ predicate File(short theFileID, boolean activeState,
12         quad<DedicatedFile, byte[], short, any> info) =
13         ElementaryFile(theFileID, ?dedFile, ?dta, activeState,
14         ?sz, ?ifo) &*% info == quad(dedFile, dta, sz, ifo); @*/
15     /*@ predicate ElementaryFile(short fileID,
16         DedicatedFile parent, byte[] data, boolean activeState,
17         short size, any info) =
18         this.File(File.class)(fileID, activeState, _) &*%
19         this.parentFile /-> parent &*% this.data /-> data &*%
20         data != null &*% this.size /-> size &*%
21         array_slice(data, 0, data.length, _) &*%
22         size >= 0 &*% size <= data.length &*% info == unit; @*/
23
24     ...
25 }

```

*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 this Java Card applet, 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.

The eID applet and helper classes consist of 1004 lines of Java Card code. In order to verify the project, we added 684 lines of VeriFast annotations (or about seven lines of annotations for every ten lines of code). The majority of these annotations were *requires/ensures* pairs (88 pairs, one for each method). 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.

During the past months, a lot of progress has been made to reduce the annotation overhead by automatically inferring open and close statements. This progress can be clearly seen when we compare the first annotated version of the eID applet with the latest version. In the first version, presented in [10], the verification needed 99 open and 112 close statements. With the latest version of VeriFast, the number of required statements has been reduced to 26 and 17 respectively.

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.

*Bugs and other problems in the applet.* 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.

#### 4. Embedded Linux network management software

The second case study presented in this article is on applying VeriFast to an implementation of a Policy Enforcement Point (PEP) for Network Admission Control scenarios. The case study originates from an industry partner of the SECURECHANGE project. Due to a non-disclosure agreement with that partner, the case study, in particular the source code of PEP, cannot be revealed in full.

The PEP program consists of a total of 1194 lines of C code, including comments. It is designed to run on embedded Linux-based gateways and facilitates the application of security policies in Network Admission Control scenarios. More specifically,

for an authenticated network device, PEP will receive an access policy from a Policy Decision Point. This policy is then put in place by configuring the gateway's network interfaces and firewall rules accordingly.

*Case study goals and verification properties.* The PEP implementation is split into 9 C source files and 8 C header files. In total, 53 functions are implemented. The core module of PEP is the file `pep.c`, which comprises 429 lines of code in 13 functions. Although PEP itself is relatively small, it involves a range of Linux libraries – namely *libpcap*, *libdumbnet*, *libssl*, and the POSIX threads API – that increase the complexity of the verification effort substantially.

For a thorough verification that proves the absence of runtime errors and functional correctness, the entire PEP code, including the Linux system libraries would have to be annotated. While, having such a verified software stack would certainly be a great contribution, doing so is beyond the scope of SECURECHANGE. Thus, we restrict this case study to annotating a subset of the above libraries' APIs and the PEP program's core component, `pep.c`.

In general, one would expect that post-hoc verification deals with a program as-is and reports it to be either safe or pinpoints a number of defects – which is exactly what VeriFast aims to do. Yet, since VeriFast is still a prototype its support for the C language is incomplete, which forced us to modify the program under verification in a few places. An important goal of the exercise was to conduct the verification with as few of such modifications as possible so as to efficiently communicate bugs reported by our verification team to the developers of the PEP program. Therefore VeriFast's support for C was extended substantially in the course of this project.

With respect to verification properties, we aimed at proving that the PEP implementation does not perform illegal operations such as dividing by zero or illegal memory accesses. PEP also exploits concurrency, using the POSIX threading API. Thus, a second objective was to verify that the PEP implementation is free of data races. Considering that PEP implements essential security functionality on a gateway by enforcing policies that protect network resources from unauthorized access, violations of the above safety properties do have security implications. Examples for such cases are exploitable buffer overflows (i.e., illegal memory access) or thread synchronization errors that may render the PEP program unresponsive. The consequences of such bugs may be severe: the gateway could be incapable of updating security policies or, even worse, set up policies forged by an attacker.

#### 4.1. Specification of library APIs

Initially alarmed by PEP's use of a number of Linux libraries for performing network access, encryption and thread handling, we were interested in quantifying the effective numbers of functions and type definitions from these libraries, that had to be considered in the verification process. To extract a small but sufficient core of required system headers, we generated pre-processed versions of PEP's source files (using the gcc C compiler's `-E` option) and then iteratively eliminated definitions that were not required for compiling PEP. The resulting definitions were accumulated in a single header file `sys_includes.h` that contains a 99-lines excerpt of the libraries' APIs and the Linux system header files. After all, this file would contain only 20 function prototypes and 19 struct and type definitions. To provide contracts for these functions, we implemented annotations that reflect the functions' documented behavior. More specifically, we consulted the libraries' man-pages and online documentation to develop a total of 162 lines of annotations that sufficiently describe the pre- and postconditions of each function.

*Wrapper functions.* For a small number of functions, such as `pthread_create()` from the POSIX threads API, or `sscanf()` from the standard C library, we decided to implement wrapper functions so as to facilitate easier annotations. This is because VeriFast's support for the C language is still incomplete. In particular, it does not support functions with a variable number of arguments and function pointer declarations. To give examples for this, we compare and contrast the prototypes of `pthread_spin_lock()` and `pthread_create()` as given in `pthread.h` on a recent Linux system with the equivalent headers we annotated for verifying the PEP program.

