# A Bottom-Up Formal Verification Approach for Common Criteria Certification: Application to JavaCard Virtual Machine

Adel Djoudi, Martin Hána, Nikolai Kosmatov, Milan Kříženecký, Franck Ohayon
*Thales*
France & Czech Republic

Patricia Mouy, Arnaud Fontaine
*ANSSI*
France

David Féliot
*CEA-Leti*
France

## I. Introduction

The quality and security of critical software have become nowadays a major concern. The Common Criteria (CC) for Information Technology Security Evaluation [1] provide an international standard for computer security certification. Its highest assurance levels EAL6–EAL7 require a formal Security Policy Model (SPM) and an associated mathematical proof of security properties (i.e. confidentiality, integrity). Thales recently conducted a formal verification of a JavaCard platform module [14] in a novel EAL6 certification project of a smart card product. This certification project was evaluated by CEA-Leti (an evaluation center, or ITSEF) with the supervision of ANSSI (the French national cybersecurity agency, and the French certification body).

Historically, since the verification of real-life code was not feasible for large industrial projects, the certification usually followed a top-down approach, where a separate abstract model was used to verify the specified security properties, and then refined to the code. A classical approach of applying formal verification on a JavaCard platform consists in building a high-level formal model of target sub-modules. The need to bridge the gap between the formal model and the implementation and to provide stronger guarantees for the real-life code was reported by experts [2].

In our work, we adopt a novel bottom-up methodology relying on verification of the real-life code of a JavaCard Virtual Machine using the Frama-C verification platform [3]. We expressed all specified features and properties as annotations in a formal specification language, called ACSL (ANSI C Specification Language), inserted directly in the source code. We annotated over 7,000 lines of C code in ACSL, and over 50,000 proof goals were generated and formally proved by the tool. An earlier paper [4] focused on technical and scalability issues of the proof without addressing the certification methodology. In this paper, we focus on methodology aspects: we describe this bottom-up approach, discuss its benefits and challenges and compare it to previous top-down approaches.

## II. Common Criteria Certification Process

### A. Overview of Common Criteria Evaluation

The international standard ISO/IEC 15408—which defines the Common Criteria (CC)—is an international agreement on security evaluation of IT products. It contains a Common Evaluation Methodology (CEM) describing the general evaluation process from EAL1 up to EAL5. SOG-IS[1] is the European mutual recognition agreement that was concluded in 2010 and involves ten countries. For EAL6 and EAL7, formal methods have to be used, but the CEM does not detail how to use them to demonstrate the respect of the security properties. For this reason, the CEM is completed by an additional interpretation by the National Certification Body. Within the SOG-IS, only three countries mutually emit and recognize certificates up to EAL7: France, Germany and Netherlands. Each of them defines its own interpretation for EAL6 and EAL7 (AIS-34 in Germany and Note 12 [5] in France). Recognition agreements beyond the EU (between more than 30 countries) are defined in the Common Criteria Recognition Arrangement (CCRA).

[1]See https://www.sogis.eu/

A Common Criteria evaluation is initiated by the owner of the product to evaluate. The product owner (generally, an industrial) establishes a contract with an approved evaluation center (ITSEF) and registers the evaluation with the Certification Body. At the end of the evaluation, an Evaluation Technical Report (ETR) is produced by the ITSEF to be reviewed by the Certification Body.

The national certification bodies recognized to deliver EAL6–EAL7 certificates do not prescribe the use of any particular formal method or tool [6]. Up to now, only methods relying on high-level models in B and Coq were recognized by ANSSI in the Common Criteria context. A guidance document about the use of formal methods was published by ANSSI [7]. A more recent guidance written by ANSSI and INRIA research teams was published but with a focus on Coq [8] with a dedicated paper [9], which details the rationale of the guidelines and requirements from ANSSI. The respect of these guidelines has to be verified during the evaluation process.

The adoption of a new formal method or tool requires a pilot evaluation. It implies a tripartite work between the developer, the ITSEF and the certification body, and additional effort to inspect formal assurance.

For an evaluation, the developer has to supply the following evidence:

- the source code of the Formal Security Model (SPM),
- an explanatory document presenting a description of the model, a complete list of the associated hypotheses (explicit ones but also implicit ones due to modeling choices), their justification and their consistency,
- explicit links between the model, the security target and the security properties (further discussed below),
- a clear justification of the level of confidence for the method and tools used, and
- all the necessary information to allow the evaluator to reproduce the proof(s).

