Formally Proved Anti-tearing Properties of Embedded C Code

June Andronick
Security Labs – Gemalto
june.andronick@gemalto.com

Abstract— **In smart card embedded programs, some operations must not be suddenly interrupted, because if they are, the card is left in an inconsistent state. Since the card can be removed at any time from the terminal, which interrupts any running program, some instructions must be executed at each reset in order to verify if a tearing occurred and to restore a consistent state if necessary. In this case, the card is said to ensure the anti-tearing property. This paper presents a method to formally prove that a C program verifies the anti-tearing property for a given "tearing-sensitive" operation. The back-ground methodology, presented in [1], [2], enables to prove global properties from source code. It is here illustrated by the proof of anti-tearing properties, which requires an extension of the method in order to specify and verify functions behaviour in the case of a sudden interruption of their execution.**

I. Anti-tearing Properties

Smart cards are small plastic cards with embedded memory and microprocessor which aim at being a secure and safe gate to the digital world. One of the challenges to take up is to ensure a consistent state of the card, even if the card is, maliciously or accidentally, removed from the terminal. Indeed, since a smart card is supplied with power only when inserted into a terminal, the card can suddenly be removed from the terminal, provoking the interruption of the program that was running on the card. Such a *tearing* or *card tear* may leave the card in an inconsistent state, that could generate the crash of the card or that could be maliciously used. For instance, some operations have to be executed *atomically*, i.e. either all instructions are executed or none is. When only a part of the instructions are executed, the card is in an inconsistent state. When some loyalty points are transformed in a credit of money, the system is inconsistent during the time the number of loyalty points has already been decremented and the account has not yet been incremented (or vice-versa). When a PIN – Personal Identification Number – is asked from the holder of a credit card, the system is inconsistent during the time a wrong PIN has already been given and the number of tries has not yet been decremented (giving the opportunity to a malicious holder to try infinitely many PINs).

If an atomic operation is interrupted, the anti-tearing property is ensured by a "roll-back" at reset: all the instructions executed from the beginning of the operation are cancelled. This is usually implemented using a *transaction* mechanism, which consists in considering the instructions as *conditional*, i.e. effective only when all the instructions are executed. The Java Card language (see [3]), specially designed for the programming of smart card applications, offers support for transactions. The correctness of this mechanism has been studied in various related work (e.g. [4], [5], [6], [7]), that will be detailed along this paper. In our work, we are interested in lower levels of the card. In stead of studying embedded application, we target the verification of the underlying operating system, i.e. programs written in the C language.

In this context, atomic operations are not the only operations sensitive to tearing. Operations such as the erasing of a memory block also leave the card in an inconsistent state if they are interrupted. If the card is suddenly removed from the terminal during the erasing of the block, then the reading or use of this inconsistent block may provoke fatal errors. Here the anti-tearing property is ensured if the interruption of the erase operation is detectable and if, at each reset, the erase is performed again if it has been interrupted.

More generally, to each "tearing-sensitive" operation f, is associated an *abort* function f_{ab}, called at each reset, which restores a consistent state if the execution of f has been interrupted. The notion of "consistent state" depends on the operation f. For the erase operation, the card is consistent if the block is erased, whereas for any atomic operation, the card is consistent if there is no transaction ongoing (i.e. if any ongoing transaction has been rolled-back).

Therefore, proving that a given operation f verifies the anti-tearing property means proving that if f_{ab} is called at each reset, then the execution of f always results in a consistent state, even if it has been interrupted. In fact, a tearing may also occur during the abort function. Therefore the card should be in a consistent state when at last the abort function is executed without being interrupted. This property is illustrated in Figure 1, where PO represents the fact that a *power off* occurred.

This paper presents a method to formally prove that the C implementation of a given "tearing-sensitive" operation f ensures the anti-tearing property. As illustrated in Figure 1, this implies to formally model each function as a transition between global states of the card. In [1], [2], we presented a general method to build a formal transition system from a source code written in the C language. On one hand, the transition system is automatically extracted from a formal specification of the code. On the other hand, a program verification tool, called Caduceus (see [8], [9]), is used to

$$\begin{tikzcd}
 & s'_n & & & & & \\
\forall\ s_0.\ \ s_0 \arrow[ur, "f,\ \neg PO"] \arrow[dr, "f,\ PO"'] & & & & & \Rightarrow & \begin{array}{c} is\_consistent(s'_n) \\ \land \\ is\_consistent(s_n) \end{array} \\
 & s_1 \arrow[r, "f_{ab},\ PO"] & s_2 \arrow[r, "f_{ab},\ PO"] & \dots \arrow[r, "f_{ab},\ \neg PO"] & s_n & &
\end{tikzcd}$$

Fig. 1. The anti-tearing property

prove that the source code verifies this formal specification, establishing a formal link between the code and the transition system. This paper shows how this method is useful to prove anti-tearing properties and explain how Caduceus has been extended in order to specify properties in the case on the interruption of the execution of a function.

