A Formal Security Model of a Smart Card Web Server

Pierre Neron[1] and Quang-Huy Nguyen[2]

[1] Ecole Polytechnique
92128 Palaiseau, France
[2] Trusted Labs
5, rue du Bailliage, 78000 Versailles, France

Abstract. Smart card Web server provides a modern interface between smart cards and the external world. It is of paramount importance that this new software component does not jeopardize the security of the smart card. This paper presents a formal model of the smart card Web server specification and the proof of its security properties. The formalization enables a thoughtful analysis of the specification that has revealed several ambiguities and potentially dangerous behaviors. Our formal model is built using a modular approach upon a model of Java Card and Global Platform. By proving the security properties, we show that the smart card Web server preserves the security policy of the overall model. In other words, this component introduces no illegal access to the card resources (*i.e.,* file system and applications). Furthermore, the smart card Web server provides a means for securely managing the card contents (*i.e.,* resources update).

1 Introduction

Since the beginning of the smart card era, the I/O interface is defined by the ISO-7816 standard [10] in terms of APDU (Application Protocol Data Unit) commands and responses. For example, to access to a binary file, the card reader sends a `SELECT FILE` command to select the file, then a `GET BINARY` command to retrieve its contents. For the next generation of multimedia SIM cards that may embed up to one gigabyte of data, this interface becomes outdated: because the multimedia data are heterogeneous and stored in complex multi-leveled file systems, the APDU commands do not provide an efficient access method. Furthermore, the ISO-7816 standard requires ad-hoc software on the handset to communicate with the SIM card. On the other hand, most handsets own a Web browser to access to the MNO (Mobile Network Operator) services and it is tempting to use the HTTP protocol for interfacing with the SIM card. In this interface, the SIM card embeds a Web server and the handset accesses to the card resources via a Web browser.

The so-called smart card Web server (SCWS) provides a modern interface and dramatically simplifies the access to multimedia services. Standardized by Open Mobile Alliance (OMA)[12], the SCWS also allows the MNOs to remotely

administer their clients via an OTA (over-the-air) infrastructure. Recently, this remote administration capacity has been leveraged by the Global Platform consortium [8] to a content management method for multi-actor multi-application smart cards. For Java Card framework, the European Telecommunications Standards Institute (ETSI) also specifies additional APIs to transform a Java Card applet into a servlet (*i.e.,* a Web application)[6]. In other words, SCWS is a cross-industry specification based on existing infrastructures and it is worth to investigate its overall consistency before any implementation.

On the other hand, the Web server also exposes the SIM card to numerous well-known threats and it is of paramount importance that it does not compromise the security services provided by the card. In particular, it is essential to ensure that no illegal access to card resources can be done through the SCWS.

In this paper, we present a formalization the SCWS specifications [12, 8, 6, 17, 16] and a formal proof of the security properties related to the card resources access. The SCWS is formalized as a state machine that receives a HTTP request, interacts with the other card components before sending back the corresponding HTTP response. All formal models and proofs have been developed in Coq [15], a proof assistant based on higher-order type theories. This choice was firstly motivated by the safety of Coq that has well-studied mathematical foundation and a robust implementation: all proofs are re-checked by a (tiny) kernel that is the only Coq trusted reasoning base. Secondly, the expressive power of the logic underlying Coq deals more efficiently with the (universally) quantified security properties. Furthermore, this work benefits from the Coq libraries built for the certification of the Java Card platform in a previous work [5] and hence, smoothly handles the interaction between different card components (*e.g.,* SCWS, Java Card and Global Platform). The reuse of an existing formal infrastructure also reduces the modeling and proving workload.

The rest of this paper is organized as follows. Section 2 summarizes the different specifications related to the SCWS and discussed its expected security properties. The Coq model of the SCWS is described in Section 3. In Section 4, we present the Coq formalization and the proof of the security properties. We discuss the related work and give some concluding remarks in Sections 5 and 6.

2 Smart Card Web Server

This section provides an overview of the SCWS functions as specified in [12, 8, 6, 17], then discusses several ambiguities and dangerous behaviors discovered while formally analyzing these specifications. An analysis of the SCWS security policy is also presented.