### *B. Security Specification*

A Common Criteria certification of a product helps the customer to determine whether the security of a product is sufficient to meet their needs and to ensure that the security properties are satisfied. For the product owner, the Common Criteria help to identify security issues, define security objectives, establish security requirements relying on a standardized catalog and then define a precise Target of Evaluation (TOE) that is usually only part of the entire IT product. The security specification plays an important role in the certification process.

The Common Criteria offer a catalog of Security Functional Requirements (SFRs) and Security Assurance Requirements (SARs) [1]. The SFRs define the TOE security characteristics. The SARs define confidence degree in the enforcement of the security objectives of the TOE. The assurance level is increased by increasing the scope, depth and rigor of the evaluation effort in six assurance classes: Security Target Evaluation (ASE), Development (ADV), Guidance (AGD), Life Cycle Support (ALC), Tests (ATE) and Vulnerability Assessment (AVA). At EAL6–EAL7 levels the ADV assurance requirement class mandates to build a formal security policy model (SPM) as the most rigorous way to identify and eliminate ambiguous, inconsistent, unenforceable or contradictory security policy elements [1]. For instance, the Common Criteria Action Element **ADV_SPM.1.1D** mandates the developer to identify the security policies that should be formally modeled. Note that the National Certification Body provides guidance for the interpretation of such statements [10] in order to satisfy the target security objectives. The identification of the security policy implies a list of SFRs to be formally modeled.

### *C. Application to JavaCard*

JavaCard system is a well-known security-critical product. Many JavaCard products have been subject to Common Criteria evaluation. The Security Target Evaluation class (ASE) enforces the definition of a Security Target (ST) for an identified product (e.g. a particular JavaCard product). A Security Target is an implementation-dependent statement of security needs. It states what is to be evaluated before the evaluation is performed (and thus helps to understand after the evaluation what was actually evaluated). The Security Target may claim conformance (strict or demonstrable) to a **Protection Profile** for a generic TOE type (such as a JavaCard system) [11]. A protection profile provides a standardized statement of Security Policies to be tailored according to the defined scope of evaluation in Security Targets. Over the years, the JavaCard System Protection Profile has established as one of the most recognized smartcard industry reference and is typically mandated in tenders or requested explicitly by customers. For instance, the Open Configuration Protection profile of JavaCard systems defines the **Firewall Security policy/aspect** [11] as follows:

"**#.FIREWALL**: The Firewall shall ensure controlled sharing of class instances, and isolation of their data and code between CAP files (that is, controlled execution contexts) as well as between CAP files and the JCRE context. An applet shall not read, write, compare a piece of data belonging to an applet that is not in the same context, or execute one of the methods of an applet in another context without its authorization."

The protection profile also instantiates this security aspect with a security objective (**O.Firewall**) and provides a rationale for the list of SFRs to be satisfied in order to meet this objective. The Security Target of a product may then instantiate the Common Criteria requirement **ADV_SPM.1.1D** as follows:

"**ADV_SPM.1.1D**: The developer shall provide a formal security policy model for the **Firewall Security Policy**."

and provide a rationale (according to Common Criteria security components ASE_OBJ and ASE_REQ) for the (sub)set of SFRs that are formally modeled in order to meet this requirement. In the sequel of this document, we consider the two following (simplified) SFRs for illustration purposes:

**SFR1** The Target of Evaluation Security Functions shall enforce the Firewall access control policy to provide restrictive default values for security attributes that are used to enforce the security policy. The objects' security attributes of the access control policy are created and initialized at the creation of the objects. Afterwards, these attributes are no longer mutable.

**SFR2** The Target of Evaluation Security Functions shall enforce the following rule to determine if an operation among controlled objects is allowed: the **Currently Active Context** may freely perform any memory access operation upon any object whose **Lifetime** attribute has value "Persistent" **only if** the **object's owning Context** attribute has the same value as the **Currently Active Context**.

Further details about the interpretation of these SFRs are provided in the following sections. In particular, we illustrate the mapping of these SFRs to our formal model in Section V.