The next section explains the building of the transition system from the source code, using Caduceus. The tool is briefly described and the method is illustrated by a trivial erase function. In Section III, we describe how the tool has been extended in order to be able to specify and prove a function behaviour in the case of a sudden interruption of its execution. Then, in Section IV, we show how the transition system extraction, together with the modelling of interruptions, are used to formally model anti-tearing properties. Finally the last section concludes the paper.

II. Formal Transition System from C source code

Our goal is to model any C function f as a transition ($f\_transition\ \mathbf{x}\ \mathbf{x}'$) between global memory states $\mathbf{x}$ and $\mathbf{x}'$. Some existing work, namely [7], proposes the definition of these memory states withing a specification language. This work focuses on the Java Card language and explains how to define temporal properties in the JML specification language (see [10]). In our context, where the C language is targeted, an analog approach would be to define the memory states within a specification language for C programs, such as the one of the Caduceus tool.

Our approach is different. We want to take advantage of the work done by a verification tool, like Caduceus, which needs to build a memory model to generate the verification conditions. More precisely, we propose a method (described in detail in [1], [2]) to build a transition system, formally linked to the source code and such that the memory states are computed by the Caduceus tool.

Caduceus.

Caduceus (see [8], [9]) is a verification tool at the source code level. From an annotated C program, it generates so-called *verification conditions*, whose validity implies that the specification is verified by the source code. More precisely, Caduceus is based on an underlying tool called Why (see [11], [12]), whose input language is specifically designed for verification purpose. In other words, Caduceus translates annotated C programs into Why input language (see Figure 2). Then the Why tool is in charge of generating the verification conditions, which may be translated in various theorem provers (including Coq [13]) or decision procedures.

The annotations are used to define the functional properties of each function, in a specification language inspired by the Java Modelling Language (JML, see [10]). They may express functions preconditions (with the keyword `requires`), side-effects (with `assigns`), postconditions (with `ensures`), global invariants, loop invariants, etc. Moreover, in the postcondition, the construction `\result` may be used for the result returned by the function and `\old` for the initial state of the function.

From an annotated C program, Caduceus computes the memory states of the program, according to its intern memory model. Each function and each specification is then expressed in terms of transformation of these states. For instance, a memory variable m will be used to represent the part of the memory where arrays of integers are allocated. Moreover, the notions of pointer and arrays are identified in the memory model, in a type `pointer` . A variable `p` of type `pointer` is either the `null` pointer, or a pair `(base_addr(p),offset(p))` . More precisely, the memory variable m can be seen as a function which associates each base address corresponding to an allocated array of size n to a piece of memory of size n containing the integer values in the array (see Figure 3).

For example, if a program contains the declaration `int block[SIZE]` , a variable m is used to represent the block of memory where `block` is allocated, and an access `block[i]` will be translated by $(acc\ m\ (block[i]))$, where acc is a function enabling to manipulate the Why variables representing memory blocks.

Usually the user only sees the verification conditions. But when the Coq proof-assistant is used as output, the user can also access to the formal model of the functions and the specifications. Indeed, for each function f, a *validation term* T_f of type τ_f is provided, where T_f is the function model and τ_f is the specification model :

$$T_f\ :\ \forall \mathbf{x}.\ Pre_f(\mathbf{x})\ \rightarrow\ \exists \mathbf{x}'.\ Post_f(\mathbf{x},\mathbf{x}')$$

The variables $\mathbf{x}$ and $\mathbf{x}'$ are the global memory states of the program. More precisely, for each function f taking in argument a set $\overrightarrow{a}$ of parameters, Caduceus computes the set

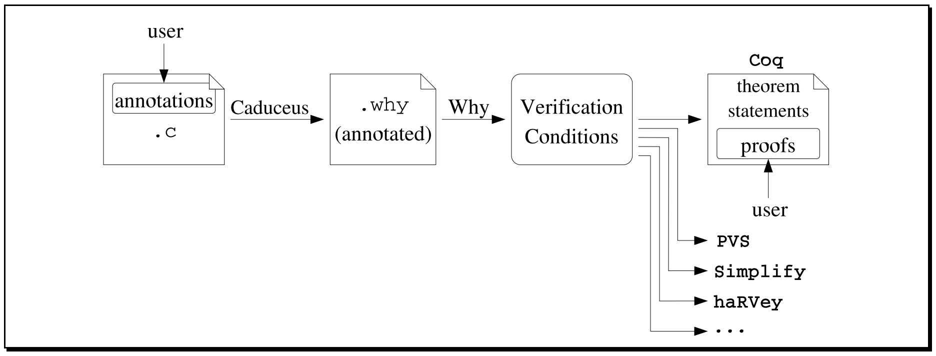

Fig. 2. Caduceus architecture

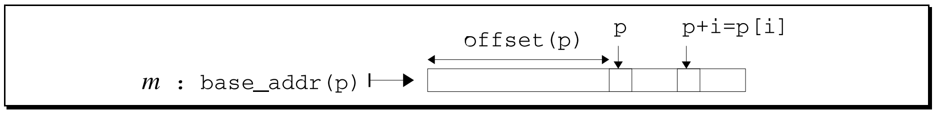

Fig. 3. Caduceus memory model

$\overrightarrow{z}$ of variables (global variables and variables representing memory blocks) that are only read by the function and the set $\overrightarrow{t}$ of the variables read and modified. The validation term has thus the following form:

$$\begin{aligned} T_f \ : \ & \forall \overrightarrow{a}.\ \forall \overrightarrow{z}.\ \forall \overrightarrow{t}. \\ & Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t}) \ \rightarrow \\ & \exists result.\ \exists \overrightarrow{t'}.\ Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t}, \overrightarrow{t'}) \end{aligned}$$

where $\overrightarrow{t'}$ corresponds to the values after the function call and $\overrightarrow{t}$ to the values at function call.

TRANSITION SYSTEM.

The idea is to use the memory states computed by Caduceus to build the transition system. Indeed, all the hard work of computing $\mathbf{x}$ and $\mathbf{x}'$ in the relation $(f\_transition\ \mathbf{x}\ \mathbf{x}')$ can be omitted by using the memory states computed by Caduceus.

There are two ways to define the transition: using the model of the function or using the model of its specification. In other words the transition is defined either by:

$$(f\_transition\ \mathbf{x}\ \mathbf{x}') \quad \equiv \quad Pre_f(\mathbf{x}) \ \wedge \ \mathbf{x}' = \bar{f}(\mathbf{x})$$

where $\bar{f}$ represents the model of the function body, or by:

$$(f\_transition\ \mathbf{x}\ \mathbf{x}') \quad \equiv \quad Pre_f(\mathbf{x}) \ \wedge \ Post_f(\mathbf{x}, \mathbf{x}').$$

In the first case, $\mathbf{x}'$ is *the* resulting state defined by f, such that the postcondition is satisfied. This solution is the most precise, but also the heaviest since the model of the code itself has to be considered.

Whereas in the second case, $\mathbf{x}'$ is *any* value satisfying the postcondition. Therefore, only properties where $\mathbf{x}'$ is universally quantified will be verified by the code. Moreover this solution requires the validity of the verification conditions. Indeed, if a property P is verified for any $\mathbf{x}'$ which satisfies the postcondition, then it is also verified for the one defined by f, *but only if we have proved that the code of f satisfies the postcondition.*

In our method, we will use this latter approach, using only the model of the specification. More precisely, our method consists in:

- specifying each function behaviour using annotations;
- proving that the source code satisfies this specification (i.e. proving the verification condition generated by Caduceus);
- define the transition by
  $(f\_transition\ \mathbf{x}\ \mathbf{x}') \quad \equiv \quad Pre_f(\mathbf{x}) \ \wedge \ Post_f(\mathbf{x}, \mathbf{x}')$
  (which can be expressed by a simple expression depending only on the validation term);
- prove global properties where the resulting state is universally quantified.

EXAMPLE.

Let us consider a trivial function which erases a block of memory. In order to be able to detect if the erasing has been interrupted, a status variable indicates when the erase operation starts and when it is finished:

```
int block[SIZE];

#define  STARTED    0
#define  FINISHED   1
char  erase_state;

void  erase() {
  int  i;
  erase_state=STARTED;
  for  (i=0;i<SIZE;i++)    { block[i]=0;    }
  erase_state=FINISHED;
}
```

In order to ensure the anti-tearing property, the erase function is associated to an abort function, called at reset, which erases again the block if the erase operation has been interrupted:

```
void erase_abort() {
  int i;
  if (erase_state==STARTED) {
    for (i=0;i<SIZE;i++) { block[i]=0; }
    erase_state=FINISHED;
  }
}
```

A simple specification of the erase function is that, at the end of the execution, the block is erased. This is formally stated in the specification language as follows.

```
/*@ ensures
  @   \forall int k; 0<=k<SIZE => block[k]==0 */
```

Concerning the abort function, if the erase state indicates that a tearing occurred, then the block is erased at the end of the function. Besides, if no tearing occurred, the value of the block is unchanged. The formal specification is the following:

```
/*@ ensures
  @   (\old(erase_state)==STARTED        =>
  @        (\forall int k; 0<=k<SIZE   =>
  @             block[k]==0)    )
  @ &&( \old(erase_state)==FINISHED       =>
  @        (\forall int k; 0<=k<SIZE   =>
  @             block[k]== \old(block[k]))    ) */
```