Figure 1 shows the integration of the SCWS in a multi-actor and multi-application Java smart card (*i.e.,* a smart card that supports both Java Card and Global Platform). The SCWS includes three components:

- a HTTP server
- a repository containing the SCWS data
- an administration task manager that updates the SCWS repository

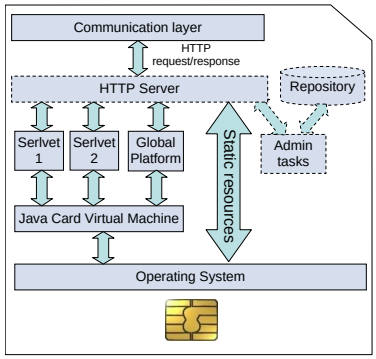

Fig. 1. SCWS inside a Java smart card

The external entity (*e.g.,* a Web browser or an OTA infrastructure) communicates with the card using the HTTP requests and responses. After being relayed by the communication layer, the HTTP request is sent to the HTTP server. Note that the multi-request management is ensured by the communication layer: a HTTP server (instance) is supposed to receive and manage one request per session. A SCWS session starts with the reception of the request and finishes by the emission of the response. The HTTP server dispatches the request to its destination indicated by the URL included in the header of the request:

- If this URL points to a servlet (*i.e.,* an applet registered to the SCWS), then the request is forwarded to this servlet.
- If this URL points to a static resource (*i.e.,* a file in the file system), then this file is accessed following the HTTP method (`GET`, `PUT`, etc) of the request. The access to the file system is done through the OS low-level services.
- If the URL points to the administration task manager, then this is an administration command for updating the SCWS repository.
- If the URL points to the Global Platform component (also known as the Card Manager), then the request contains a command that updates the card contents (*i.e.,* update the servlets).

Interface between SCWS *and servlets.* On a Java Card platform, a servlet is an applet that was registered to the SCWS. A servlet needs to implement a specific Java interface for each HTTP method. For example, the `doPost()` interface must be implemented to handle the POST requests. The servlet registration and Java interfaces are defined by ETSI in a dedicated API [6].

The interface between the SCWS and the servlet is done by method invocation. For example, for a `POST` request, the SCWS identifies the servlet, and invokes the `doPost()` method of this servlet. The HTTP request is passed to this method as a global variable. The method constructs the HTTP response

that is also a global variable. Finally, the SCWS envelops the HTTP response and sends it back to the communication layer.

This interface corresponds to the "normal" servlets that receive a HTTP request and returns a HTTP response. The SCWS also manage the "interception" servlets that only collect information from the header of the HTTP requests. The interception servlets do not return any HTTP response.

Access control. According to [12], the HTTP access control relies on the *protection sets*. A protection set contains a list of users. If a URL subtree is mapped to a protection set, then access to the resources inside this subtree is reserved to the users defined in that protection set.

The user authentication may use different protocols:

- HTTP Basic or Digest authentication as defined in [16], which is a simple authentication with username and password contained in the request's header
- HTTPS protocol which uses cryptographic procedure to authenticate users
- ADMIN protocol *i.e.,* the current user is the remote administration server

Building the HTTP response. In successful cases, the result is put in the body of the response and the status `0x200` is set in the header of this response. Otherwise, an error status is sent back in the header. Several categories of status are defined by the HTTP protocol *e.g.,* `0x2xx` for successful cases, `0x4xx` for client errors, `0x5xx` for internal errors while processing the request.

2.1 Specification analysis and recommendations

The formal analysis of the SCWS specification has revealed several ambiguities and dangerous behaviors that we discuss here. Furthermore, several recommendations are also provided to overcome these issues.

1. **URL sharing**: sharing a URL between servlets and static resources is not forbidden in the specification. In some cases, URL sharing does not generate any conflict, *e.g.,* between a normal servlet and an interception servlet. In other cases, *e.g.,* between a servlet and a file, the data returned by the SCWS is not precisely defined. In order to ensure the consistency, URL sharing is not allowed in the formal model.
2. **Multi-role applet**: can an applet be registered as both interception servlet and normal servlet? Multi-role applet is not forbidden by the specification but an interception servlet can only access to the header while a normal servlet has access to all the HTTP request. In other words, there is an inconsistency on the access rights of the applet. The formal model clearly separates interception servlets from normal ones: multi-role applet is not allowed.
3. SCWS/Java Card **consistency**: can a servlet be uninstalled or updated during a servlet invocation (that is handled by Java Card)? In principle, during the execution of a method, the list of servlets may be updated. However, a servlet cannot be uninstalled or updated during its invocation.