```

1  typedef unsigned long pthread_t;
2  typedef volatile int pthread_spinlock_t;
3
4  int pthread_spin_lock(pthread_spinlock_t *lock);
5
6  int pthread_create(pthread_t *thread, const pthread_attr_t
7    *attr, void *(*start_routine) (void *), void *arg);

```

The above listing presents an excerpt of the file `pthread.h` taken from an up-to-date Linux installation. The full documentation of the two functions can be found in the POSIX standard [11].<sup>5</sup> Essentially, `pthread_create(thread, attr, run, arg)` creates a new thread which is executing `run(arg)`. The function `pthread_spin_lock(lock)` is used in thread synchronization to lock a spin lock referenced by `lock`. As can be seen, the types `pthread_t` and `pthread_spinlock_t` both default to integers. Variables of these types may be used to reference more elaborate data structures in the internal implementation of POSIX threads, which is not exposed to the programmer.

In the following listing we present the annotated version of these function prototypes. As can be seen, we annotated `pthread_spin_lock` directly but renamed `pthread_create()` by prefixing `vf_`. The reason is that the type definition

<sup>5</sup> Header files and interface documentation of IEEE Std 1003.1-2008 are available online at [pubs.opengroup.org/onlinepubs/9699919799/](http://pubs.opengroup.org/onlinepubs/9699919799/).

for the parameter `start_routine` in the original header file, a function pointer definition, does not parse in the current version of VeriFast. In order to quickly achieve verification results, we annotated a function that uses a parameter `void *run` as the function pointer.

```

1  int pthread_spin_lock(pthread_spinlock_t *lock);
2  /*@ requires [?f]pthread_spinlock(lock, ?lockId, ?p)
3      @*@ lockset(currentThread, ?locks)
4      @*@ lock_below_top_x(lockId, locks) == true; @*/
5  /*@ ensures pthread_spinlock_locked(lock, lockId, p,
6      currentThread, f) @*@ p()
7      @*@ lockset(currentThread, cons(lockId, locks)); @*/
8
9  int vf_pthread_create(pthread_t *pthread, void *attr,
10 void *run, void *arg);
11 /*@ requires is_thread_run_joinable(run) == true
12     @*@ thread_run_pre(run)(arg, ?info)
13     @*@ u_integer(pthread, _); @*/
14 /*@ ensures pthread_thread(?pthread_id, run, arg, info)
15     @*@ u_integer(pthread, pthread_id) @*/

```

Our function contract for `pthread_create()` guarantees that `run` points to a function of the right type, which is captured by the `is_thread_run_joinable(run)` annotation. The predicate `thread_run_pre(run)(...)` encapsulates the preconditions of the function `run` is pointing to, and `pthread` is the integer-type thread ID assigned to each new thread by the `pthread` library to identify the thread. Our annotations of `pthread_create()` employ a ghost predicate `pthread_thread` for the same purpose. Yet, `pthread_thread` may encapsulate further information, such as a thread's arguments, for verification purposes.

The annotations for `pthread_spin_lock()` require permission to some fraction of a `pthread_spinlock` predicate and a `lockset` are passed as arguments, such that the particular lock identified by the information in `pthread_spinlock` is not an element of the `lockset`. The function ensures that a predicate chunk `pthread_spinlock_locked` is produced and the lock is added to the `lockset` for a symbolic execution in which no other thread is holding lock. The data objects protected by `lock` are encapsulated in the predicate `p`, which can be opened after `pthread_spin_lock()` terminated successfully.

To facilitate compilation and to actually run the annotated version of PEP, which is essential to validate bug reports and to communicate confirmed errors to the developers, we have to provide an implementation of `vf_pthread_create()`. As we show in the listing below, the wrapper for `pthread_create()` is trivial to implement since it only avoids prototype definitions that are incompatible from the compiler's point of view. That is, we use implicit casts to a generic `void` pointer and back to the function pointer type. Our method contracts guarantee the parameter `run` to be a valid function pointer. In the same way, implementations of all wrappers we employ is straightforward – they could be removed by investing additional effort in extending VeriFast and further annotating the system APIs.

```

1  int vf_pthread_create(pthread_t *thread, void *attr,
2      void *run, void *arg)
3  { return(pthread_create(thread, attr, run, arg)); }

```

#### 4.2. Verification of PEP

The verification effort on the PEP program consumed a total of about five man-months starting from November 2010. An initial assessment of the feasibility of applying VeriFast to PEP was carried out, concluding that verifying PEP is viable. Yet, VeriFast's support for C needed to be extended substantially to support C language constructs that were not available in VeriFast back then. Work on improving VeriFast and conducting extended case studies on PEP consumed roughly three man-months. In particular, VeriFast now supports C arrays, global structs and nested structs, together with definitions of a number of default types. The time dedicated exclusively to developing the final annotations of the PEP sources is at the scale of two man-months. This includes time spent to modify the PEP program so as to fix bugs by, e.g. adding null-pointer checks and thread synchronization.

After specifying contracts for the Linux system APIs, PEP's core file `pep.c` and internal header files were annotated and verified with respect to our safety properties. Since VeriFast requires all functions declared in the code base to have a pre- and post condition, we started with annotating these functions with stub contracts such as the following:

```

1  void sendEAPoL(char *mac, char *eap, int len)
2  /*@ requires true; @*/
3  /*@ ensures true; @*/
4  { ...

```

*Custom preprocessing.* The resulting source file would not parse correctly in VeriFast since a number of features of the C compiler and preprocessor are still not supported by VeriFast's parser. In particular, this involves conditional compilation

and macro expansion<sup>6</sup> and some uses of the `sizeof` operation. To work around these issues, the build scripts of the PEP program were extended so as to perform “custom preprocessing” by means of search-and-replace based on regular expressions. Importantly, our verification process attempts to compile the PEP program before invoking VeriFast. This is to make sure that we do only verify a program source file that is still API compatible, with respect to the C compiler’s weak type-checking, with the original. Ideally one would also run a set of test cases to gain confidence that the program’s semantics did not change. Yet, such test cases could not be obtained from the stake-holder of the case study.