## III. Bottom-Up Approach based on Deductive Verification

### A. JavaCard Virtual Machine

JavaCard applets are compiled to bytecode, which is executable in the JavaCard Virtual Machine (JCVM). A binary file (called CAP file), loadable to the platform, encapsulates mainly the bytecode together with class definitions. It may contain several Java packages and applets. A unique context is associated to each CAP file during loading to the card. Prior to loading a CAP file, a ByteCode Verifier (BCV) is run off-card to perform a static analysis (type-level abstract interpretation) of the applets [12]. This ensures that the code does not attempt to perform ill-typed operations that may bypass security protections ensured by the JCVM. Indeed, the virtual machine ensures bytecode interpretation and offers higher-level, more secure abstractions of data than the hardware processor, such as object references instead of memory addresses. Memory access operations on the stack, heap or code area are performed by the JCVM for each interpreted bytecode.

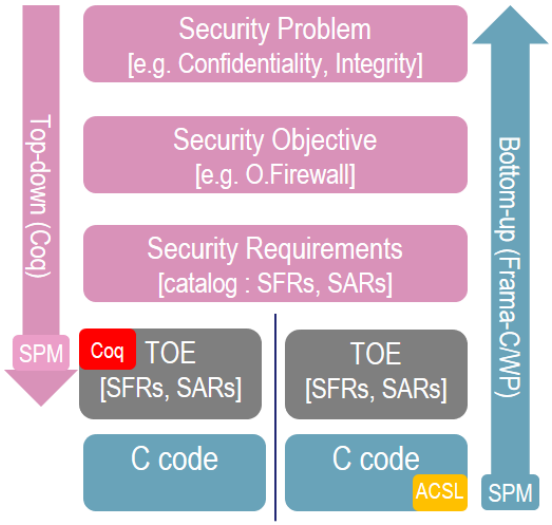


Fig. 1. Top-down/bottom-up approaches.

The JCVM firewall enforces runtime-protection of applet security properties against major security concerns: developer mistakes and design oversights allowing sensitive data "leakage" from the applet owning the data to another applet without explicit permissions [13]. In general, the firewall blocks access to the data in one CAP file to an applet in another CAP file (having different execution contexts), except some well-defined exceptions (such as global arrays, ArrayViews or class variables).

### B. Target of Evaluation

The target of evaluation (TOE) of our project includes a set of Security Functions (TSF) that support the `O.Firewall` Security Objective enforcement. The implementation of these Security Functions in our project consists in a large subset of C functions of the JCVM. This subset features all possible functionalities of a single JCVM run that ensures the execution of any applet being selected. The JCVM specification [14] describes the data life cycle of an applet: transient deselect data is erased during owning application deselection, transient reset data is erased only when the smart card is reset, and persistent data is preserved anytime. Data and the associated life cycle of an undefined number of applets are carefully considered in our Security Policy Model (SPM) in order to ensure isolation properties. The SPM is deemed to comply with security functional requirements (SFRs) of the `O.Firewall` security objective defined in the JavaCard System Protection Profile [11], according to which "an applet shall not read, write, compare a piece of data belonging to an applet that is not in the same context, or execute one of the methods of an applet in another context without its authorization".

### C. Formal Security Policy Model (SPM)

Before this work, ANSSI's requirements for formal methods in the context of Common Criteria evaluations were only successfully fulfilled using B and Coq methods. This project was part of a pilot evaluation due to the adoption of a new formal method based on Frama-C [3], a verification platform for C code. Frama-C uses ACSL (ANSI C Specification Language) [15], a formal specification language for C programs. It allows the user to specify annotations that express the expected program properties. The Frama-C/WP plugin can then be used to prove them for each function of the code: this technique is called *(modular) deductive verification*. In order to build the formal security policy model (SPM), we follow a bottom-up approach, in which the C code implementation is enriched with annotations instead of merely making a separate model and a formal representation of a set of SFRs being claimed (see Fig. 1). The security properties are proved to be enforced by the actual implementation design. Examples of ACSL annotations and security properties are given in the next section.