4. SCWS/Global Platform **consistency**: the list of registered servlets and the Global Platform registry table that manages the list of (on-card) applets shall be consistent. Because a servlet is an applet that was registered to the SCWS, any modification on the list of applets (kept in Global Platform registry table) shall be synchronized with the SCWS. Furthermore, the modification rights of these lists shall be reserved to the same group of users.
5. **Servlet collaboration**: if several servlets are invokable on a request (for example, one URL is covered by two different servlets), a priority order is defined between these servlets. Usually, the servlet that is mapped to the closer URL has greater priority. The SCWS invokes the servlets following this order and if a servlet is invoked but refuse to handle the request, then the next servlet will be invoked. However, the refusing servlet is still able to access to the HTTP request. In other words, if a refusing servlet is mapped to the directory `/a/b/c`, then it can learn about the activities of the other servlets that are mapped to the directories `/a` and `/a/b`. Hence, the URL distribution to servlets (done by the SCWS administrator), shall be carefully done to avoid data leaking between them. Another solution would be to restrict the servlet collaboration by only allowing the servlet mapping to the exact URL to access to the HTTP request.
6. **Unsafe default configuration**: if no protection set is defined, then any resource is accessible. This mechanism is advocated in the HTTP protocol to avoid unnecessary authentications to the Web servers. However, on a smart card, this behavior seems to be dangerous. It is recommended to map an empty protection set to the root URL and hence, force an authentication on any access.
7. **Unsafe Fail**: in an unsuccessful operation, the HTTP response is only required to contain an error code that is different from `0x200`. This is necessary to inform the handset about the failed operation. However, a confidential information may still be leaked through the other components of the response. Hence, it is recommended that the HTTP response does not contain other data than the error code.

While the three first points correspond to the imprecision of the specification, the other issues are directly related to the security of the SCWS. The formal model described in Section 3 takes into account the above security recommendations to deal with these issues.

2.2 Security policy

The principal SCWS security policy consists in preventing illegal access to the card resources. This security policy can be decomposed into several sub-policies as follows:

(1) URL separation: Static resources, interception servlets and normal servlets shall be separated in terms of mapped URL.
(2) No illegal access to static resources: an access cannot target

  - the un-mapped static resources,
  - the static resources not mapped to the request's URL, and
  - the unauthorized static resources.

(3) No illegal invocation of smart card applications: an invocation cannot target
  - the unregistered servlets,
  - the unmapped servlet,
  - the servlets not mapped to the request's URL, and
  - the unauthorized servlets.

(4) The smart card data outside the scope of the SCWS cannot be accessed *i.e.,* any data in a HTTP response shall belong to a SCWS-managed resource.

(5) Safe Fail: in an unsuccessful operation, the HTTP response does not contain other data than the error code.

(6) Secure card content management: the card contents (*i.e.,* static resources, applets and repository) can only be updated by an"Admin" user.

These sub-policies are then decomposed into simpler security properties in order to ease the Coq formalization.

Property 1. (URL separation) *All normal servlet, interception servlet and static resources are mapped to separate URLs.*

Property 2. (Invalid static resource) *If a static resource is not mapped to any URL, then its data cannot be sent out by a HTTP response.*

Property 3. (Unauthorized access to static resource) *If a static resource is protected by a protection set, then any HTTP request to a static resource will fail if no username/password were provided or the provided username/password are not correct.*

Property 4. (Unregistered application) *An unregistered servlet cannot be invoked by any HTTP request.*

Property 5. (Unmapped application) *A registered servlet that is not mapped to any URL cannot be invoked by any HTTP request.*

Property 6. (Unrelated application*) A servlet that is not mapped to the URL of the HTTP request and any of its ancestors cannot be invoked by this request.*

Property 7. (Unauthorized access to application) *A servlet that is mapped to the URL of the HTTP request cannot be invoked if this request is not authorized (no username/password were provided or the provided username/password are not correct).*

Property 8. (Access to administrative tasks) *Only an "Admin" user can perform the administrative tasks on the* SCWS.

Property 9. (Access to GlobalPlatform operations) *Only an "Admin" user can perform the* Global Platform *tasks.*

Property 10. (Safe fail) *If the status code of a HTTP response is not* `0x200`*, then this response contains no data.*

It is straightforward to see that, the policy (1) is fulfilled by Property 1; the policy (2) is fulfilled by Properties 1, 2, and 3; the policy (3) is fulfilled by Properties 1, 4, 5, 6 and 7; the policy (4) is fulfilled by Properties 1, 2; the policy (5) is fulfilled by Property 10; the policy (6) is fulfilled by Properties 8 and 9.