**Basic Function Contracts.** In a next step, the stub contracts of the functions would be completed gradually to match the actual requirements of the implementation. For the above function `sendEAPoL()`, for example, the precondition has to mention the function’s parameters and any global variables that may be accessed:

```

1 void sendEAPoL(char *mac, char *eap, int len)
2 /*@ requires [?f1]array<char>(mac, ?mac_len, sizeof(char),
3     character, ?mac_chars)
4     @*@ mac_len >= 6
5     @*@ [?f3]array<char>(eap, ?eap_len, sizeof(char),
6     character, ?eap_chars)
7     @*@ eap_len == len @*@ 1 <= len @*@ len <= 1024
8     @*@ [?f4]pointer(eth, ?eth_handle)
9     @*@ [?f2]array<char>(brmac, 6, sizeof(char),
10    character, ?brmac_chars); @*/
11 /*@ ensures true; @*/
12 { ...

```

The first lines of the contract specify that `sendEAPoL()` requires access to a fraction `f1` of a byte array named `mac`, which stores `mac_len` bytes, where the assertion `mac_len >= 6` holds. Similarly, the following lines specify the relation between the byte array `eap` and the parameter `len`. The last two lines of the `requires` state that the function also needs access to two global variables, namely `eth` and `brmac`.

Running VeriFast on a function annotated in this way will, given that the function’s precondition is complete, most probably result in an error report stating that `sendEAPoL()` leaks heap chunks. This is because our postcondition does not specify anything about the input chunks of the function. Thus, VeriFast will assume that the function is meant to destroy these chunks, which is not the case here. In the example above we can simply mention all chunks from the precondition as `sendEAPoL()` is not modifying any properties or assignments:

```

1 ... @*/
2 /*@ ensures [f1]array<char>(mac, mac_len, sizeof(char),
3     character, mac_chars)
4     @*@ [f3]array<char>(eap, eap_len, sizeof(char),
5     character, eap_chars)
6     @*@ [f4]pointer(eth, eth_handle)
7     @*@ [f2]array<char>(brmac, 6, sizeof(char),
8     character, brmac_chars); @*/
9 { char *buf;
10   char *eapol;
11   int leapol;
12   ...
13   encode((char*)&eapol, &leapol, eap);
14   buf=malloc(leapol+14);
15   ...
16   eth_send(eth, buf, leapol+14);
17 }

```

Our function `sendEAPoL()`, however, will still not verify. As can be seen in the above listing, `sendEAPoL()` invokes `encode()`, which is part of PEP, as well as `malloc()` and `eth_send()`, which are part of Linux’ C library and `libdumbnet`, respectively. The contract of `encode()` specifies that this function will assign to the first parameter (`eapol`) either `NULL` or a freshly allocated buffer that holds a specific encoding of the data pointed to by the third parameter. `malloc()` typically allocates a fresh buffer or returns `NULL`, and `eth_send()` requires a pointer to a struct describing an Ethernet device `eth` and a buffer `buf` of a length specified in the third parameter.

Obviously, if `malloc()` fails, the precondition of `eth_send()` is not met. Furthermore, if `encode()` and `malloc()` are successful there will be two fresh heap chunks that are not mentioned in the postcondition of `sendEAPoL()`. Thus, VeriFast will report further memory leaks. Only after we have fixed `sendEAPoL()` by checking the result of `malloc()` before calling `eth_send()`, and by deallocating the memory chunks pointed to by `buf` and `eapol`, the function will verify correctly.

To verify PEP’s core module `pep.c` all functions were annotated in this way. In addition to simple pre- and postconditions, most functions will require loop invariants<sup>7</sup> and additional annotations that make transformations on VeriFast’s symbolic

<sup>6</sup> Preprocessor support has been added to VeriFast after work on this final part of the PEP case study started.

<sup>7</sup> A loop invariants in VeriFast is essentially a contract for a loop body; c.f. the VeriFast tutorial mentioned in Section 2.1.

heap explicit by, e.g. opening and closing predicates. Annotating the program in this manner, initially leaving concurrency aside, we wrote a total of approximately 450 lines of annotations and discovered 28 bugs, similarly to the two memory leaks and the null pointer dereference we identified in `sendEAPoL()` above.

*Adding concurrency.* Concurrency was only considered in the last step of the verification effort. That is, initially we assumed calls to `pthread_create()`, a function from POSIX threads library which is used for creating a new thread, to be equivalent to invoking the new thread's start routine in a sequential context. This allowed us to prepare initial method contracts for these start routines without having to worry about sharing fractional permissions for heap chunks or locking resources. However, adding fractions and deciding which global variables actually need to be protected by locks was everything but trivial. Of course, the familiarity with the code base gained in the previous phases of the cases study helped a lot to revise the body of code and annotations.

In the course of specifying the concurrency related behavior of the PEP program, we discovered another 13 bugs. Fixing these and finally verifying the program correct was difficult, as this involved locking and unlocking resources in several places.

VeriFast's symbolic debugger turned out to be particularly useful in the step of adding concurrency to the contracts. In general, most errors in interactive verification with VeriFast manifest as a missing data object in VeriFast's symbolic memory. That is, the "chunk" in question may be absent altogether or it might not have the expected properties, e.g. content, size or permissions. With the debuggers ability to stepwise symbolic execution and backtracking, it becomes easy to find out where in program execution a particular chunk would be consumed or where its properties change. This way we discovered a number of program locations where or initial non-concurrent annotations over-approximate the program's behavior. A typical case of this would be two functions that read a particular data object but are given full (write) permission to that object. Of course, when concurrently running these two functions, only one of them may actually get full permission to the data object and verification will fail. While this error in the specification is easy to resolve, it shows the pattern by which data races are detected: the two functions would attempt to write to the same data object, and thus need full permission.

Details on the bugs we discovered and on the total amounts of annotations and modifications we applied to PEP's source code are given as verification results in the following section.

