Verifying Extended Criteria for the Interoperability of Security Devices

Maurizio Talamo[3,1], Franco Arcieri[1], Giuseppe Della Penna[2], Andrea Dimitri[1], Benedetto Intrigila[3], and Daniele Magazzeni[2]

[1] Nestor Lab - University of Roma "Tor Vergata", Italy
[2] Department of Computer Science, University of L'Aquila, Italy
[3] Department of Mathematics, University of Roma "Tor Vergata", Italy

Abstract. In the next years, smart cards are going to become the main personal identification document in many nations. In particular, both Europe and United States are currently working to this aim. Therefore, tens of millions of smart cards, based on hardware devices provided by many different manufacturers, will be distributed all over the world, and used in particular to accomplish the security tasks of *electronic authentication* and *electronic signature*. In this context, the so called *Common Criteria* define the security requirements for digital signature devices. Unfortunately, these criteria do not address any interoperability issue between smart cards of different manufacturers, which usually implement digital signature process in still correct but slightly different ways.
To face the interoperability problem, we realized a complete testing environment whose core is the *Crypto Probing System* ©*Nestor Lab*, an abstract interface to a generic cryptographic smart card, embedding a standard model of the correct card behavior, which can be used to test the digital signature process behavior, also in the presence of alternate or disturbed command sequences, in conjunction with automatic verification techniques such as *model checking*. The framework allows to verify *abstract behavior models* against *real smart cards*, so it can be used to automatically verify the Common Criteria as well as the extended interoperability criteria above and many other low-level constraints. In particular, in this paper we show how we can verify that the card, in the presence of a sequence of (partially) modified commands, rejects them without any side effect, remaining usable, or accepts them, generating a correct final result. To exemplify our framework, in the present paper we show how it can be used to check that the card [STM-Incrypto34] is actually robust with respect to unknown or alternate command sequences during the digital signature process. Even in this simple experimentation, it has been possible to point out some anomalous behavior of the card, such as the unexpected acceptance of wrong commands. Therefore, this verification can be considered as an instance of a whole new kind of testing processes for the digital signature devices.

1 Introduction

Starting from 2010, the *European Citizen Card* [1] is going to be developed and distributed to all European citizens. In the next years, tens of millions of smart cards, based on hardware devices provided by many different manufacturers, will be distributed in

Europe, and one of the main purposes of such cards will be to provide an easy and safe way for the European citizens to generate and use security data for *electronic authentication* and for *electronic signature*.

Therefore, many European countries have started projects to test and validate such cards before they are put on the market. In particular, the problem has been faced in the Italian National Project for Electronic Identity Card [2], designed by one of the authors of this paper. Among the other issues addressed by this project, there is the very important problem of the *interoperability* between smart cards produced by different manufacturers.

In this context, the so called *Common Criteria* (ISO/IEC 15408, [3]) and the CWA-1469 [4] standards define the requirements of security devices for digital signature conforming to the annex III of EU directive 1999/93/CE, and the criteria to be followed to verify this conformance. In particular, they define a set of general rules and formats for the microprocessor of a smart card to work correctly as a digital signature device. Similar standards are going to be defined for other security tasks, like the electronic authentication based on smart cards.

The digital signature process, defined by these standards, is coherent and unitary but its implementation is different from a smart card to another.

Indeed, smart cards should generally communicate with their readers using the APDU protocol defined by ISO 7816-4 and 7816-8, but smart card families can implement the APDU protocol in a variety of ways, and the Common Criteria compliancy only certifies that, given the *specific* sequence of commands implemented by the smart card manufacturer, the card generates the required digital signature respecting the security requirements requested by the Common Criteria.

In this context, much responsibility is left to the card readers and client applications, which must have intimate knowledge of the specific command set of the smart card they are communicating with. Observe that one relevant problem is related to *extra-capabilities* added to the software by some weakness in the production process. In particular, one can consider: new commands not included in the standards, the possibility of inappropriately storing relevant information related to the transaction and so on. It is clear that due to the complexity of the production process, which is moreover not standardized, even a trustable producer cannot certify the complete trustability of the software. This makes the signature of the software only a partial solution: ”the signature guarantees the origin of the cardlet, not that it is innocuous” [5]. Another possibility is to make an effort to standardize the production process and make use of suitable software engineering approaches such as *Correctness By Construction* [6], [7]. The current situation, described above, makes however this solution not viable.