For this program, only one variable m is used, to represent the memory block where `block` is allocated. The transition system for these functions is defined by:

$$\begin{aligned}
&(erase\_transition\ st\ block\ m\ st'\ m')\ \equiv\\
&\quad (\forall\ k \in [0,\mathtt{SIZE}[.\ (acc\ m'\ (block[k])) = 0)
\end{aligned}$$

$$\begin{aligned}
&(abort\_transition\ st\ block\ m\ st'\ m')\ \equiv\\
&\quad (st = \mathtt{STARTED}\ \Rightarrow\\
&\quad\ \ (\forall\ k \in [0,\mathtt{SIZE}[.\ (acc\ m'\ (block[k])) = 0))\ \wedge\\
&\quad (st = \mathtt{FINISHED} \Rightarrow\\
&\quad\ \ (\forall\ k \in [0,\mathtt{SIZE}[.\\
&\quad\quad (acc\ m'\ (block[k])) = (acc\ m\ (block[k]))))
\end{aligned}$$

Now, if we look at the Figure 1 representing an anti-tearing property, the definitions above corresponds to the transition relations when no tearing occurred, i.e. the following transitions:

$$s \xrightarrow{f,\ \neg\,\mathtt{PO}} s' \qquad \text{and} \qquad s \xrightarrow{f_{ab},\ \neg\,\mathtt{PO}} s'$$

The transitions in the case of a card tear remains to be formalised. The idea is to extend Caduceus tool so that the specification may include properties in case of the interruption of the function execution. This enables to use the same methodology to extract the transition system, i.e. the memory states will still be computed by Caduceus. The next section explains this extension of the tool and Section IV will illustrate its use to model anti-tearing properties.

III. Modelling of Card Tears in C programs

In this section, we present how the Caduceus tool has been extended in order to specify and verify a function behaviour in the case of the sudden interruption of its execution.

A program interruption is a sudden modification of the control flow, which provoke the return of the function and that must be "propagated" in the calling functions (if a called function is interrupted, so is the calling function). This description corresponds exactly to the semantic of *exceptions*. For this reason, an interruption can be modelled as an exception that is never caught. More precisely, an interruption can be represented by a call to a function which may non-deterministically raise an interruption exception. Specifying the program behaviour in the case of an interruption thus comes down to specifying the property to hold in the case of the raising of the interruption exception.

This approach has been followed both in [4] and in [5], for Java Card programs. More precisely, these papers propose an extension of JML in order to express properties in the case of a card tear and an extension of a Java Card verification tool, namely LOOP (see [14]) for the first paper and Krakatoa (see [15]) for the second, to prove that these properties are verified at each program point.

Other studies follow a different approach, like [6], which presents the definition of strong invariants (i.e. verified at each program point) in a specific Java modelling logic, called *Dynamic Logic*. But the properties that may be expressed in the case of a card tear in this approach are general properties of the whole program. They are not specific to one function, which is necessary to define anti-tearing properties.

All the related work mentioned deal with the Java Card language and aim at proving the correctness of its transaction mechanism.

In our work, we follow the first approach, but for C programs. The Caduceus tool has been extended in order to model a program interruption by a call to a function that may raise an interruption exception. More precisely, this extension consists in:

- defining an interruption exception;
- declaring an interruption function, whose implementation is not needed and whose specification only indicates that the function may raise the interruption exception;
- adding, at each program point, a call to this interruption function;
- adding a new clause, let say `ensures_interrupt`, to the specification language, whose semantic is to define the property in the case of the raising of the interruption exception.

To be rigorous, it should also be added:

- giving an undefined value to all objects stored in volatile memory.

This last point is not necessary for the proof of anti-tearing properties since the consistency of the card state depends only on persistent objects. It should however be implemented to provide a rigorous methodology for the verification of program correctness and it is part of our future work in this context.

The first problem to solve is that the exception mechanism does not exist in the C language. However, it exists in Why input language, therefore the interruption exception and the interruption function are defined in Why input language:

```
exception   POExc
parameter   tearing_parameter:     tt:unit  ->
   { }
   unit  raises  POExc
   {true  | POExc=>true  }
```

In the same way, the calls to this function have been added not in the C program but in its translation in the Why language. Finally, Caduceus specification language has been extended with a new clause `ensures _interrupt` . This clause is translated in Why language by the property in the case of the raising of the interruption exception. In other words, the following specification:

```
/*@ requires  Pre
  @ assigns   a1,  ...,  am
  @ ensures   Post
  @ ensures_interrupt    Posti  */
T f(t1  p1,  ...,  tn pn);
```

will be translated in Why by:

```
parameter   f_parameter   : p1:t1  -> ...   -> pn:tn  ->
   { Pre }
   T reads  ...  writes  ...  raises  POExc
   { Post  and  assign(...)   | POExc  => Posti  }
```

The second problem to solve is to choose the granularity of an interruption, i.e. what is meant by "at each program point". Indeed, a function may be interrupted between two instructions, but also within a single instruction. For example, an `if-then-else` instruction may be interrupted during the evaluation of the conditional expression. In the same way, if the execution of an instruction such as `((*b1|*b2++)== *b1++)` is interrupted, then the values of `b1` and `b2` depends on the exact moment the interruption occurred. But this instruction is a concise form, or *syntactic sugar*, for the set of instructions `((*b1|*b2)== *b1); b2++; b1++;` . Therefore, a granularity at the level of instructions would imply that two programs that are semantically equivalent (like the two above) would not have the same interpretation. Therefore, in some related work such as [4], the programs are *"desugared"* before analysis, i.e. all the syntactic sugar is eliminated to come down to a minimal language.