### 4.3. Evaluation

The key result of the verification exercise reported on in this section certainly is that we have proven a relatively complex example for an embedded network management software correct with respect to a number of safety properties, which substantially increases our confidence in the overall safety and security of this software component.

Since the PEP program is production code that implements a security module which has potentially been deployed with thousands of home gateways, our expectations to find critical bugs were relatively low.

*Errors Found and Error Severity.* Surprisingly, we discovered a total of 41 bugs in code. In more detail, our verification effort revealed 16 program locations at which a null-pointer may be de-referenced, 6 memory leaks, 6 out-of-bounds access on buffers, 6 global variables that are involved in race conditions, 4 cases in which errors in input/output operations are not handled and 3 unconfirmed functional errors.

The high number of null-pointer errors is largely due to the fact that, throughout the code, the return value of calls to `malloc()` is typically not checked to be unequal to NULL. Arguably, the programmer's assumption is that the PEP program allocates so few memory that `malloc()` will never fail. Yet, the program allocates new buffers regularly throughout its execution, which, in combination with the memory leaks we discovered, will inevitably lead to memory exhaustion. Especially since the program is intended to run on a low-cost embedded device with rather limited resources, we consider the null-pointer errors and memory leaks as a serious problem with respect to the reliability of the PEP program.

The next group of severe bugs are those we report as buffer overflows. Out of the six buffer overflows identified during verification, five are related to reading a malformed configuration file. From a security perspective, this could be abused if an attacker gains access to the configuration file. Evaluating the chances for this is beyond the scope of our work.

More severe is the 6th out-of-bounds access that we detected. VeriFast identified a buffer overread related to parsing a network package received by the PEP program. That is, the PEP program employs the function `pcap_next()` provided by *libpcap* to read network packets from an interface. `pcap_next()` returns a pointer to a char buffer containing the raw packet data and assigns a `struct pcap_pkthdr` with status information of the buffer, including the size of the packet buffer. The PEP implementation only checks whether this packet buffer is at least 14 bytes long. Yet, under some conditions up to 18 bytes of the buffer are read. The content of the overread 4 bytes is used to specify the length and assignment of another buffer, which gives rise to an overflow. We did not investigate whether this bug may be directly abused to manipulate PEP's control flow so as to gain access or to crash PEP.

Furthermore, we discovered a number of race conditions on global variables. PEP spawns a number of threads that listen for particular packets on the several network interfaces the program manages. Yet, all these threads' operations share some global variables to store data, such as MAC addresses, host names, interface names, and file descriptors, that are assigned by one thread and used by the others. We discovered that six of these globals were not protected by appropriate mechanisms to prevent undefined program behavior due to data races. However, the developers of PEP put some `sleep()` statements in place to "ensure" that one thread receives sufficient CPU time to perform a set of critical operations before the next thread

is started. Obviously, the developers were aware of synchronization issues but decided, for unknown reasons, against the use of locks or mutexes.

*Effort and Overhead.* To prove the PEP program correct, we attempted to fix all the above errors. That is, we introduced a number of NULL-checks, `free()` statements, and locks into the program and re-verified it to be correct. Importantly, our fixes make the program safe with respect to our API specifications. Yet, it may still not be functionally correct. That is, e.g., adding and initializing a global lock and `pthread_spin_lock()` and `pthread_spin_unlock()` directives around each access to the particular data object the lock is intended to protect, prevents that data object from being read from and written to at the same time. This sort of locking does, however, not enforce the intended order of execution. In a similar manner, failure of `malloc()` is handled by safely terminating the program, which is most probably not what a security engineer's advice would be: terminating the PEP program might result in the gateway being unable to grant or revoke privileges on demand. Thus, the decision on how to finally address the programming errors discovered in our verification effort is left to the developers and security engineers of our industry partner.

Initially, `pep.c` consisted of 429 non-empty lines of code. The file was modified by removing a number of `printf()` statements which were only executed when the program runs in a “debug mode” but not in actual deployment scenarios. We further added 70 LOC to work around issues in VeriFast. Another 91 LOC were added to fix bugs in the code, resulting in `pep.c` containing 508 non-empty LOC. Finally, verifying `pep.c` required 801 lines of annotations in that file, and another 215 lines of annotations for internal header files. A relatively high effort of approximately two man-months was spent to verify the program, producing 1.57 lines of annotations per line of source code for PEP's main source file. This annotation overhead is roughly twice as high as overheads reported in the previous case study (c.f. Section 3), which is due to the involvement of thread management and resource locking in PEP. The fully annotated version of `pep.c` verifies in VeriFast in just under 20s on an 800 MHz AMD Turion machine running Linux. The peak memory consumption of VeriFast is 31 MB.

## 5. Other case studies with VeriFast

Other nontrivial case studies have also been performed with VeriFast. This section describes our experience with two additional case studies: a Linux driver and an industrial Java Card applet.

### 5.1. The linux USB BP keyboard driver

Kernel and driver code are particularly challenging types of software to verify. They contain a lot of low-level code to interact with hardware, and typically also have strict synchronization and security requirements. However, because they are of paramount importance to an operating system and because they are typically self-contained modules, they are excellent targets for software verification.

There is not much published work that shows whether or not verification of real-world kernel code is feasible. To work towards addressing this question, we applied VeriFast on a device driver taken from the Linux kernel. The driver code subject to verification is Linux's USB Boot Protocol keyboard driver (`usbkbd`). While being small, this driver contains a bigger than expected subset of kernel driver complexity. It involves asynchronous callbacks, dynamic allocated memory, synchronization and usage of complex APIs. During verification, we identified and fixed a number of bugs. For these bugs we submitted patches that have been accepted by the driver's maintainer and included in Linux 3.3.