So we are faced with the problem of setting up a comprehensive verification process able to severely test a given card and its software applications. The task of this verification is to give (at least partial) answers to the many questions not addressed by the standards. What happens if, for example, a card receives a command sequence that is certified as correct for another card? This may happen if the card client cannot adapt to its specific command set, or does not correctly recognize the card. Obviously, this problem may be also extended to a more general context, where the card is deliberately *attacked* using incorrect commands.

Unfortunately, the common criteria do not address any interoperability issue of this kind. This could lead to many problems, especially when designing the card clients, which should potentially embed a driver for each family of card on the market, and be easily upgradable, too. This would be difficult and expensive, and could slow down the diffusion of the smart cards. Therefore, the interoperability problem must be addressed in detail.

1.1 Our Contribution

To face the interoperability problem, we realized a complete testing environment whose core is the *Crypto Probing System* ©*Nestor Lab*, an abstract interface to a generic cryptographic smart card, which embeds a standard model of the correct card behavior.

The Crypto Probing System can be used to transparently interface with different physical smart cards and test the digital signature process. Indeed, it will be used by the Italian Government to test the card conformance to the Italian security and operational directives for the Electronic Identity Card project.

However, the Crypto Probing System can be also used to test the digital signature process behavior in the presence of alternate or disturbed command sequences. Indeed, in this paper we will use this feature to check a first interoperability issue, i.e., automatically and systematically verify the card behavior when stimulated with unexpected input signals and/or unexpected sequences of commands or parameters. Note that, despite of this simple level of heterogeneity, the common criteria cannot ensure the smart cards interoperability even in this case.

Verifying that a specific card is compliant with such extended criteria requires a verification that is much more extensive than the usual standard black box testing process. Therefore, we decided to exploit *model checking techniques* to automate the verification process. We decided to use the model checker Murφ [8]. Actually we used the tool CMurφ [9], realized by some of the authors with other researchers, which extends Murφ, among others, with the capability of use external C/C++ functions. The model checker will be interfaced to the Crypto Probing System and through it, transparently, to the smart card.

To validate our integrated verification framework, in the present paper we show an experiment used to check that the card [STM-Incrypto34] is actually compliant with the extended criteria mentioned above. This verification can be considered as an instance of a whole new kind of testing processes for the digital signature devices.

1.2 Related Works

Although much research is being done in the field of smart card verification[6, 10, 11], most of the works in the literature address the more generic problem of (Java) *byte code verification*. In these works, the verification is performed on the applications that should run over the card hardware and operating system, to check if the code respects a set of safety constraints. From our point of view, the major drawback of this approach is that it assumes the correctness of the smart card hardware/firmware and of the Java Virtual Machine which is embedded in the card. Thanks to our abstract machine, the Crypto Probing System, in this work we show how such application code can be tested

directly *on the real card*. In this way, we are able to validate both how the code affects the behavior of the card, and how the card hardware can affect the code execution.

Moreover, currently we are not interested in testing a specific smart card software, but rather in verifying the correctness of the smart card output after a sequence of possibly disturbed or unexpected commands, and in particular we aim to use these experiments to address the smart card interoperability issue.

2 Extending the Common Criteria

As mentioned above, the Common Criteria [3] define a set of general rules for a smart card-based digital signature process to work *correctly*. However, there are technical aspects that the Common Criteria do not address at all, being too ”high level“ to analyze issues related to the card application code. For instance, no requirements are given for the card behavior when an unexpected command is received during the digital signature process. In this case, what should be defined as ”the correct behavior” of the card processor?

We considered the following *extended criteria* for a correct behavior:

- if a result can be obtained it is always the correct one; or
- the wrong command is rejected but the card remains usable.

Observe that the second requirement is needed to avoid denial of service attacks or malfunctioning. Here we assume that an error in the command (as opposed to an error in parameters) should not compromise the availability of the card. This assumption can be of course questioned, and this shows the need of more detailed criteria.

2.1 Robustness of the Signature Process

As a first example of extended smart card security property, in this paper we propose the following problem related to the digital signature process.