In our approach, since the function call is added in the Why file, the program has in fact been *desugared* by the translation. Therefore, the call is added after every elementary instruction.

Example.

Let us illustrate this extension by the example of the erase function and its corresponding abort function. If the erase function is interrupted, the only thing that can be said is that if the erase status has already been changed to `FINISHED` when the tearing occurred, then the erase of the block has been completed before interruption. This is formalised as follows:

```
@ ensures_interrupt
@   (erase_state==FINISHED       =>
@       \forall  int  k;  0<=k<SIZE   =>  block[k]==0)    */
```

Concerning the abort function, if the erase status was `FINISHED` at the beginning of the function, nothing is changed even if a tearing occur. If the status was `STARTED` , then the abort function starts again an erasing, that may be interrupted. Here, what can be said is again that if the erase status is `FINISHED` when the tearing occurred, then the block is erased:

```
@ ensures_interrupt
@   (\old(erase_state)==STARTED        &&
@    erase_state==FINISHED       =>
@       (\forall  int  k;  0<=k<SIZE   =>  block[k]==0)    )
@ &&( \old(erase_state)==FINISHED        =>
@       (\forall  int  k;  0<=k<SIZE   =>
@            block[k]==   \old(block[k]))    ) */
```

Using the extension of Caduceus, we can *prove* that these properties are verified at each point of the source code of the functions.

IV. Formal Proof of Anti-tearing Properties

Using the transition extraction method presented in Section II, together with the extended version of Caduceus explained in Section III, we may automatically define, for any function f, the following modelling in terms of transitions:

$$\begin{aligned}
&(f\_transition\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t}\ \overrightarrow{t'}\ result\ status) \equiv\\
&\quad Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t}) \wedge\\
&\quad (match\ status\ with\\
&\qquad |Val\ \Rightarrow\ Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t}, \overrightarrow{t'})\\
&\qquad |Exn \Rightarrow Posti_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t}, \overrightarrow{t'})\ \ )
\end{aligned}$$

This enables to represent both the "normal" transitions:

$$s \xrightarrow{f,\ \neg\, \mathtt{PO}} s' \text{ and } s \xrightarrow{f_{ab},\ \neg\, \mathtt{PO}} s'$$

by (respectively)

$$(f\_transition\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t}\ \overrightarrow{t'}\ result\ Val)$$
$$(abort\_transition\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t}\ \overrightarrow{t'}\ result\ Val)$$

and the transitions in the case of a card tear:

$$s \xrightarrow{f,\ \mathtt{PO}} s' \text{ and } s \xrightarrow{f_{ab},\ \mathtt{PO}} s'$$

by (respectively)

$$(f\_transition\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t}\ \overrightarrow{t'}\ result\ Exn)$$
$$(abort\_transition\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t}\ \overrightarrow{t'}\ result\ Exn)$$

Therefore, the anti-tearing property, represented in Figure 1, can be formally stated and proved, using the Coq proof system.

Example.

For instance, the transition relations for the erase function and its corresponding abort functions are defined as follows.

Definition 1:

$$\begin{array}{l}(\mathit{erase\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m'\ \mathit{status}) \equiv \\ (\mathit{match}\ \mathit{status}\ \mathit{with} \\ \quad |Val\ \Rightarrow\ (\forall\ k \in [0,\mathtt{SIZE}[.\ (\mathit{acc}\ m'\ (\mathit{block}[k])) = 0) \\ \quad |Exn \Rightarrow\ (\mathit{st}' = \mathtt{FINISHED} \Rightarrow \\ \qquad\qquad (\forall\ k \in [0,\mathtt{SIZE}[.\ (\mathit{acc}\ m'\ (\mathit{block}[k])) = 0)))\end{array}$$

Definition 2:

$$\begin{array}{l}(\mathit{abort\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m'\mathit{status}) \equiv \\ (\mathit{match}\ \mathit{status}\ \mathit{with} \\ \quad |Val\ \Rightarrow \\ \qquad (\mathit{st} = \mathtt{STARTED}\ \ \Rightarrow\ (\forall\ k \in [0,\mathtt{SIZE}[. \\ \qquad\qquad (\mathit{acc}\ m'\ (\mathit{block}[k])) = 0))\ \wedge \\ \qquad (\mathit{st} = \mathtt{FINISHED} \Rightarrow\ (\forall\ k \in [0,\mathtt{SIZE}[. \\ \qquad\qquad (\mathit{acc}\ m'\ (\mathit{block}[k])) = (\mathit{acc}\ m\ (\mathit{block}[k])))) \\ \quad |Exn \Rightarrow \\ \qquad (\mathit{st} = \mathtt{STARTED} \wedge \mathit{st}' = \mathtt{FINISHED} \Rightarrow\ (\forall\ k \in [0,\mathtt{SIZE}[. \\ \qquad\qquad (\mathit{acc}\ m'\ (\mathit{block}[k])) = 0))\ \wedge \\ \qquad (\mathit{st} = \mathtt{FINISHED}\ \Rightarrow\ (\forall\ k \in [0,\mathtt{SIZE}[. \\ \qquad\qquad (\mathit{acc}\ m'\ (\mathit{block}[k])) = (\mathit{acc}\ m\ (\mathit{block}[k])))))\end{array}$$