3 A Coq Model of the Smart Card Web Server

The SCWS is formalized as a state machine. The SCWS state formalizes the repository while a SCWS transition formalizes the modification caused by a HTTP request on a state.

3.1 SCWS state

The state defines the global components of the SCWS as the following record:
SCWS_*State* $\triangleq$ {
registered_servlets : *registered_servlet_table*;
servlet_mapping : *url_servlet_table*;
interception_servlets : *url_servlet_intercept_table*;
users_id : *user_table*;
ps_defined : *ps_list*;
ps_table : *path_ps_table*;
listen_http : *http_state*;
auth_status : *scws_status*;
curr_request : *option http_request*;
curr_response : *http_response*
}
where the components are formalized as record fields associated to their types:

- *registered_servlets* is the table of all servlets (*i.e.,* applets registered to the SCWS).
- *servlet_mapping* and *interception_servlets* are respectively two tables mapping the URLs to the normal servlets and the interception servlets.
- *users_id* is the table of all registered users in the SCWS. This table maps each user's identifier to its password.
- *ps_defined* is the table of all registered protection sets. This table maps each protection set to its parameters (protocol, list of authorized users, etc).
- *ps_table* is the table mapping each protected URL to its protection set.
- *listen_http* indicates if the SCWS is currently ready for receiving a HTTP request.
- *auth_status* indicates whether the current user is an "Admin" user (in this case *auth_status* is "ADMIN").

- *curr_request* is the HTTP request being processed by the SCWS: the request is defined as an optional type to handle "no request" error.
- *curr_response* is the HTTP response being processed by the SCWS: the SCWS always returns a (error or success) response.

All these elements define the current state of the SCWS and determine if a static resource is accessible or if a servlet is invokable by the current request.

URL tree. The resources managed by the SCWS are addressed by their URLs. In the model, the URL path name includes the directory names stored in the reverse order to speed up the recursive search in the ancestors of a name. For example, the path name `/scws/appl/epurse` is stored by the list $epurse \rightarrow appl \rightarrow scws$. There are three constructors to generate a set of URLs from a path name as follows:

$$\begin{aligned} path\_url \triangleq\ & exact\_url : path\_name \rightarrow path\_url \\ | \ & directory\_url : path\_name \rightarrow path\_url \\ | \ & *\_url\_from : path\_name \rightarrow path\_url. \end{aligned}$$

For example, `/scws/appl/epurse` is an exact URL, `/scws/appl/` is a URL directory and `/scws/appl/*` is a URL subtree.

SCWS *file system.* The static resources are effectively stored in the smart card file system. However, the SCWS only manages part of the smart card file system that is in its scope. Without loss of generality, a SCWS file system is represented by a table mapping the URLs to the associated files.

3.2 SCWS transition

The transition formalizes the modification of the state caused by the process of a HTTP request. Each transition is defined by the relations between the input and the output states. The card components that can be modified by a transition are:

- the SCWS state *e.g.,* due to an administrative tasks
- the SCWS file system *e.g.,* due to `PUT` and `DELETE` requests
- the JCVM (Java Card virtual machine) state[3] *e.g.,* due to the invocation of a servlet by the SCWS

The process of a HTTP request is done in three steps. First, the authentication checks if the user has the sufficient right to process the request. Then, a servlet is resolved and invoked if necessary. Finally, the request is processed *w.r.t.* the contents of the request (see Section 2).

[3] A record that contains the global components of the JCVM.

Authentication. The authentication process is defined according to [12]:

- `PUT` and `DELETE` requests require the current user to be an "Admin" user
- `TRACE` request that returns the routing information towards the server (*e.g.,* the list of proxies), does not require any authentication
- for any other request, the authentication process relies on the defined protection sets. We need to find if the requested URL or its ancestors are mapped to any protection set:
  - If no protection set is found, then the URL is in free access.
  - If the found protection set requires the ADMIN protocol, then the current user must be an "Admin" user.
  - If the found protection set requires the HTTP protocol with no realm, then the URL is in free access.
  - If the found protection set requires the HTTP protocol with the realm "Basic authentication", then the user and password must be present in the request and must correspond to a valid user defined in the protection set (and $user\_id$ table).

Servlet resolution and invocation The SCWS first attempts to find a servlet that is exactly mapped to the requested URL using the `servlet_mapping` table. If such a servlet is not found, then the search is done recursively in the ancestors of the URL.