## IV. Formal Specification of Contracts and Security Properties

### A. Function Contracts

Deductive verification with Frama-C/WP requires that each C function be annotated with a *(function) contract* expressed in ACSL language. Such a contract contains *preconditions* (in **`requires`** clauses), which express properties of program variables that must be respected before the function is called, and *postconditions* (in **`ensures`** clauses), which express properties that must be ensured after the function terminates. A special kind of postconditions is expressed by **`assigns`** clauses, which give the list of variables that the function is allowed to modify. All other variables cannot be modified by the function. Each function should then be proved by Frama-C/WP to respect its contract.

```
1 /*@ requires \valid(p1) && \valid(p2) && \separated(p1,p2);
2     ensures \old(*p1) == *p2 && \old(*p2) == *p1;
3     assigns *p1, *p2; */
4 int swap(int *p1, int *p2){
5   int tmp = *p1;
6   *p1 = *p2;
7   *p2 = tmp;
8 }
```

Fig. 2. A C function `swap` which permutes the values referred to by two given pointers `p1` and `p2`, and its ACSL contract.

All functions in our Formal Security Policy Model are annotated by ACSL contracts (see [4] for detailed examples). For lack of space, we illustrate an ACSL contract in Fig. 2 for a very simple C function `swap` that swaps the values `*p1` and `*p2` referred by the two pointers given as function arguments. Line 1 expresses a precondition stating that input pointers `p1` and `p2` must refer to *valid* memory locations, that is, locations that can be safely accessed, and that these locations are *separared*, that is, the underlying bytes are disjoint. The validity is necessary here to ensure the absence of runtime errors, also known as *undefined behaviors*. The separation assumption is necessary to avoid overwriting some bytes of `*p2` when modifying `*p1`, and vice versa. Line 2 expresses a postcondition: the values after the function returns are swapped. Keyword **`\old`**`(e)` used in a postcondition allows

```
1 /*@ // === A security property: object headers remain intact ===
2   predicate object_headers_intact{L1, L2} =
3     \forall integer i, off; 0 <= i < \at(gNumObjs,L1) &&
4       \at(gHeadStart[i],L1) <= off < \at(gHeadStart[i],L1) + 8 ==>
5       \at(ObjHeader[off],L1) == \at(ObjHeader[off],L2); */
```

Fig. 3. Example of a security-related predicate expressing that object headers are not modified between program points `L1` and `L2`.

```
1 /*@ // === Metaproperties: persistent object data written/read only by the object owner ===
2   meta \prop,\name(persi_objects_integrity),\targets(\ALL),\context(\writing),
3     ( \forall integer i; 0 <= i < gNumObjs && !gIsTrans[i] && ObjHeader[gHeadStart[i] + 0] != JCC ==>
4       \separated(\written,PersiData+(gDataStart[i]..gDataEnd[i])) );
5
6   meta \prop,\name(persi_objects_confidentiality),\targets(\ALL),\context(\reading),
7     ( \forall integer i; 0 <= i < gNumObjs && !gIsTrans[i] && ObjHeader[gHeadStart[i] + 0] != JCC ==>
8       \separated(\read,PersiData+(gDataStart[i]..gDataEnd[i])) ); */
```

Fig. 4. Metaproperties for integrity and confidentiality of persistent object data.

one to refer to the value of expression `e` before the call. Finally, line 3 states that `*p1` and `*p2` are the only memory locations the function is allowed to modify.

### *B. Security Properties Expressed as Predicates*

Some security properties (typically, for integrity) can be specified in ACSL as invariant properties maintained by relevant functions and directly proved by Frama-C/WP. We illustrate one of such properties, expressed by predicate `object_headers_intact` shown in Fig. 3, stating that object headers are not modified between program points `L1` and `L2`. For lack of space, we give here only the main ideas of this definition and refer the reader to [4] for detailed explanations. This predicate states that for any allocated object (represented by its index `i`) and for any offset `off` within the object header, the byte at offset `off` in the object header of object `i` has the same value in program points `L1` and `L2`. A typical usage of this predicate is to include the postcondition **`ensures`** `object_headers_intact{Pre, Post};` in the contract of every function. This postcondition ensures that the object headers are not modified by the function between program points `Pre` and `Post`, which refer to the states before and after the function call.