Now a sequence of interrupted calls to the abort function, ending in a non interrupted call, can be defined by a transitive closure, as follows:

Definition 3:

$$\begin{array}{l}(\mathit{abort\_seq}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m'\ i) \equiv \\ (\mathit{match}\ i\ \mathit{with} \\ \quad |\ \ 0\ \ \Rightarrow\ (\mathit{abort\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m'\ Val) \\ \quad |\ (j+1)\ \Rightarrow \\ \qquad (\exists\ \mathit{st\_aux}.\ \exists\ \mathit{m\_aux}. \\ \qquad\qquad (\mathit{abort\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st\_aux}\ \mathit{m\_aux}\ Exn) \\ \qquad\quad \wedge\ (\mathit{abort\_seq}\ \mathit{block}\ \mathit{st\_aux}\ \mathit{m\_aux}\ \mathit{st}'\ m'\ j)\ \ )\end{array}$$

Finally, the erase operation, as described in the Figure 1, is defined as follows:

Definition 4:

$$\begin{array}{l}(\mathit{erase\_op}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m')\ \equiv \\ \qquad (\mathit{erase\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m'\ Val) \\ \quad \vee\ (\exists\ \mathit{st\_aux}.\ \exists\ \mathit{m\_aux}.\ \exists\ i. \\ \qquad\qquad (\mathit{erase\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st\_aux}\ \mathit{m\_aux}\ Exn) \\ \qquad\quad \wedge (\mathit{abort\_seq}\ \mathit{block}\ \mathit{st\_aux}\ \mathit{m\_aux}\ \mathit{st}'\ m'\ i)\ )\end{array}$$

Let us note that the two previous definitions are general definitions that may be applied to any transition relations defining a function and its corresponding abort function.

Now, the anti-tearing property for the erase operation can be stated as follows:

Theorem 1:

$$\begin{array}{l}\forall\ \mathit{block}.\ \ \forall \mathit{st}.\ \ \forall\ m.\ \ \forall\ \mathit{st}'.\ \ \forall\ m'. \\ \quad (\mathit{erase\_op}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m') \Rightarrow \\ \quad (\forall\ k \in [0,\mathtt{SIZE}[.\ (\mathit{acc}\ m'\ (\mathit{block}[k])) = 0))\end{array}$$

Proof: Of important note is that the various definitions stated in this paper have been defined in Coq, so is the theorem statement. Therefore, the proof has been formally built and mechanically checked in Coq, representing around 30 lines of Coq script. Here we give a proof "by hand" of the theorem, giving the global architecture of the formal Coq proof.

From the definition of $\mathit{erase\_op}$ (Definition 4), the theorem hypothesis implies two cases.

1) The first case is when the erase function is not interrupted:

   $$(\mathit{erase\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st}'\ m'\ Val)$$

   Using definition of $\mathit{erase\_transition}$ (Definition 1), this implies:

   $$(\forall\ k \in [0,\mathtt{SIZE}[.\ (\mathit{acc}\ m'\ (\mathit{block}[k])) = 0)$$

   which is the goal of the theorem.

2) The second case is when the erase function is interrupted and a sequence of abort follows the reset:

   $$\begin{array}{l}(\ \exists\ \mathit{st\_aux}.\ \exists\ \mathit{m\_aux}.\ \exists\ i. \\ \qquad (\mathit{erase\_transition}\ \mathit{block}\ \mathit{st}\ m\ \mathit{st\_aux}\ \mathit{m\_aux}\ Exn) \\ \quad \wedge\ (\mathit{abort\_seq}\ \mathit{block}\ \mathit{st\_aux}\ \mathit{m\_aux}\ \mathit{st}'\ m'\ i)\ )\end{array}$$

   Using the definition of $\mathit{erase\_transition}$ (Definition 1), this implies:

   $$\begin{array}{lr}(\ \exists\ \mathit{st\_aux}.\ \exists\ \mathit{m\_aux}.\ \exists\ i. & \\ \quad (\mathit{st\_aux} = \mathtt{FINISHED} \Rightarrow (\forall\ k \in [0,\mathtt{SIZE}[. & \\ \qquad (\mathit{acc}\ \mathit{m\_aux}\ (\mathit{block}[k])) = 0))) & (1) \\ \ \wedge\ (\mathit{abort\_seq}\ \mathit{block}\ \mathit{st\_aux}\ \mathit{m\_aux}\ \mathit{st}'\ m'\ i)\ ) & (2)\end{array}$$