Once a servlet is resolved, the SCWS invokes the servlet's method that corresponds to the request (*e.g.,* `doPost()`, `doGet()`). This invocation is done by the method invocation mechanism provided by the Java Card model. The execution of the servlet's method may modify the JCVM state but this modification is managed by the Java Card model. The SCWS only manages the modification caused by this method on the current HTTP response (formalized by $curr\_response$).

If the resolved servlet's method produces no effect on the current HTTP response, then this servlet refuses the current request and the search is continued in the URL tree to locate the next candidate.

Request processing. The actions to be done by the SCWS depend on the method of the request (*i.e.,* `GET, POST, PUT, DELETE, HEAD, OPTION, TRACE` or `CONNECT`) and the requested URL (see Section 2):

- if the URL points to the administration task manager, then the SCWS state is modified according to the definitions in [12].
- if the URL points to the Global Platform component (*i.e.,* the Card Manager), the appropriate method of the Global Platform model is invoked.
- if this is a POST request that accesses to a servlet, then this servlet is resolved and invoked using the requested URL as described above.
- otherwise, the SCWS checks if the requested URL is mapped to a valid static resource, and returns it (or an eventual error).

The SCWS transition is defined by the following relation:

$$transition(A, scws\_st_{in}, jcvm\_st_{in}, fs_{in}, scws\_st_{out}, jcvm\_st_{out}, fs_{out})$$

where A is the action that causes the transition, $scws\_st_{in}$, $jcvm\_st_{in}$, fs_{in} respectively represent the input states of the SCWS, the JCVM and the file system while $scws\_st_{out}$, $jcvm\_st_{out}$, fs_{out} respectively represent the output states of the SCWS, the JCVM and the file system.

Example 1. The model of the `GET` request in the case where no servlet provides a response is as follows:

$$\begin{aligned}
&transition(GET_request, scws_st_{in}, jcvm_st_{in}, fs_{in},\\
&\quad scws_st_{out}, jcvm_st_{out}, fs_{out}) \triangleq \exists\ request\ \exists scws_st_{aux}.\\
&(0)\ fs_{in} = fs_{out} \wedge\\
&(1)\ curr_request(scws_st_{in}) = Some(request) \wedge\\
&(2)\ servlet_resolution_and_invocation(doGet(), request, scws_st_{in}, jcvm_st_{in},\\
&\quad fs_{in}, scws_st_{aux}, jcvm_st_{out}, fs_{out}) = NO_ERROR \wedge\\
&(3)\ curr_response(scws_st_{aux}) = curr_response(scws_st_in) \wedge\\
&(4)\ get_file(fs_{in}, scws_st_{aux}, scws_st_{out})
\end{aligned}$$

(0) means that the file system is not modified by this transition.
(1) means the current SCWS state ($scws\_st_{in}$) contains some request to be processed.
(2) means the servlet invocation produces the output state of the JCVM ($jcvm\_st_{out}$), an temporary SCWS state ($scws\_st_{aux}$) and no error.
(3) means the servlet invocation has no effect on the HTTP response (because the response component of $scws\_st_{aux}$ is that of $scws\_st_{in}$). In other words, all servlets refuse the HTTP request. In this case, the request is forwarded to the file system (static resource).
(4) returns a file (pointed by the requested URL) or an error included in the output SCWS state ($scws\_st_{out}$).

4 Proof of the Security Properties

This section describes the formal statement and the proof of the security properties presented in Section 2.2. The security properties are formally expressed as Coq theorems in the following form:

$$\begin{aligned}
&\forall\ scws_st_{in}, scws_st_{out}, jcvm_st_{in}, jcvm_st_{out}, fs_{in}, fs_{out}.\\
&transition(A, scws_st_{in}, jcvm_st_{in}, fs_{in}, scws_st_{out}, jcvm_st_{out}, fs_{out}) \Rightarrow\\
&Pre(A, scws_st_{in}, jcvm_st_{in}, fs_{in}) \Rightarrow Post(A, scws_st_{out}, jcvm_st_{out}, fs_{out})
\end{aligned}$$

where Pre states the conditions on the input states of the transition and $Post$ states the property of the output states.