### *C. Security Properties Expressed as Metaproperties*

Other properties (for confidentiality or some cases of integrity) are stated as *metaproperties* [16]. The main principle of a metaproperty is to state a global property for a specified set of target functions and a specified context. Two examples of key metaproperties are shown in Fig. 4. Again, we give here the main ideas of their definition, a more detailed presentation being available in [4]. Metaproperty `persi_objects_integrity` states that the data of a persistent object (represented by its index `i`) cannot be modified unless the current context (`JCC`) is the object owner (which is stored at offset 0 in the object header). Similarly, metaproperty `persi_objects_confidentiality` states that the data of a persistent object (represented by its index `i`) cannot be read unless the current context is the object owner.

Each metaproperty is instantiated by the Frama-C/MetAcsl plugin into assertions in relevant program points in all target functions. For example, the first metaproperty has the writing context, therefore the corresponding property must be checked each time when a memory location is modified. So an assertion is automatically added by the Frama-C/MetAcsl plugin at all those memory locations, where the metavariable `\written` is replaced by the written memory location. The proof of the resulting assertions ensures that the metaproperty is globally respected by the code.

The proof of real-life code in our project requires a careful combination of several ingredients (see Fig. 6): macros, companion ghost code, global preservation properties in addition to lemmas and proof scripts. Macros reduce redundancy in specifications and facilitate updates and maintenance. Ghost code is mainly used to describe the memory model and to offer an alternative encoding of low-level operations, amenable to automatic provers. This combination made it possible to efficiently reason about non-trivial code fragments involving bitwise operations without the use of external interactive tools (e.g. Coq) with a high level of automatic proof.

## V. Traceability Between CC Requirements and Implementation

*a) General Approach:* As described in Section II, Security Objectives and related Security Functional Requirements are summarized in a Protection Profile describing a particular product type. In this project, the Security Target is related to the Protection Profile for JavaCard System [11]. To establish a correspondence between formal (SPM) and informal concepts (ST), the developer must establish and describe the links between them, as mandated by [10]. In fact, ANSSI and BSI (the German national certification authority) have driven the use of formal methods in Common Criteria evaluations with the publication of guidance for developers and evaluators (Note-12 [10] and AIS34 [17]). Formal analyses in CC context consist in giving a proof that the TOE Security Functions correctly implement the expected security objectives. Several "representations" of the TSFs are provided as shown in Fig. 5:

- SPM: the Security Policy Model contains only the mechanisms directly supporting Security Objectives enforcement,
- FSP: the Functional Specification of the implementation where low-level details are abstracted away,
- TDS: the TOE design or a simplified version of the implementation,
- IMP: the most concrete representation.

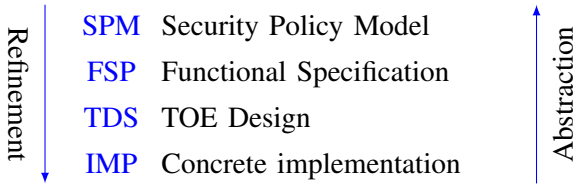


Fig. 5. Representations of the TSFs

The CC Assurance Development (ADV) components : Security Policy Model (ADV_SPM), Functional Specification (ADV_FSP), Target of evaluation Design (ADV_TDS), Implementation representation (ADV_IMP) provide a list of requirements to be fulfilled by each of the Security Function representations. Developers are also expected to establish a "formal equivalence" of these various representations of the TSF. The primary objective is to formally establish the correctness of the SPM w.r.t. security objectives. The rationale is to authorize reasoning at an abstract level (SPM) and to propagate the result toward the implementation (IMP).

The lack of proof of refinement until the implementation is the rub with the top-down strategy [18], [19], [20]. This is all the more unfortunate since most of the time the formal model is provided a posteriori, only for CC evaluation, and is not used to guide the design of the implementation.