   From now, the proof is done by induction on i.

   a) If $i = 0$, then the definition of $\mathit{abort\_seq}$ (Definition 3) and (2) imply:

      $$(\mathit{abort\_transition}\ \mathit{block}\ \mathit{st\_aux}\ \mathit{m\_aux}\ \mathit{st}'\ m'\ Val)$$

      Using the definition of $\mathit{abort\_transition}$ (Definition 2), this implies:

      $$\begin{array}{lr}(\mathit{st\_aux} = \mathtt{STARTED}\ \Rightarrow\ (\forall\ k \in [0,\mathtt{SIZE}[. & \\ \quad (\mathit{acc}\ m'\ (\mathit{block}[k])) = 0))\ \wedge & (3) \\ (\mathit{st\_aux} = \mathtt{FINISHED} \Rightarrow (\forall\ k \in [0,\mathtt{SIZE}[. & \\ \quad (\mathit{acc}\ m'\ (\mathit{block}[k])) = & \\ \quad (\mathit{acc}\ \mathit{m\_aux}\ (\mathit{block}[k])))) & (4)\end{array}$$

      If $\mathit{st\_aux} = \mathtt{STARTED}$, then (3) gives the theorem goal. If $\mathit{st\_aux} = \mathtt{FINISHED}$, then (4) gives:

      $$(\mathit{acc}\ m'\ (\mathit{block}[k])) = (\mathit{acc}\ \mathit{m\_aux}\ (\mathit{block}[k]))$$

      and (1) gives:

      $$(\mathit{acc}\ \mathit{m\_aux}\ (\mathit{block}[k])) = 0.$$

      Therefore we have the theorem goal:

      $$(\mathit{acc}\ m'\ (\mathit{block}[k])) = 0.$$

   b) Now, if $i = (j+1)$, then the definition of $\mathit{abort\_seq}$ (Definition 3) and (2) imply:

$$(\exists\ st\_aux'.\ \exists\ m\_aux'.$$
$$(abort\_transition\ block\ st\_aux\ m\_aux\ st\_aux'\ m\_aux'\ Exn) \quad (5)$$
$$\wedge\ (abort\_seq\ block\ st\_aux'\ m\_aux'\ st'\ m'\ j)) \quad (6)$$

Besides, the induction hypothesis is the following:

$$\forall\ st\_aux'.\ \forall\ m\_aux'$$
$$(st\_aux' = \texttt{FINISHED} \Rightarrow (\forall\ k \in [\texttt{0}, \texttt{SIZE}[.\ (acc\ m\_aux'\ (block[k])) = \texttt{0})))) \Rightarrow \quad (H1)$$
$$(abort\_seq\ block\ st\_aux'\ m\_aux'\ st'\ m'\ j) \Rightarrow \quad (H2)$$
$$(\forall\ k \in [\texttt{0}, \texttt{SIZE}[.\ (acc\ m'\ (block[k])) = \texttt{0}))$$

Therefore, the theorem is proved by applying the induction hypothesis. The hypothesis $(H2)$ is given by (6). It only remains to prove hypothesis $(H1)$. Let us assume that:

$$st\_aux' = \texttt{FINISHED} \quad (H)$$

and prove that:

$$(\forall\ k \in [\texttt{0}, \texttt{SIZE}[.\ (acc\ m\_aux'\ (block[k])) = \texttt{0}) \quad (CCL)$$

Using the definition of $abort\_transition$ (Definition 2), the hypothesis (5) implies:

$$(st\_aux = \texttt{STARTED}\ \wedge\ st\_aux' = \texttt{FINISHED} \Rightarrow (\forall\ k \in [\texttt{0}, \texttt{SIZE}[.\ (acc\ m\_aux'\ (block[k])) = \texttt{0}))\ \wedge \quad (7)$$
$$(st\_aux = \texttt{FINISHED}\ \Rightarrow (\forall\ k \in [\texttt{0}, \texttt{SIZE}[.\ (acc\ m\_aux'\ (block[k])) = (acc\ m\_aux\ (block[k]))))) \quad (8)$$

If $st\_aux = \texttt{STARTED}$, since $st\_aux' = \texttt{FINISHED}$ by (H), the hypothesis (7) gives the conclusion (CCL). If $st\_aux = \texttt{FINISHED}$, the hypothesis (8) gives:

$$(acc\ m\_aux'\ (block[k])) = (acc\ m\_aux\ (block[k]))$$

and the hypothesis (1) gives:

$$(acc\ m\_aux\ (block[k])) = \texttt{0}$$

Therefore we have the conclusion (CCL):

$$(acc\ m\_aux'\ (block[k])) = \texttt{0}.$$

Case Study.