Theorem 1. *(No map on static resource) Mapping a servlet to a URL already mapped to a static resource, returns error status code and this mapping will not be registered.*

Theorem 2. *(Admin access required) Access to a URL protected by a protection that requires ADMIN authentication is only allowed if the authentication status of the input state is ADMIN.*

Theorem 3. *(Invalid Static Resource) If the* SCWS *provides a response containing some data (i.e., some file), then this file was already in the file system before the request processing and the status code of the response is* $0x200$ *(success).*

Theorem 4. *(Invalid Authentication) If the requested URL is protected by a protection set (or inherits a protection set from its ancestors) and if the authentication failed i.e., (i) no authentication parameter in the request or, (ii) wrong password or, (iii) the user is not defined in the protection set, then the response is an error status code.*

4.1 External observation of servlet invocation

In order to formalize the properties requiring that a servlet is not invoked by the SCWS, we use the external observation approach. A servlet is not invoked by the SCWS if the final state of any SCWS transition is independent of the code of this servlet. In other words, the servlet's behavior has no effect on the SCWS transitions. This approach simplifies the formalization of these properties and keeps it independent of the Java Card model.

Theorem 5. *(Unregistered servlet) Unregistered servlets have no effect on the final state of a* SCWS *transition.*

To this end we suppose the determinism of the JCVM transitions and of the file system transitions (*i.e.,* those transitions are functions). The JCVM transition describes the modification of the JCVM due to the invocation of a servlet by the SCWS while the file system transition describes the modifications of the file system due to static resource access requests (*e.g.,* `PUT` or `GET`). Intuitively, all JCVM and file system operations (bytecode execution and file access) are deterministic. Those are two properties of the JCVM and file system model that are not in the scope of the SCWS.

The formalization is based on two actions $call\_servlet_1$ and $call\_servlet_2$ that represent the effects of any two different codes of the unregistered servlet

(these two actions having an identical effect on other servlets).

$$\begin{aligned}
&\forall\ S, scws_st_{in}, scws_st_{out1}, scws_st_{out2}, jcvm_st_{in}, jcvm_st_{out1}, jcvm_st_{out2} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad fs_{in}, fs_{out1}, fs_{out2}. \\
&\quad transition(call_servlet_1,\ scws_st_{in}, jcvm_st_{in}, fs_{in}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad scws_st_{out1}, jcvm_st_{out1}, fs_{out1}) \Rightarrow \\
&\quad transition(call_servlet_2, scws_st_{in}, jcvm_st_{in}, fs_{in}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad scws_st_{out2}, jcvm_st_{out2}, fs_{out2}) \Rightarrow \\
&\quad \forall\ serv.(S \neq serv \Rightarrow call_servlet_1(serv) = call_servlet_2(serv)) \Rightarrow \\
&\quad unregistered(scws_st_{in}, S) \quad \Rightarrow \quad scws_st_{out1} \quad = \quad scws_st_{out2}
\end{aligned}$$

Theorem 6. *(Unmapped servlet) Unmapped servlets have no effect on the final state of a* SCWS *transition.*

This theorem is formalized similarly as Theorem 5 using the hypothesis that the unmapped servlet is not mapped to the requested URL and any of its ancestors.

4.2 Proof

Notice first that these Coq theorems are sufficient to verify the security properties presented in Section 2.2. Indeed:

- Property 1 is verified by Theorem 1; Property 2 is verified by Theorem 3; Property 3 is verified by Theorem 4; Property 4 is verified by Theorem 5.
- Property 5 is verified by Theorem 6; Property 6 is verified by Theorem 6; Property 7 is verified by Theorem 4.
- Properties 8 and 9 is verified by Theorem 2 under the hypothesis that the URLs used in the administration commands and the Global Platform commands are only accessible to an "Admin" user. This hypothesis corresponds to the security policy required by the SCWS administration and the Global Platform card content management [8].
- Property 10 is the corollary of Theorem 3.

The proof of the Coq theorems is interactive. According to the data structures used in the model, a relevant proof scheme (*e.g.,* case analysis, induction) is used. Most of the proofs of the different properties relies on a case analysis of the different transitions provided by the SCWS and modeled through the *transition* predicate. Therefore, for each of this case there are two possibilities :

- The "nominal" case corresponds to the proof hypotheses (*e.g.,* wrong password), thus the *transition* relation provides us the corresponding conclusion (*e.g.,* sending an error code) is in general simple.
- The specific cases contradict the proof hypotheses, and we have to construct the right premise (from the current context)that allows us to prove the contradiction. This construction can be costly due to the size and the complexity

of the context. If a recursive data structure or action is involved in the property (*e.g.,* the URLs or the servlet resolution), then the proof requires an induction on this structure or action.