Verification of the driver is against the original API. Wrapper functions are only used in a few cases where API functions return a struct (i.e. not a pointer to a struct) because this is currently not supported by VeriFast. The APIs that `usbkbd` uses are the USB API, the input API, spinlocks, and some generic functions like `memcpy`. Verification thus consists of (1) writing formal specifications for these APIs, based on official documentation and reading the API implementation for the underspecified or undocumented parts, and (2) of adding annotations to `usbkbd.c`. These annotations consist of contracts (pre- and postconditions written in separation logic), predicates to describe data structures, predicate family instances to instantiate callback function contracts, lemmas (i.e. ghost functions), and ghost-code like folding and unfolding predicates.

The verified properties are absence of data races in the presence of concurrent callbacks, absence of illegal memory accesses, and correct API usage. This does not include a formal proof of correctness of the hand-written API formalization.

`usbkbd` is one of the smallest Linux kernel drivers. It consists of 329 non-empty lines of C code. Verifying the driver required 822 lines of annotations in the driver's code base, i.e. about 2.5 lines of annotations per line of source code. The API specifications count up to 769 lines of code. On an Intel L9400 1.86GHz running the verifier takes about one second. Writing annotations, studying the API documentation, studying the API implementation for undocumented parts and studying the driver implementation is estimated to sum up to about 56 working days. More details about this work can be found in Penninckx et al. [12]. The annotated sources of `usbkbd`, specifications for the used APIs and the patches submitted to the driver's maintainer are available at <http://people.cs.kuleuven.be/~willem.penninckx/usbkbd/>.

### 5.2. An industrial Java Card applet

VeriFast was also used to verify the code of a Java Card applet that was developed and supplied by a commercial smart card vendor in the context of the SECURECHANGE project.

The 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. It was annotated by a VeriFast specialist and took about 5 man-days, excluding the time it took to add some new required features to VeriFast.

We found a number of bugs in the commercial applet, even though it had already been verified with another verification technology [13] 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.

## 6. Related work

In this section we give pointers to related verification tools that employ separation logic and discuss software verification tools and case studies that are relevant in the context of verifying Java Smart Card applets, low-level system management software and operating system components. The reader is referred to [2,14] more background and a discussion of the related work on VeriFast.

### 6.1. Verification using separation logic

Smallfoot [4] is a verification tool based on separation logic which given pre- and post-conditions and loop invariants can fully automatically perform shape analysis. It has been extended for greater automation [15], for termination proofs [16,17], fine-grained concurrency [18] and lock-based concurrency [19].

jStar [20] is another automatic separation logic based verification tool which targets Java. One only needs to provide the pre- and post-conditions for each method, after which it attempts to verify it without extra help. It is able to infer loop invariants, for which it uses a more generalized version of the approach described by Distefano et al. [21]. This is achieved by allowing the definition of user-defined rules (comparable to lemmas in VeriFast) which are then used by the tool to perform abstraction on the heap state during the fixed point computation.

A third verification tool based on shape analysis and separation logic is SpaceInvader [21,15], which performs shape analysis on C programs. Shape analysis aims at automatically inferring, e.g., whether a variable points to a cyclic or acyclic list. Shape analysis can be employed to verify pointer safety, guaranteeing that the shape of data structures is maintained throughout program execution. It has been applied to Windows [22] and Linux [15] drivers. Abductor [23], an extension of SpaceInvader, uses a generalized form of abduction, which gives it the ability not only to infer loop invariants and postconditions, but also preconditions. Notably, Abductor has been applied in large-scale case studies on open source programs, showing that bi-abduction is capable of automatically synthesizing specifications for a majority of data structures used in these programs [24]. We are investigating the integration of shape analysis techniques into VeriFast with the goal of reducing the annotation burden [25].

### 6.2. Other verification techniques for Java Card programs

The Extended Static Checker for Java (ESC/Java) [26] is another program verifier that has been used to verify Java Card programs [27,28]. However, Esc/Java is unsound (c.f. Appendix C.0 of [29]). 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 [30].

Gomes et al. [31] 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.

To avoid having to write annotations, Huisman et al. [32] 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 [33] 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 [34] verifier. The KeY tool uses semiautomatic deductive verification of Java code annotated with JML contracts – as described in the design by contract paradigm – in order to statically prove the code's correctness with regards to its specifications. The approach has been further used to verify an implementation of the Mondex electronic purse system [35]. 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 [33], 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 Java programs. This article 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.

### 6.3. Other verification techniques for C programs

A number of automated tools for verifying C programs have been introduced. Notably, CEGAR-based [36] model checkers such as BLAST [37] and SLAM/SDV [38] have been applied to check the conformance of device drivers with a set of API usage rules. In contrast with our work, these tools do not provide support for identifying errors with respect to the inherently concurrent execution environment device drivers are operating in. Focusing on API usage rules, these tools either assume memory safety [38] or, as shown in [39], perform poorly when checking OS components for memory safety.

In [40] a model checker with support for pointers, bit-vector operations and concurrency is evaluated on a case study on Linux device drivers. The tool checks for buffer overflows, pointer safety, division by zero and user-written assertions. Yet, it requires a test harness with a fixed number of threads to be generated for each driver. VeriFast, in contrast, handles concurrency implicitly, is sound, and implements assume–guarantee reasoning using generic API contracts. Therefore, VeriFast can check each function of a driver in isolation, which contributes to the scalability of our approach.

Bounded model checking and symbolic execution have been successfully applied to the source code [41,42] and to the object code [43] of kernel modules. Unlike the VeriFast approach, these techniques are not generally sound but are effective for finding bugs with little or no human interaction involved: bounded model checking is neither sound nor complete and suffers from severe limitations with respect to reasoning about concurrently executing programs. While VeriFast can efficiently reason about such programs and actually prove the absence of errors, our approach requires extensive human-provided annotations.