Let us note that, for readability reasons, a trivial erase function has been given in example in this paper. However, this method has been used to prove a module of an real embedded operating system. In this real case study, the anti-tearing property has been proved for a function erasing a block of memory, but in Flash memory. It implies that the process to know if the erase operation has been interrupted is much more complicated. What is highlighted from this case study is that the anti-tearing proof itself is not much more complicated that the one presented in this paper, the hard task being to prove that the specification given in the annotations is verified by the source code (i.e. proving the verification conditions generated).

V. Conclusion

The anti-tearing property is critical in the smart card world, in terms of both security and safety. A sudden remove of the card from the terminal may lead to a fatal error or may be maliciously used, if the card is not well programmed. In order to avoid such possible problems and ensure a safe and secure behaviour of the card, embedded programs, specially those of an operating system, must be developed in such a way that they may resist to card tears. This implies the programming of anti-tearing measures. Proving the correctness of such measures becomes crucial, but requires to define a high level model of the source code in terms of a transition system, where the a card tear may be modelled.

This paper presents a methodology to formally prove anti-tearing properties of C source code embedded into smart cards. The main advantages are the automatic extraction of the transition system and the formal link between this transition system and the source code of the program. This enables to formally prove that some high level properties, such as the anti-tearing property, are verified by the source code itself. It has required the extension of a C program verification tool, called Caduceus, where a card tear, i.e. a sudden interruption of a program, has been formalised. This extension may be used, by itself, to prove the correctness of a program in the case of a card tear. Combined with the extraction of a transition system, it enables to keep the advantages of the automatic extraction of the transition system to prove global properties in the case of a card tear, such as anti-tearing properties.

The use of this method in a concrete case study of an embedded operating system demonstrates its feasibility and interest and represents a promising approach for wider use of formal methods to strengthen the confidence in embedded programs.

Acknowledgement.

The author acknowledge Christine Paulin-Mohring and Boutheina Chetali for their advises for this work.

References

[1] J. Andronick, "Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur – Plate-Forme Java Card et Système d'Exploitation," Ph.D. dissertation, Université Paris-Sud, 2006.

[2] J. Andronick, B. Chetali, and C. Paulin-Mohring, "Formal Verification of Security Properties of Smart Card Embedded Source Code," in *Formal Methods, International Symposium of Formal Methods Europe (FM'05)*, ser. LNCS, J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds., vol. 3582. Springer-Verlag, 2005, pp. 302–317.

[3] "Java Card," http://java.sun.com/products/javacard/.

[4] E. Hubbers and E. Poll, "Reasoning about Card Tears and Transactions in Java Card," in *Fundamental Approaches to Software Engineering (FASE'04)*, ser. LNCS, M. Wermelinger and T. Margaria, Eds., vol. 2984. Springer, 2004, pp. 114–128.

[5] C. Marché and N. Rousset, “Verification of Java Card Applets Behavior with respect to Transactions and Card Tears,” in *Conference on Software Engineering and Formal Methods (SEFM’06)*, 2006, to appear.

[6] B. Beckert and W. Mostowski, “A Program Logic for Handling JAVA CARD’s Transaction Mechanism,” in *Fundamental Approaches to Software Engineering (FASE’03)*, ser. Lecture Notes in Computer Science, M. Pezzè, Ed., vol. 2621. Springer, 2003, pp. 246–260.

[7] M. Huisman and K. Trentelman, “Extending JML Specifications with Temporal Logic,” in *Algebraic Methodology And Software Technology (AMAST’02)*, ser. LNCS, vol. 2422. Springer-Verlag, 2002, pp. 334–348.

[8] J.-C. Filliâtre and C. Marché, “Multi-Prover Verification of C Programs,” in *Sixth International Conference on Formal Engineering Methods (ICFEM)*, ser. LNCS, vol. 3308. Seattle: Springer-Verlag, 2004, pp. 15–29.

[9] J.-C. Filliâtre, C. Marché, and T. Hubert, “The Caduceus tool for the Verification of C Programs,” http://why.lri.fr/caduceus/.

[10] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, “JML: Notations and Tools Supporting Detailed Design in Java,” in *OOPSLA 2000 Companion*. ACM, 2000, pp. 105–106.

[11] J.-C. Filliâtre, “Verification of Non-Functional Programs using Interpretations in Type Theory,” *Journal of Functional Programming*, vol. 13, no. 4, pp. 709–745, 2003.

[12] J.-C. Filliâtre, “The Why Verification Tool,” http://why.lri.fr/.

[13] The Coq Development Team LogiCal Project, *The Coq Proof Assistant Reference Manual*, http://coq.inria.fr.

[14] “The LOOP Tool,” http://www.sos.cs.ru.nl/research/loop.

[15] C. Marché, C. Paulin-Mohring, and X. Urbain, “The Krakatoa Tool for Java Program Verification,” 2002, http://krakatoa.lri.fr/.