Since our verification approach is based on formal properties written in ACSL annotations directly on the implementation, we link all informal concepts to formal ACSL annotations. Fig. 6 summarizes how the CC Assurance Development (ADV) components (Implementation representation (ADV_IMP), Target of evaluation

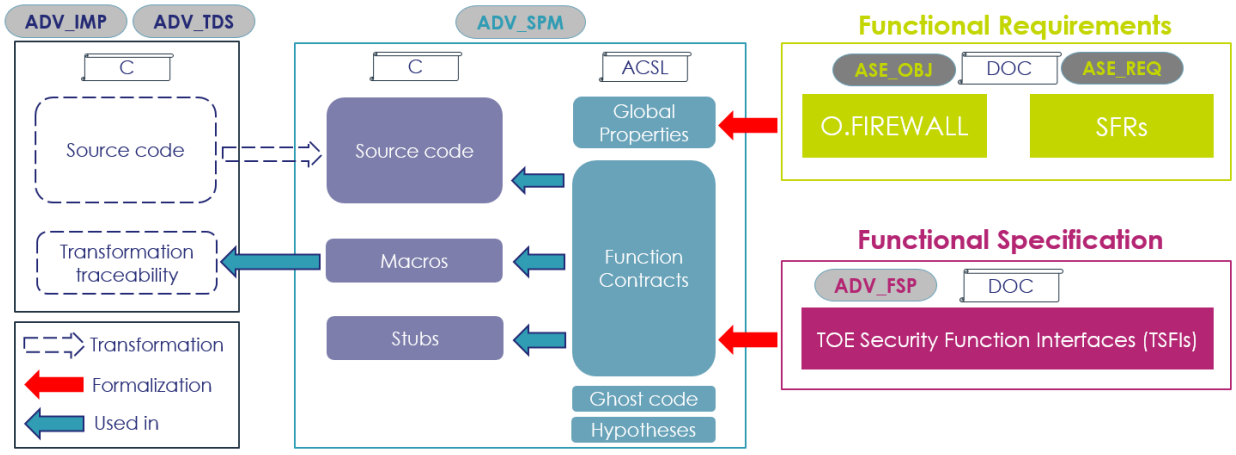


Fig. 6. Structure of the SPM with respect to CC requirements.

Design (ADV_TDS), Security Policy Model (ADV_SPM), Functional Specification (ADV_FSP)) are structured with respect to the actual product source code.

The proposed bottom-up approach intrinsically encompasses the refinement from the functional specification through the design to the implementation that is usually required in top-down methodologies.

*b) Mapping Security Objectives and Requirements:* The Security Objective defined in the Security Target (in our case, `O.FIREWALL`) is mapped to global ACSL annotations expressing the required isolation properties (either as ACSL predicates used as invariants, or as metaproperties for Frama-C/MetAcsl). A second mapping is then done for Security Functional Requirements (SFRs), describing the characteristics of the model (i.e. the contained functionality) for which the Security Objective is proved. Both mappings (Security Objectives and SFRs) are helpful to clearly articulate the security behavior chosen to be modeled, in other words, to clearly define the scope of the Security Policy Model (SPM). Security Functional Requirements are linked against some ACSL annotations in the SPM, it can be for example the contract of the firewall function or assignment specification of important variables decisive for firewall result (current context or context of objects). To make the mapping easier to evaluate, separate tables are provided to "translate" the Common Criteria terms of low granularity (e.g. Sharing, Currently Active Context) into their counterparts inside the Security Policy Model. It helps the evaluator to check the correctness of the mappings for particular SFRs based on these terms. For instance, SFRs introduced in II-C are mapped to their formalization in ACSL (depicted in Fig. 3 and 4) as follows:

**SFR1** As sensitive security attributes are stored in object headers, global invariants are used to prove that the headers of objects are intact during the entire VM run. The predicate `object_headers_intact{Pre, Post}` used in a function contract, ensures that the content of object headers is the same at the end of the function as it was before executing the function. Thus, preserving this predicate throughout all relevant function calls during the VM run ensures the integrity of sensitive security attributes.

**SFR2** Metaproperties `persi_objects_integrity` and `persi_objects_confidentiality` target integrity and confidentiality of object data respectively. In case the owner of a persistent object is different from the active context (`JCC`), any accessed memory location must be separated from the body of this object (see also Section IV-C).

## VI. Specification Effort and Proof Results

| JCVM C code | | ACSL Annotations | | | | |
|---|---|---|---|---|---|---|
| | | User provided annotations | | | Generated by MetAcsl | Generated by RTE |
| # Functions | # Loc C | # Loc Ghost | #Loc Metaproperties | # Loc other ACSL annotations | # Loc ACSL | # Loc ACSL |
| 381 | 7,014 | 162 | 350 | 35,480 | 396,603 | 2,290 |

Fig. 7. Specification effort for real-life code.