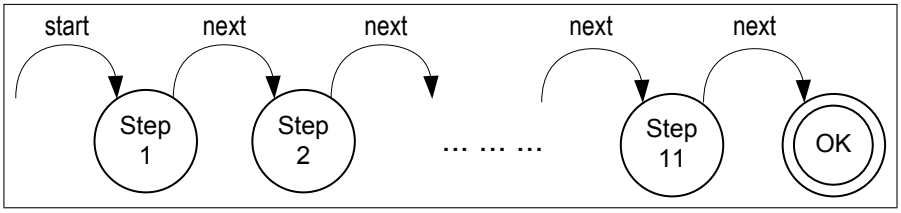

Fig. 1. Digital Signature Process Model

The digital signature process can be generally split in eleven main steps [4]. At each step, the card expects a particular set of commands and parameters to proceed to the next step:

1. Logically reset the Smart Card.
2. Change directory to the electronic signature directory.
3. Select the public certificate (file) for electronic signature.
4. Define the selected public certificate (file) as the public certificate to be used for the next operation of electronic signature.
5. Select, inside the smart card, the private key (security object) to be used for the next operation of electronic signature.
6. Ask the smart card for a Random Number (security operation).
7. Give a random number to the smart card (security operation).
8. Ask the smart card to enable the usage of the selected private key for a signature operation (security operation).
9. Ask the smart card for a Random Number (security operation).
10. Give a random number to the smart card (security operation).
11. Give to the smart card the buffer to be signed and get the resulting signature (security operation).

An abstract view of the process is sketched in Fig.1).

Now we want to consider possible random disturbances in the commands sent to the smart card. In the presence of such disturbances, we may expect the process to behave, at each step, as sketched in Fig.2. In other words, as discussed above, either the invalid commands is refused leaving the process unaltered, or the wrong command is accepted but the final result is *identical* to the one obtained with the right command. The existence of *erroneous accepted command* is due to the presence of bits which are *uninfluential* on the parsing of the command syntax.

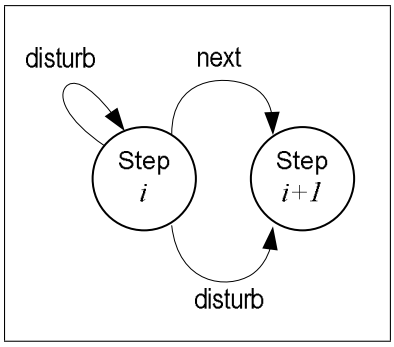

Fig. 2. Disturbed Step

In this scenario, we want to check the *smart card robustness*. That is, we want to verify that *any* possible disturbance is *correctly* handled by the card, where *correctness* refers to the model discussed above. It is clear that such a verification cannot be performed manually, but requires an automatic framework where the model of Fig. 2 can be analyzed and exhaustively compared with the behavior or the real smart card

hardware. As we will see in Section 4, model checking is a perfect candidate for this task.

3 The Crypto Probing System

The core of a smart card is its *microprocessor*, which contains, on board, a cryptographic processor, a small EEPROM random access memory ($\approx$ 64 KBytes), an operating system and a memory mapped file system.

The microprocessor can execute a restricted set of operations named APDUs (*Application Control Data Units*), which can be sent from external software applications through a serial communication line.

The standard ISO 7816 part 4, specifies the set of APDU's that any compatible smart card microprocessor must implement. In particular, an APDU consists of a mandatory header of 4 bytes: the Class Byte (*CLA*), the Instruction Byte (*INS*) and two parameter bytes (*P1*,*P2*). The header can be followed by a conditional body of variable length, which is composed by the length (in bytes) of the data field (*Lc*), the Data field itself and the maximum number of bytes expected in the data field of the response (*Le*). Responses to any APDU are encoded in a variable length data field and two mandatory trailer bytes.

However, as described in the introduction, the microprocessor manufacturers develop the APDUs with some deviations from the standards or, in some cases, they create new APDUs not defined by the standards.

Therefore, in order to interface with any kind of card, the client applications should know in advance their command set: no insurance is given that the same APDU sequence will be accepted by all the cards. To investigate this issue, we developed the Crypto Probing System (*CPS*), an executable abstract smart card model, with a simplified set of commands that can act as a *middleware* between the external applications and the real smart cards.