A competing toolkit to VeriFast is the Verifying C Compiler (VCC) [44]. VCC verifies C programs annotated with contracts in Boogie. The tool generates verification conditions from the annotated program, which are then discharged by an SMT solver. As a result of this VCC does not ship with a symbolic debugger such as VeriFast's, that can be used to inspect traces of a program under verification. VCC can be expected to require fewer annotations than VeriFast, however, at the expense of a less predictable search time. The VCC toolkit has been employed in a case study on verifying the Microsoft Hypervisor [45].

Other approaches to OS verification involve modeling and interactive proof. Most notably, the L4.verified [46] project aims at producing a verified OS kernel by establishing refinement relations between several layers of Isabelle/HOL specifications, a prototypic kernel implementation in Haskell and the actual kernel implementation in C and assembly. This differs from our work as we do not employ refinement relations and verification is non-interactive. Also, the case studies presented in this article do not focus on proving full functional correctness.

## 7. Conclusion

This article reports on four industrial case studies with the VeriFast program verifier. We present results and give details on source code annotations for an open-source version of a Java Card applet that implements the Belgian electronic ID card, and a commercial C-implementation of a Policy Enforcement Point (PEP) for embedded Linux gateways. We further summarize two case studies, one on a Linux device driver and one on an industrial Java Card applet. All four programs have been verified for correctness with respect to the absence of certain common programming errors. In particular, the verification checked that the applications do not contain transaction errors, synchronization or multi-threading errors, performed no out-of-bounds operations on buffers, and never dereferenced null pointers or leak memory.

In all case studies, a number of bugs were discovered that could have an impact on the stability and security of these systems. Notably, the PEP implementation that had been deployed on numerous home gateways, was found to contain a surprisingly high number of memory safety bugs and race conditions, some of which might have severe implications on the reliability and security of the gateway. Also, the industrial Java Card applet that was already verified with another verification technology was found to still contain a number of bugs, indicating that either the other verification technology was unsound or was used in an unsound way.

*Annotations & annotation overhead.* The results of the case study are encouraging: the annotation overhead differs from case study to case study, but never proved prohibitively large. The annotation overhead varies between 0.69 and 2.5 lines of annotation per line of code, with an average of 1.2 lines of annotations per line of code. There is a huge difference in annotation overhead between our Java case studies and the C case studies with an average of 0.71 versus 1.96 lines of annotations per line of code, which is largely due to the involvement of concurrency in PEP and the USB BP Keyboard driver. All case studies consumed between one and two person-months to conduct. On average, we verified 2.17 LOC per person-hour. As all case studies were carried out by researchers with no or little prior experience with VeriFast, a steep learning curve was experienced. Thus, we believe that the human verification time per line of code dropped considerably towards the end of each project.

Not included in the discussion above is the effort to specify library APIs. Yet, in the case of PEP and the USB BP Keyboard driver these annotations are quite substantial and amount to 449 and 769 lines, respectively. For well-documented APIs, these specifications are fairly easy to develop since pre- and post-conditions can be derived from the library's documentation while the code body of library functions is not to be considered. More importantly, annotating APIs is typically a one-time effort and a succession of verification projects may employ the results. A good example for this is the POSIX threads API that was annotated in the PEP case study and is now part of the VeriFast distribution. Also the Linux networking APIs could be re-used in a successive verification of, e.g., a Policy Decision Point that provides the policies that are to be put in place by the PEP program.

*Absence of run-time errors versus functional correctness.* Throughout this article we focus on proving the absence of run-time errors and data races in our case studies. VeriFast, however, verifies that a function implementation satisfies its specification given in terms of pre- and post-conditions. With this, the tool is in general capable of verifying full functional correctness. For a number of reasons, verifying the case studies for functional correctness was considered to be beyond the scope of the project.

In particular, none of the programs we used as case studies was designed with verification in mind. Specifications of those programs were given in terms of informal prose or had to be reverse-engineered from the implementation. As a result of this, several run-time errors that have been identified in the course of our verification effort could only be fixed in a way such that the program does not exhibit the erroneous behavior any more. It was left to the owners of the case studies to decide whether our fixes conform with the intended behavior of the program.

Furthermore, our annotations are considerable in size already, although only program behavior relevant for checking for the absence of run-time errors is specified. For example, consider a function that returns a freshly allocated string object containing a base64 encoding of some input data. In our approach we may only provide annotations specifying that the function requires an input string of length  $n$  and ensures that this input string is not modified, and that a freshly allocated string of length  $m$  with  $n \leq m$  is returned. Specifying that this function performs the encoding correctly, and that its callers, e.g. PEP's `sendEAPoL` function, produces an RFC 5247 compliant EAP message, etc., was plainly beyond our capacities and not requested by the owners of the case studies. Yet, it would be interesting to experiment with more complete specifications in the future.

*Lessons Learned.* The aim of the case studies described in this article was to show that separation logic-based reasoning, as implemented in VeriFast, can leverage software verification so that the post-hoc verification of critical industrial-scale programs becomes feasible. We presented VeriFast as a general-purpose software verification tool that combines an interactive verification experience with a high degree of automation and features reasoning about programs that involve concurrency and dynamic memory allocation.

As mentioned earlier, all case studies were carried out by researchers with almost no prior experience in using VeriFast. Yet, all investigators do have a background in formal methods and VeriFast's key developers were always available to help out. Thus, a steep learning curve was experienced while conducting the case studies, especially with respect to learning the annotation syntax, understanding why seemingly correct contracts do not verify, working around shortcomings in VeriFast's support for C and Java, and acquiring an intuition in how to efficiently annotate methods. VeriFast ships with an extensive tutorial and a large number of examples and test cases. Those helped a lot to familiarize with the tool. Yet, being a research prototype, both documentation and language support need to be updated and improved. Potentially we should aim at integrating one of the case studies from this article into the tutorial documentation.