Applying deductive verification on large real-life industrial code requires a lot of care in order to avoid or at least mitigate scalability issues. In this project, we managed to ensure the scalability of our approach despite of the big size of the analyzed C code (7014 lines of C code split into 381 functions). As low-level operations are difficult to handle by automated provers, it was necessary to make abstraction of such low-level operations in a sound way. For that purpose, we introduced 162 lines of ghost code, carefully chosen to help automatic provers. As shown in Fig. 7, 35,480 lines of user-provided ACSL annotations were required in order to formally specify the security policy of the targeted virtual machine. However, we succeeded to reduce this effort up to 13,432 lines of ACSL annotations thanks to the usage of parameterized macros that gather redundant annotations. The effort is still considerable. 396,603 lines of ACSL annotations were automatically generated from 36 metaproperties (written in 350 lines of ACSL) only by Frama-C/MetAcsl. 2,290 lines of ACSL annotations were automatically generated by Frama-C/RTE.

Our proof scales reasonably well with an increasing number of proof goals. In particular, thanks to the translation of metaproperties into annotations that do not overload proof contexts, the metaproperty-based approach scales very well, despite a great number of generated annotations. Overall, it takes almost 3h30m to prove 52,198 proof goals.

99% of proof goals are automatically discharged by automatic provers. However, an important manual effort is required to maintain the proof of remaining proof goals when the code or the formal model are updated.

## VII. Related Work

A classical approach of applying formal verification on a JavaCard platform relies on a high-level formal model [18], [19], [20]. Several case studies have adopted this approach. An executable formal semantics of the JavaCard Virtual Machine (JCVM) and the ByteCode Verifier (BCV) is proposed in [21] with 15,000 lines of specification in the Coq proof assistant. An operational semantics of a language modeling the JCVM behavior is proposed in [22], [23]. Authors of [24] describe a refinement-based approach, relying on the Coq proof assistant, to show that a native JavaCard API function fulfills its specification. In general, in such approaches, the traceability of formally proven properties may require an important effort to be justified because of the gap between the formal model and the source code [2]. Among tools designed and/or used for the purpose of providing formal guarantees about JavaCard platform security properties (but not in a Common Criteria context) we can list: Key [25], KRAKATOA [26] and Caduceus [27].

This work is also broadly related to other projects in which real-world software is verified. For instance, formal verification of the seL4 microkernel (comprising 8700 lines of C and 600 lines of assembly) was performed in a certification context [28]. Heitmeyer et al. [29] report on evidence for a Common Criteria evaluation of an embedded software system, which uses a separation kernel (of over 3,000 lines of C and assembly code). Although the separation kernel enforcing data separation was annotated with pre- and postconditions in the style of Hoare and Floyd, the machine-checked proof is not directly applied on the C code. A mechanized formal proof is performed on a Top Level Specification (TLS) of the separation-relevant behavior of the kernel. A correspondence between the annotated code and the TLS was established separately.

In general, while the usage of formal methods is, indeed, a costly validation technique, it is often seen as counterproductive to the current industrial development processes, even for having an advantage over competitors. Especially when the obtained certificates need to be renewed regularly, in order to try to cope with the dynamic landscape of a product's life-cycle [30]. We believe that our approach is well-suited to optimize, albeit still high, the investments to reach EAL7 certifications using formal methods. Further enhancements of our approach for automatic generation of Common Criteria documentation like in [32], [33], [34], may be a step further for a better integration in current industrial development processes.

## VIII. Discussion and Lessons Learned

### A. From the Developers' Point of View

An important drawback of top-down certification approaches, based on a high-level model (e.g. in Coq) is the traceability issue: the difficulty to relate the model with the real-life code. It can be complex to ensure that the model faithfully represents the behavior of the code. Another issue is the maintainability of the model for the developers: changes can be difficult to integrate and can require a very significant review of the whole proof.

*a) Benefits:* The presented bottom-up approach facilitates traceability since the SPM in that case is (a subset of) the real-life code, with the same structure (same functions, variables, data structures, etc.). Another benefit of the proposed approach is a better maintainability (in particular, in case of minor code updates or scope extensions) and a more straightforward extension from EAL6 to EAL7. Compared to a top-down approach, where significant model and proof changes are often required for more complex properties or a larger scope, integration of new properties or functions for an EAL7 certification in the presented approach can more significantly rely on the proof performed for the EAL6 level. Small design changes can easily be integrated in order to check if the security properties are affected.