Proof of the external observation properties (Theorem 5 and 6) also relies on case analysis of the two separate transitions (that respectively use $call\_servlet\_1$ and $call\_servlet\_2$):

- if the transitions are identical and the request does not invoke the (unregistered or unmapped) servlet S, then we can use the determinism hypotheses to conclude that both output SCWS states are also identical
- if the transitions are identical and the request invokes S, then one can prove that because S is either unmapped or unregistered, the SCWS does not invoke it and thus S has no effect on the output state
- otherwise, the transitions being different, the model is proved to be deterministic by showing that the intersection of the premises in two transitions is empty.

Dedicated tactics have been used to factorize the proof and reduce the maintenance workload. However, the proof still requires a lots of interactions to allow the Coq's kernel to re-check the all proof at the end (to increase the tool's trustworthiness). In other words, user-interactions are somewhat the cost to pay for getting a higher level of trust.

5 Related Work

The SCWS is a pretty new software component and to our knowledge, no formal analysis has been done on it. However, formal techniques have been used by numerous researchers to analyze the security of the Web-related systems. The network protocols are the preferred targets for formal methods (see [11, 13, 4, 3]). The SCWS uses the well-known HTTP and HTTPS protocols that have been intensively investigated and hence, are not in the scope of our model.

In [1], the authors formalize part of a SMTP mail server in Coq. This model only covers the mail receiving process. The authors build the model following a Java implementation rather than the SMTP specification. In contrast, we aim at preserving the generality of the SCWS specifications as much as possible. If a choice is required between several specifications, the generality is considered as the first criterion. Our objective is to ensure that no new smart card backdoor is introduced by the SCWS and the security services provided by the smart card are preserved. We do not focus on a specific implementation as done in the source-code verification approach. Source-code verification is trendy because it may detect bugs on a real implementation. However, in the current state of the art, source-code verification is usually limited by the size and the complexity of the code.

Model checking is also widely used to formalize and verify the properties of the Web applications (or services). The drawback of the model checking approach

is that an abstraction is usually required on the properties in order to avoid state explosion. Finding a correct abstraction is not always possible for any property. The paper [9] describes an attempt to apply SPIN model checker to the verification of the Web applications. The authors formalize the properties in a communicating finite automate and use SPIN to verify these properties. Also using SPIN, [7] presents a tool built upon it to formally analyze the Web services. In [14], a recovery framework of the Web services is modeled and checked. The Web applications (or servlets) are not in the scope of our work. Actually, we focus on the interactions between the servlets and the external world through the SCWS. Hence, in some sense, this work complements the verification of Web applications.

The difficulty in designing secure Web systems is partly due to the lack of a formal foundation: [2] is a recent attempt to bridge this gap. That paper presents several attack models on the Web system and evaluates some well-known counter-measures *w.r.t.* these models. This is a promising approach to formally analyze the robustness of the Web systems.

6 Concluding Remarks

In this paper, we presented a formal security model of the SCWS. The formalization identified some ambiguities as well as dangerous behaviors of the specification (see Section 2.1). This feedback is useful for the standard institutions such as `Global Platform` consortium, OMA and ETSI in order to improve their specifications. On the other hand, several security properties of the SCWS were proved on the model. These security properties ensure that the SCWS preserves the overall security of the Java Card/Global Platform model. These security policies are also required by different protection profiles[4] of SCWS-embedded products. Hence, this work can be used to fulfill the ADV_SPM.1 evaluation task[5] that is the only formal method related task of the Common Criteria **EAL6** level.

It is worth to highlight the modular approach advocated by this work. The reuse of the model (and associated proof) bricks provided by the existing Java Card/Global Platform baseline model[6] reduces the overall workload to roughly two man-months for a Formal Methods expert having general knowledge on smart (Java) cards. Three thousand lines of model and ten thousand lines of proof have been developed in this project. The SCWS model also extends the baseline model by new bricks such as "HTTP protocol" and "servlet management". Without the existing bricks, building the Coq model and proof for a new software component from the scratch may not comply with the industrial constraints. Indeed, the

[4] A protection profile defines the set of security requirements on an IT product for a Common Criteria security evaluation.

[5] This task requires to build a formal security model of the product and to prove the evaluation security objectives on this model.

[6] More than 100 000 lines of Coq model have been developed for fulfilling the highest requirements on the ADV (Application DeVelopment) assurance class [5].

formal analysis is best done between the specification/design freeze and the implementation. Otherwise, the formal analysis does not generate sufficient added value on the final product.