Throughout our experiments, VeriFast's symbolic debugger turned out to be an indispensable tool. Being based on symbolic execution, the debugger presents errors in terms of a path in the program under verification such that the program location at which an assertion is violated is clearly displayed. It offers the possibility for stepwise execution and backtracking on that path while monitoring variable assignments, the symbolic heap, and VeriFast's assumptions. It often becomes clear immediately, whether an error report is due to an error in a contract, a missing verification step, or whether there is an actual bug in the code. Furthermore, the symbolic debugger enables efficient communication of error traces to software developers.

Applying VeriFast generally becomes easier in software projects with a well-structured code base. That is, starting to annotate a number of small, separate modules that interact via well-documented APIs yields quick feedback and a feeling of success for the verification engineer. To some degree this has to do with VeriFast's requirement to have all functions in a source file (and in imported interfaces or header files) to be annotated before any analysis is performed. As a result, one typically starts with stub annotations, as explained in Section 4.2, that are then slowly extended to reflect the actual methods' behavior. For example, when verifying PEP's main source file we had to start with annotating all function declarations in that file and in all header files involved in the project with stubs, dealing with VeriFast's – back then – rather incomplete support for the C language immediately with no easy achievements visible. A rather discouraging part of the work.

With respect to handling concurrency, we experienced good results with first annotating the project as if it was meant to execute sequentially. That is, our initial contracts for functions such as `pthread_create(. . . , start_routine, . . . )` assumed that `start_routine` would be invoked directly, i.e. without starting a new thread. This enabled us to develop valid contracts for all methods before considering concurrent interactions in the program, keeping our annotations initially simple and computation times in VeriFast short.

*Final Remarks.* Given the strong guarantees that VeriFast provides in return of the annotation effort, and the empirical evidence of the many bugs we discovered in the four case studies presented in this article, we are coming closer to the point where the approach might in some projects be cost-effective. This is especially true in the domain of security critical and safety critical applications, where bugs may have severe consequences. We hope that the details we present on the verification process, our annotations, and the advantages and pitfalls of using VeriFast will be helpful for future industrial or academic verification projects.

Of course, VeriFast is under active development and its language support improves gradually with each case study. For example, recent work on annotation inference [25] has greatly reduced the amount of annotations that is required by VeriFast, and we strive to further reduce the amount of required annotations in future versions of VeriFast.

## Acknowledgments

We thank the editors and anonymous reviewers of *Science of Computer Programming*, *AVoCS 2011* and *NFM 2012* for their valuable comments on this article and the previously published extended abstracts, respectively. This research is partially funded by IWT, by the Research Fund KU Leuven, and by the EU FP7 projects SecureChange and NESSoS. Our work was carried out with financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CENTRE). Jan Smans is a postdoctoral fellow of the Fund for Scientific Research Flanders (FWO). We acknowledge support from Microsoft Research Cambridge as part of the Verified Software Initiative.

## References

- [1] J. Woodcock, P.G. Larsen, J. Bicarregui, J. Fitzgerald, Formal methods: practice and experience, *ACM Comput. Surv.* 41 (4) (2009) 1–36.
- [2] B. Jacobs, J. Smans, F. Piessens, A quick tour of the VeriFast program verifier, in: *APLAS*, 2010, in: *LNCS*, vol. 6461, Springer, Heidelberg, 2010, pp. 304–311.
- [3] P. O'Hearn, J. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: *CSL*, 2001, in: *LNCS*, vol. 2142, Springer, Heidelberg, 2001, pp. 1–19.
- [4] J. Berdine, C. Calcagno, P. O'Hearn, Symbolic execution with separation logic, in: *APLAS*, 2005, in: *LNCS*, vol. 3780, Springer, Heidelberg, 2005, pp. 52–68.
- [5] R. Bornat, C. Calcagno, P. O'Hearn, M. Parkinson, Permission accounting in separation logic, in: *POPL* 2005, ACM, New York, 2005, pp. 259–270.
- [6] ISO/IEC, 9899:2011, Information technology – Programming languages – C, available at [http://www.iso.org/iso/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=57853](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853), 2011.
- [7] M. Parkinson, Class invariants: the end of the road? in: *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming*, IWACO, 2007.
- [8] Oracle, Java card technology, available at <http://www.oracle.com/technetwork/java/javacard/overview/>, 2011.
- [9] D. De Cock, K. Wouters, B. Preneel, Introduction to the Belgian EID card, in: *Public Key Infrastructure*, in: *LNCS*, vol. 3093, Springer, Heidelberg, 2004, pp. 621–622.
- [10] P. Philippaerts, F. Vogels, J. Smans, B. Jacobs, F. Piessens, The Belgian electronic identity card: a verification case study, in: *AVOCS*, in: *Electronic Communications of the EASST*, vol. 46, 2011, pp. 1–16.
- [11] IEEE standard for information technology – portable operating system interface (POSIX) base specifications, Issue 7, Tech. Rep. 1003.1-2008, IEEE, 2008.
- [12] W. Penninckx, J.T. Mühlberg, J. Smans, B. Jacobs, F. Piessens, Sound formal verification of Linux's USB BP keyboard driver, in: *NFM 2012*, in: *LNCS*, vol. 7226, Springer, Heidelberg, 2012, pp. 210–215.
- [13] J.C. Filliâtre, C. Marché, The Why/Krakatoa/Caduceus platform for deductive program verification, in: *CAV 2007*, in: *LNCS*, vol. 4590, Springer, Heidelberg, 2007, pp. 173–177.
- [14] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, VeriFast: A powerful, sound, predictable, fast verifier for C and Java, in: *NASA Formal Methods 2011*, in: *LNCS*, vol. 6617, Springer, Heidelberg, 2011, pp. 41–55.
- [15] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, Scalable shape analysis for systems code, in: *CAV 2008*, in: *LNCS*, vol. 5123, Springer, Heidelberg, 2008, pp. 385–398.