The CPS is able to translate its simplified instructions to the corresponding sequence of APDUs to be sent to the connected physical smart card and to translate the smart card responses in a common format. Moreover, to further simplify the interface with the smart card, the CPS knows in advance *the correct sequence of APDUs* to be sent in each step of the digital signature process, and is able to generate *alternate command sequences* to test the card responses in different situations. In this way, the CPS offers a simple interface for testing applications to verify the process correctness and robustness on different physical devices.

The CPS, whose overall architecture is shown in Fig. 3, can be invoked via command line, to interactively test the command sequences, or used as a daemon, which stays in execution and accepts commands on TCP/IP connections. The CPS instruction set is the following.

- `reset`: resets the card.
- `start`: initializes the signature process.
- `next`: executes the next command in the process sequence using the correct cla and ins values and advances to the next step.

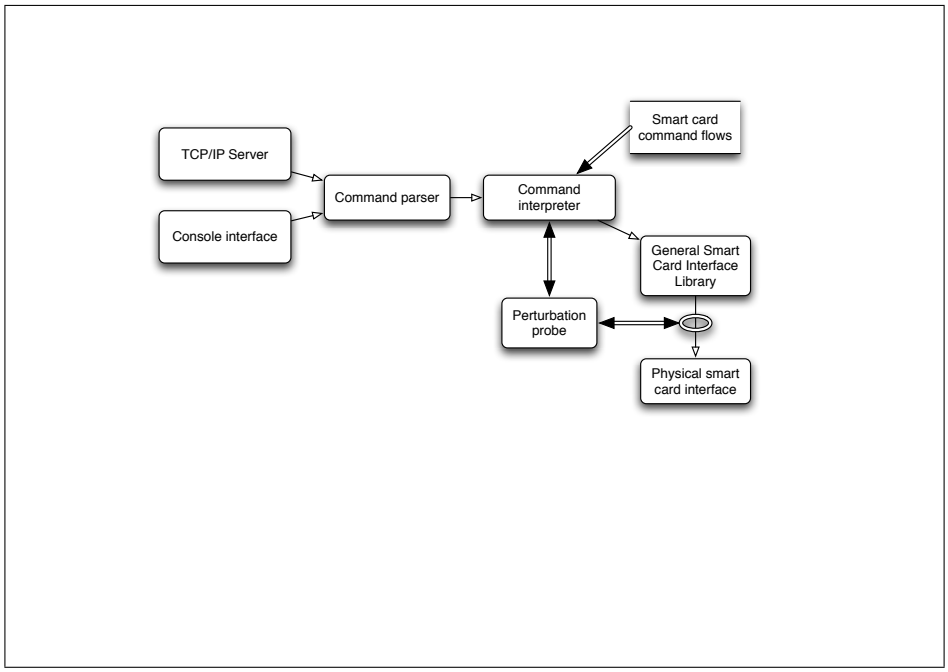

Fig. 3. Architecture of the Crypto Probing System

– `stay [cla | ins] [r | s] [leave | restore]`: executes the next command in the process sequence, applying a disturbance to its CLA or INS parameters, but does not advance to the next step. In particular, the `[r | s]` modifiers specify a random (uniform generator rand48) or sequential (starting from the current value of the parameter) disturbance, whereas the `[leave | restore]` modifiers tell the CPS to leave the last value or restore the original value of the other non disturbed parameter.
– `accept`: executes the next command in the process sequence using the same values of CLA and INS of the last `stay` command and advances to the next step.

All the instructions above return:

1. the current hexadecimal node number (01..0B), representing the reached step in the signature process,
2. the values of CLA and INS sent to the card by the last command,
3. the result code of the command (where a nonzero value indicates an error condition),
4. the overall status of the signature process, i.e.,
   – TERMINATED if the card has correctly reached the final step of the process, obtaining the same result as the right sequence,
   – UNTERMINATED if the card has not still reached the final step,
   – or WRONG if the card has reached the final step but with an incorrect result, that is with a result different from the one of the right sequence.

After a `start`, each `next` sends the next correct APDU to the card and advances to the next step of the process. After the last `next` command is issued, the card must return a TERMINATED status.

The `stay` command sends to the card a *disturbed* version of the next correct APDU, but does not advance to the next step of the process. This is indeed accomplished by issuing an `accept` command, which commits the previous modified command generated by a `stay` and proceeds with the process.