Further work consists in refining the security model in more concrete representations (*e.g.,* "Functional Specification" and "Design" in Common Criteria scheme) to get a complete formal chain from the specification to the implementation and hence, fulfills the Common Criteria highest requirements on the ADV assurance class that ensures the correctness of the product. Again the development of the formal chain will be facilitated by the existing bricks provided by [5]. Note that in order to get a full **EAL7** certificate, the other assurance classes are still to be addressed, in particular, the ATE (Application TEsting) assurance class and the AVA (Application Vulnerability Analysis) assurance class. These classes are not really related to formal methods: in the current state of the art, formal models and proofs are mainly used to ensure the correctness (the robustness of the product being ensured by the ATE and AVA assurance classes).

Finally, it is worth to mention that in the latest Java Card specification (version 3.0 - connected edition), the Web server is part of the virtual machine that also includes almost all Java features such as garbage collection and multi-threading. Java Card 3.0 platform requires significantly more computing resources than the previous version and is still yet to be accepted by the market. On the contrary, the SCWS is a pragmatic implementation of a HTTP-based I/O for the currently deployed smart cards. A security model of the Java Card 3.0 Web server can also be built at a reasonable cost using the model bricks provided by this work.

Acknowledgment. We thank the anonymous reviewers for their precious comments on the previous version of this paper.

References

1. R. Affeldt and N. Kobayashi. Formalization and verification of a mail server in coq. In M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *ISSS*, volume 2609 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2003.
2. D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *Proc. of the 23rd IEEE Computer Security Foundations Symposium*, pages 290–304. IEEE Computer Society, 2010.
3. A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proc. of 17th USENIX Security Symposium*, 2008.
4. K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of ws-security protocols. In M. Bravetti and G. Zavattaro, editors, *Proc. of 3rd WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, 2006.
5. B. Chetali and Q. H. Nguyen. Industrial use of formal methods for a high-level security evaluation. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *Proc.*

of 15th Intl. Symposium on Formal Methods, volume 5014 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 2008.

6. European Telecommunications Standards Institute. *Smart Cards; Application invocation Application Programming Interface (API) by a UICC webserver for Java Card platform (Release 7)*, 2008. Ref: ETSI TS 102 588. Available at `http://pda.etsi.org/pda/queryform.asp`.
7. X. Fu, T. Bultan, and J. Su. Wsat: A tool for formal analysis of web services. In R. Alur and D. Peled, editors, *Proc. of 16th Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 510–514. Springer-Verlag, 2004. Tool available at `http://www.cs.ucsb.edu/~su/WSAT/`.
8. Global Platform. *GlobalPlatform Card Specification 2.2 - Remote Application Management over HTTP - Amendment B, version 0.5*, 2008. Available at `http://www.globalplatform.org/specificationscard.asp`.
9. M. Haydar, A. Petrenko, and H. Sahraoui. Formal verification of web applications modeled by communicating automata. In de Frutos-Escrig David and N. Manuel, editors, *Proc. of FORTE 2004*, volume 3235 of *Lecture Notes in Computer Science*, pages 115–132. Springer-Verlag, 2004.
10. International Organization for Standardization. *Identification cards – Integrated circuit(s) cards with contacts. Part 1-11*, 1987-2007. Available at `http://www.iso.org/`.
11. J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of ssl 3.0. In *Proc. of 7th USENIX Security Symposium*, pages 201–216, 1998.
12. Open Mobile Alliance. *Smartcard Web Server V1.1*, 2009. Available at `http://www.openmobilealliance.org/technical/release_program/SCWS_v1_1.aspx`.
13. A. W. Roscoe. Modelling and verifying key-exchange protocols using csp and fdr. In *Proc. of 8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Soc Press, 1995.
14. G. Shegalov and G. Weikum. Formal verification of web service interaction contracts. In *Proc. of the 2008 IEEE International Conference on Services Computing*, pages 525–528. IEEE Computer Society, 2008.
15. The Coq Development Team. *The Coq Proof Assistant*. `http://coq.inria.fr/`.
16. The Internet Society. *HTTP Authentication: Basic and Digest Access Authentication*, 1999. Available at `http://www.ietf.org/rfc/rfc2617.txt`.
17. The Internet Society. *Hypertext Transfer Protocol – HTTP/1.1*, 1999. Available at `http://www.ietf.org/rfc/rfc2616.txt`.