- [16] J. Brotherston, R. Bornat, C. Calcagno, Cyclic proofs of program termination in separation logic, *SIGPLAN Not.* 43 (2008) 101–112.
- [17] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, M.Y. Vardi, Proving that programs eventually do something good, *SIGPLAN Not.* 42 (2007) 265–276.
- [18] C. Calcagno, M. Parkinson, V. Vafeiadis, Modular safety checking for fine-grained concurrency, in: *SAS '07*, in: *LNCS*, vol. 4634, Springer, Heidelberg, 2007, pp. 233–248.
- [19] A. Gotsman, J. Berdine, B. Cook, N. Rinetzy, M. Sagiv, Local reasoning for storable locks and threads, in: *APLAS '07*, in: *LNCS*, vol. 4807, Springer, Heidelberg, 2007, pp. 19–37.
- [20] D. Distefano, M.J. Parkinson, jStar: Towards practical verification for Java, in: *OOPSLA'08*, ACM, New York, 2008, pp. 213–226.
- [21] D. Distefano, P. O'Hearn, H. Yang, A local shape analysis based on separation logic, in: *TACAS 2006*, in: *LNCS*, vol. 3920, Springer, Heidelberg, 2006, pp. 287–302.
- [22] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, H. Yang, Shape analysis for composite data structures, in: *CAV 2007*, in: *LNCS*, vol. 4590, Springer, Heidelberg, 2007, pp. 178–192.
- [23] C. Calcagno, D. Distefano, P. O'Hearn, H. Yang, Compositional shape analysis by means of bi-abduction, *SIGPLAN Not.* 44 (1) (2009) 289–300.
- [24] D. Distefano, Attacking large industrial code with bi-abductive inference, in: *FMICS 2009*, in: *LNCS*, vol. 5825, Springer, Heidelberg, 2009, pp. 1–8.
- [25] F. Vogels, B. Jacobs, F. Piessens, J. Smans, Annotation inference for separation logic based verifiers, in: *FMOODS 2011*, in: *LNCS*, vol. 6722, Springer, Heidelberg, 2011, pp. 319–333.
- [26] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for Java, in: *PLDI 2002*, ACM, New York, 2002, pp. 234–245.
- [27] W. Mostowski, E. Poll, Midlet navigation graphs in JML, in: *SBMF 2010*, in: *LNCS*, vol. 6527, Springer, Heidelberg, 2010, pp. 17–32.
- [28] N. Cataño, M. Huisman, Formal specification of Gemplus' electronic purse case study using ESC/Java, in: *Formal Methods Europe*, in: *LNCS*, vol. 2391, Springer, Heidelberg, 2002, pp. 272–289.
- [29] K.R.M. Leino, G. Nelson, J.B. Saxe, *ESC/Java user's manual*, Tech. rep., Compaq Systems Research Center, 2000.
- [30] B. Jacobs, F. Piessens, Expressive modular fine-grained concurrency specification, in: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL* 2011, 46 (1) 2011, pp. 271–282.
- [31] B.E.G. Gomes, A.M. Moreira, D. Déharbe, Developing Java card applications with B, in: *SBMF 2005*, in: *ENTCS*, vol. 184, Elsevier, 2005, pp. 81–96.
- [32] M. Huisman, D. Gurov, C. Sprenger, G. Chugunov, Checking absence of illicit applet interactions: a case study, in: *FASE 2004*, in: *LNCS*, vol. 2984, Springer, Heidelberg, 2004, pp. 84–98.
- [33] W. Mostowski, Fully verified Java Card API reference implementation, in: *International Verification Workshop*, *VERIFY*, 2007.
- [34] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P.H. Schmitt, The KeY tool, *Soft. Syst. Model.* 4 (2005) 32–54.
- [35] P.H. Schmitt, I. Tonin, Verifying the Mondex case study, in: *SEFM 2007*, IEEE, Washington, 2007, pp. 47–58.
- [36] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, *J. ACM* 50 (5) (2003) 752–794.
- [37] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, W. Weimer, Temporal-safety proofs for systems code, in: *CAV 2002*, in: *LNCS*, vol. 2402, Springer, Heidelberg, 2002, pp. 382–399.
- [38] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, A. Ustuner, Thorough static analysis of device drivers, *SIGOPS Oper. Syst. Rev.* 40 (4) (2006) 73–85.
- [39] J.T. Mühlberg, G. Lüttgen, BLASTing Linux code, in: *FMICS*, in: *LNCS*, vol. 4346, Springer, Heidelberg, 2007, pp. 211–226.
- [40] T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher, Model checking concurrent Linux device drivers, in: *ASE 2007*, ACM, New York, 2007, pp. 501–504.

- [41] H. Post, C. Sinz, W. Küchlin, Towards automatic software model checking of thousands of Linux modules – a case study with Avinux, *Softw. Test. Verif. Reliab.* 19 (2009) 155–172.
- [42] M. Kim, Y. Kim, Concolic testing of the multi-sector read operation for flash memory file system, in: SBMF 2009, in: LNCS, vol. 5902, Springer, Heidelberg, 2009, pp. 251–265.
- [43] J.T. Mühlberg, G. Lüttgen, Verifying compiled file system code, in: SBMF 2009, in: LNCS, vol. 5902, Springer, Heidelberg, 2009, pp. 306–320.
- [44] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, VCC: A practical system for verifying concurrent C, in: TPHOLS, in: LNCS, vol. 5674, Springer, Heidelberg, 2009, pp. 23–42.
- [45] D. Leinenbach, T. Santen, Verifying the Microsoft Hyper-V hypervisor with VCC, in: FM'09, in: LNCS, vol. 5850, Springer, Heidelberg, 2009, pp. 806–809.
- [46] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, S.M. Petters, Towards trustworthy computing systems: taking microkernels to the next level, *SIGOPS Oper. Syst. Rev.* 41 (2007) 3–11.