In other words, `stay` can be used to test the card response to a modified command at a specific step, whereas the combination `stay` and `accept` can be used to build a process containing one or more modified commands.

If the cards behaves accordingly to the extended criteria mentioned above, either a modified command is rejected but the card remains in the current state and is able to reach a TERMINATED status, or the card always remains in the UNTERMINATED status. Of course, what we want to exclude is the possibility that *a perturbed command generates a different signed document*: this corresponds to never reach the WRONG status.

An example of CPS session is shown in Fig 4, where the lines beginning with a $>$ represent the input commands, followed by the CPS response.

```
>  reset
>  start
>  stay cla r restore
01 D9 A4 6986 UNTERMINATED
>  stay ins r leave
01 B4 86 6986 UNTERMINATED
>  next
01 00 A4 0000 UNTERMINATED
>  next
02 00 A4 0000 UNTERMINATED
>  stay cla s leave
03 01 A4 0000 UNTERMINATED
>  accept
03 01 A4 0000 UNTERMINATED
>  stay cla s leave
04 01 22 0000 UNTERMINATED
>  accept
04 01 22 0000 UNTERMINATED
>  next
05 00 21 6986 UNTERMINATED
>  stay ins r leave
05 00 4E 6986 UNTERMINATED
...
>  next
0A 80 86 0000 UNTERMINATED
>  next
0B 0C 2A 0000 TERMINATED
>  next
0B 0C 2A 0000 TERMINATED
```

Fig. 4. Example of signature session on the CPS

4 The Role of Model Checking

The compliancy of a smart card to the Common Criteria is a testing problem, i.e., it can be certified by manually or semi-automatically by reproducing the context and events described by each criterion and then verifying the expected card behavior.

However, more complex correctness or security properties, like those proposed in this paper (Section 2.1), which work on a lower level, cannot be handled by testing, and require more powerful verification methods. Using model checking it is possible to exhaustively check the compliance of the smart card w.r.t. an *extended* model such as the one of Fig. 2.

Indeed, from a conceptual point of view, model checking can be used to create an executable model of an hardware/software device, analyze all its possible execution states (*reachability analysis*), and finally check that any of such states satisfy a set of properties (*invariants*). Observe that there are basically two possible implementation of model checking techniques: *symbolic* (i.e. OBDD-based) algorithms and *explicit* algorithms. In our context, due to the need of a real time connection with the smart card system, symbolic algorithms are completely ruled out. As already mentioned, we decided to use the model checker Murφ [8], which implements explicit algorithms. Actually we used the tool CMurφ [9], realized by some of the authors with other researchers, which extends Murφ, among others, with the capability of use external C/C++ functions.

Of course, as it well known, when the size of the state space is too large, the so called *state space explosion* phenomenon occurs [12]. That is, the model checker cannot complete the verification problem. To attenuate this problem, a possibility is to not consider the set of all possible state but a large subset. Another strategy is based on introducing equivalence relations between states so to check only one state in each equivalence class. Finally, another possibility is to distribute the verification task using distribute architectures. The implementation of the latter strategy will be considered in a future work. In the present work we use the former one.

In our context, a *correct* smart card is modeled as a finite automaton like the one shown in Fig. 2. This is enough to check the digital signature robustness property, but the model may be extended and enriched to support the verification of almost any property: indeed, model checking is able to deal with very extended systems, having millions of states [13]. Disk-based technologies, implemented by some of the authors with others researchers, allows the verifications of systems with *billions* of states [14].

Then, the possible disturbances (or - if we think to malicious disturbances - smart card *attacks*) are also modeled within the verifier: i.e., we have a “card model” and “a disturbance model” (or an “attacker model”) that will be run in parallel by the verifier. Finally, the verifier is interfaced with a real smart card (see Section 5 for details).

In this framework, model checking will be used to generate all the possible disturbances (e.g., unexpected commands) that can be carried on the smart card (or, at least, a large subset of such possible disturbances). Then, these actions will be performed on the real smart card, and their results compared with the ones of the correct smart card model.

The property to verify is clearly that the real card has the desired correct behavior in any possible situation.

5 Integrating the Crypto Probing System with the CMurφ Model Checker

Having clarified the role of model checking in this extended smart card verification framework, in this Section we describe how this technique has been actually integrated with a smart card system to check the digital signature robustness property. However, as we will see, the presented methodology is very general, so it could be used to verify many other extended smart card properties.

5.1 The CMurφ Model Checker

In this paper we use the CMurφ tool [9]. Of course, our framework could also be used with different model checkers (however, as observed before, they must use explicit algorithms).

In CMurφ, the system to be verified is described through a set of (state) variables, which uniquely define the system state, and the system dynamics is given by a set of language constructs called (transition) *rules*. Finally, the properties to be verified are encoded in boolean expressions called *invariants*.

However, the most useful aspect of CMurφ is its extension [15], that allows to embed externally defined C/C++ functions in the modeling language.

Namely, in its standard working, Murφ is initially given a `system.m` file containing the description of the system under analysis. Then, Murφ generates a C++ file `system.C`, which contains the behavior of the system translated in C++.

With CMurφ, it is possible to use the keyword `externfun` in the `system.m` file to declare and then use functions defined in external C/C++ source files directly in the model. The only constraint to these functions is that they should not have collateral effects and must accept/return parameters of double and integer type only.

This feature will be used to interface CMurφ with the real smart card device through a suitable interface library, as described in the following sections.

5.2 The CMurφ Model

The first part of the CGMurφ model consists of the *declarative statements*. Figure 5 shows the declaration of constants, datatypes, and external functions (the `externproc` construct refers to a procedure without a returning value).

Moreover, the state of the system is represented by the set of the *state variables*. Since our system represents the *interaction between the user and the smart card*, we consider the internal state of the smart card (i.e. its current node within the authentication process and its status) and the possible actions of the user (i.e. the type of disturbance to apply and the node to be disturbed).

As second step, we have to define the *start states*, representing the first states of each authentication session we want to validate. To do this, as shown in Figure 6, we use the `ruleset x` construct that allows to consider each value expected for variable `x` and use it in the following start state definition. Note that the `restart` external function resets the smart card initializing its internal variables.

```
const
  NUM_TESTS : 250;
  NUM_DISTURBANCES : 4;

type
  node_id_type : 0..11; --0=default initial node
  dist_node_type : 0..11;
  dist_id_type : 0..NUM_DISTURBANCES*NUM_TESTS;

var --state description
  curr_node: node_id_type; --current node id
  status : 0..2; --UNTERMINATED,TERMINATED,WRONG
  dist_node : dist_node_type; --node to be disturbed
  dist_id : dist_id_type; --disturbance to apply

externfun next() : int_type "protocol.h";
externfun accept() : int_type;
externfun login() : int_type;
externproc stay(dist_id : int_type);
```

Fig. 5. Constants, datatypes, state description and external functions within the CGMurφ Model

```
ruleset dist_node : 2..9 do
  ruleset dist_id : dist_id_type do
    startstate "Start Authentication"
      dist_node := dist_node;
      dist_id := dist_id;
      curr_node := 0;
      status := restart(dist_node,dist_id);
    end;
  end;
end;
```

Fig. 6. Start states definition

In order to describe the evolution of the system, we have to define the *guarded transition rules*. Figure 7 shows the three main rules of the model:

- **next**: this rule simply models a normal command sent to the smart card;
- **disturb**: this rule sends a disturbed command to the smart card (through the `stay` function). Moreover, depending on the returned value (obtained with the `check_CODE` function), it sends the command `accept` or `next` according to their semantics given in Section 3.
- **end authentication**: this rule is executed at the end of the authentication process and checks the final status of the smart card (i.e. if the signature is correct or not).

Note that the rules are mutual exclusive, thanks to the use of the guards.

Finally, we have to define the *invariant*, that is the property which has to be satisfied in each state of the system. In our case, we want the status of the smart card to be different from `WRONG` during each step of the authentication process, as shown in Figure 8.

```
rule "next" (status=0 & current_node!=dist_node) ==>
begin
    status := next();
    if (check_CODE()!=1) then --ERROR
        dump_NODE_CODE();
    endif;
    status := get_STATUS();
    curr_node := get_NODE();
end;

rule "disturb" (status=0 & curr_node=dist_node) ==>
begin
    stay(disturbance_id);
    if (check_CODE()=1) then
        dump_DISTURB(1); --IGNORE
        status := accept();
    else
        dump_DISTURB(0); --ACCEPT
        status := next();
    endif;

    status := get_STATUS();
    curr_node := get_NODE();
end;

rule "end authentication" (status=1) ==>
begin
    dump_STATUS();
end;
```

Fig. 7. Guarded Transition Rules

```
invariant "correct status"
    (status != 2) --i.e. status!='WRONG'
```

Fig. 8. Invariant Property

5.3 The Integrated Framework

In our verification framework, the CMurφ model must be able to access and drive a real smart card.

To this aim, we set up an environment where a smart card is connected to a computer-based host running the Linux operating system. The CPS (see section 3) daemon runs on the same machine and interfaces with the card, whereas a simple TCP/IP connection library, whose functions are exported into the CMurφ model, allows the verifier to talk with the CPS.

At this point, given a model of the digital signature process, we can program CMurφ to exhaustively test the card behavior by simulating all the possible scenarios. In this way, we are able to verify the compliance of the smart card w.r.t. the model.

6 Experimentation

To verify the smart card behavior in the presence of erroneous commands during the digital signature process described in Section 2, we used the CMurφ-based model presented in Section 5 and the framework as described in the following.

Let $s_1, \ldots, s_{11}$ be the steps of the digital signature process. Moreover, let $c(s_i)$ be the command that should be sent to the card at the step s_i to correctly proceed to the next step s_{i+1}. Finally, let $disturb(x)$ a function that, given any binary data x, returns it with an added random disturbance.

Then, according to the CMurφ model described in the previous section, the verification procedure is summarized by the algorithm shown in Figure 9.

```
for i = 1 to 11 {
    for k = 1 to MAX_TESTS {
        Start a new signature session
        for j = 1 to i-1
            send c(s_j) to the card
        /* now we are at step s_i − 1 */
        let cr = disturb(c(s_j))
        send cr to the card
        let resp = the current card status
        if (resp == Error) /* we are still at step s_i − 1 */
            send c(s_j) to the card
            /* otherwise the disturbed command has been accepted, so we are at
             step s_i */
        for j = i+1 to 11
            send c(s_j) to the card
        /* now we should be at the final step */
         verify the card output validity
    }
}
```

Fig. 9. Experiment 1: one disturbed command in each authentication session

In other words, we test each step s_i of the process for robustness with respect to MAX_TESTS randomly generated bad commands. To this aim, we first execute s_{i-1} correct steps, and then issue a disturbed command. The card can accept the command and go to state s_i, or refuse it and stay in state s_{i-i}. In either case, we send the remaining correct commands until we are in state s_{11} and then test the card output.

In a first experiment, we sent *only one* disturbed command in each authentication session, obtaining the results shown in Table 1 (row "Exp. 1"). The smart card behavior was acceptable, since it did not produce any incorrect results.

However, we may note that the 1.9% of the modified commands sent to the smart card was accepted as a correct one and the execution proceeded to the next step in the digital signature process. Apparently, this did not cause any problem, since the final result was correct, but leaves some doubts about the card software. We may suppose that the card does not support only one digital signature process, but several variants triggered by alternate commands in some steps. In the worst case, however, these modified commands may create a "hidden damage" to the card, that may show its consequences only later.

To further investigate this issue, as second experiment, we stressed the smart card in a more intensive way. Namely, we sent a disturbed command *at each step* of the authentication process. The new verification procedure is sketched in Figure 10 and the results are shown in row "Exp. 2" of Table 1.

```
for k = 1 to MAX_TESTS {
  Start a new signature session.
  for j = 1 to 11 {
    let cr = disturb(c(s_j))
    let dr = disturb(d(s_j))
    send ⟨cr, dr⟩ to the card
    if receive()==Error //We are still at step s_i − 1
      send ⟨c(s_j), d(s_j)⟩ to the card
    /*else the disturbed command is accepted*/
  }
  /* now we should be at the final step */
  verify the card output validity
}
```

Fig. 10. Experiment 2: a disturbed command at each step of the authentication process

	Total # of commands sent	# of modified commands	# of *rejected* modified commands	# of *accepted* modified commands	# of incorrect results
Exp. 1	99396	9036	8864 (98.1%)	172 (1.9%)	0
Exp. 2	11000	9000	8907 (98.7%)	93 (1.3%)	0

Table 1. Experimental results

Again, the card has a correct behavior, but continues to accept a small percentage of modified commands as correct. However, an analysis of the reasons of this strange behavior is beyond the scope of the present paper. With this last experiment, we achieved our aims, showing that the card under analysis is robust with respect to any altered command sequence.

Observe that the full interaction between the model checker and the smart card, related to a single command, requires on the average 0.8 seconds. Thus, the two experiments took about 22 hours and 2 hours, respectively. However, to accelerate more complex verification tasks, we plan to make use of distributed verification architectures, which have been already developed for the Murphi verifier [16].

7 Conclusions

In this work, we have shown an integrated environment to perform the verification of smart card based signature devices, with respect to models of correct behavior which can be much more detailed than those considered in the Common Criteria. Since this

verification task goes beyond simple black box testing we integrated a model checker in the verification environment.

We tested a commercial signature device, systematically targeted with wrong commands while executing the signature of a fixed document. While the card has shown a correct behavior, with respect to a reasonable model, even in this simple experimentation it has been possible to point out some anomalous behavior such as the acceptance of wrong commands.

Many other verifications can be performed on smart cards using our integrated framework, addressing aspects not covered by the common criteria: for instance, currently we do not know if and how a card microprocessor would react to *concurrent* signing sessions.

Beyond such specific verifications, our general objective is to set up a whole family of behavioral models - defining in a (hopefully) complete way the correct behavior of a card-based digital signature device, as well as a verification environment able to prove or, at least to give strong evidence, that the system is compliant with respect to all models.

We think that if this task is accomplished, this will be a very relevant step towards the solution of the interoperability problem, as the cards so certified would perform well even in the most challenging situations.

References

1. CEN: TC224 WG15.
2. D.M. 8 novembre 2007: S.O. n. 229 della G.U. 261 del 9/11/2007 Regole tecniche della Carta d'identita elettronica (Technical Rules for the Eletronic Identity Card).
3. Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and general model 11, version 3.1, revision 1, ccmb-2006-09-001, september 2006. 5. common criteria for information technology security evaluation - part 2: Security functional requirements12, version 3.1, revision 1, ccmb-2006-09-002, september 2006. 6. common criteria for information technology security evaluation - part 3: Security assurance requirements13, version 3.1, revision 1, ccmb-2006-09-003, september 2006.
4. CEN WORKSHOP AGREEMENT: Cwa 14169, march 2004.
5. Leroy, X.: Computer security from a programming language and static analysis perspective. In: ESOP. (2003) 1–9.
6. Toll, D.C., Weber, S., Karger, P.A., Palmer, E.R., McIntosh, S.K.: Tooling in Support of Common Criteria Evaluation of a High Assurance Operating System. IBM Thomas J. Watson Research Center Report (2008)
7. Chapman, R.: Correctness by construction: a manifesto for high integrity software. In: SCS '05: Proceedings of the 10th Australian workshop on Safety critical systems and software, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2006) 43–46.
8. Murphi Web Page: `http://verify.stanford.edu/dill/murphi.html`
9. CMurphi Web Page: `http://www.di.univaq.it/gdellape/murphi/cmurphi.php`
10. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. Model Checking Software (2005) 2–23.
11. Michael, C., Radosevich, W.: Black box security testing tools. cigital, 2005.
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)

13. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors, IEEE Computer Society (1992) 522–525.
14. Della Penna, G., Intrigila, B., Melatti, I., Tronci, E., Venturini Zilli, M.: Integrating ram and disk based verification within the murφ verifier. In Geist, D., Tronci, E., eds.: Proceedings of Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003. Volume 2860 of Lecture Notes in Computer Science., Springer (2003) 277–282.
15. Della Penna, G., Intrigila, B., Melatti, I., Minichino, M., Ciancamerla, E., Parisse, A., Tronci, E., Venturini Zilli, M.: Automatic verification of a turbogas control system with the murφ verifier. In Maler, O., Pnueli, A., eds.: Proceedings of Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003. Volume 2623 of Lecture Notes in Computer Science., Springer (2003) 141–155.
16. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. In: SPIN. (2006) 108–125.