The proposed bottom-up verification approach strongly benefits from automation, which is particularly important for a large industrial product. The link between the SPM and the real-life code in our project is explicit and can be automatically exploited by various tools. For example, they include syntactic code comparison and identification of possible differences—code transformations—between the SPM and the real-life code. It is important to efficiently identify and review such transformations (used in the verified code of the SPM e.g. to avoid some tool limitations or to realize some scope restrictions). Construction of a control-flow graph can help to identify the function hierarchy,

in particular, to automatically distinguish fully verified functions (with a contract and a body), functions that are included as stubs (with a contract and a declaration but without a body) and excluded functions.

Furthermore, most security-related ACSL annotations in our approach are generated automatically from a few high-level security properties (stated as *metaproperties* [16]) by the Frama-C/MetAcsl plugin and then verified by Frama-C/WP. In this project, over 22,000 assertions are automatically generated by Frama-C/MetAcsl from (only!) 36 manually written metaproperties. Similarly, properties on the absence of *runtime errors* (RTE, also known as *undefined behaviors*) are generated automatically by another plugin, Frama-C/RTE, before being verified by Frama-C/WP.

Another major advantage of the approach is that it strongly relies on automatic proof. In the presented project, about 99% of proof goals are proved automatically by the Alt-Ergo solver or by the Frama-C/WP plugin (and its internal simplification engine Qed). A huge effort would be required to prove them interactively (that is, basically, manually) in a proof assistant. Finally, the proposed approach is suitable for a continuous integration process and it is planned to use it in the future in a continuous integration environment.

*b) Challenges and Points of Attention:* Those benefits also come with challenges for developers. An EAL6 certification with a bottom-up approach takes a more significant effort, already going closer to the implementation than actually required by the Common Criteria. Source-code level formal verification can be sensitive to tool scalability issues. Indeed, the tool has to deal simultaneously with high-level program properties and low-level properties (such as absence of runtime errors, presence of casts and bit-level operations), that can lead to a large number of relatively complex properties. A specific expertise and a good understanding of the capacities of the chosen verification tools and automatic solvers are required for the developers. Advanced tools like Frama-C/WP offer interactive proof features (proof scripts) that help the developer to finish most complex proofs.

A significant effort of manual annotation of the code is another challenge. In the presented project, $\sim$12,000 lines of annotations were written manually for $\sim$7,000 lines of C code (i.e. a factor of 1.7). Some of them rely on carefully chosen macros to avoid annotation redundancy, without using it the developers would have to write $\sim$35,000 lines of annotations (i.e. a factor of 5 compared to the C code). Another challenge is an efficient and meaningful organization of annotations and global properties—sometimes not obvious in modular verification—that can have an impact both on the capacity to prove and to define the mapping (see Section V).

### B. From the Evaluators' Point of View

*a) Benefits:*

1) the main benefit of the bottom-up approach for an evaluator is the immediate understanding of the formal entities, such as the modeling of the heap and the stack in a JCVM, because the program[2] has the same representation as the JCVM implementation, which has already been evaluated (ADV IMP);
2) as SFRs are directly represented in the program as formal annotations, the correspondence from the SFRs to the model can be easily understood by the evaluator, in particular to check that SFRs are modeled precisely enough to allow the verification of the security objectives;
3) only a single model needs to be reviewed because all the security and functional properties can be verified by the same model built on top of the implementation;
4) there is no refinement, no abstract level, no relation between multiple models to be evaluated;
5) the formal modeling of the subsystems is implicitly provided by the program;
6) apart from code transformations, the verified program has been written by a separate team (i.e. not the modeling team) which makes the relevance of the model easier to evaluate than when properties are verified on a dedicated model (in particular, a purely logical model);
7) the evaluator can directly check that well-known attack paths (e.g. type confusions) are not ignored by the model;
8) the bottom-up approach perfectly fits into the continuation of the evaluation process, unlike the top-down approach that involves a new (formal) design, which is more difficult to evaluate because of the traceability issue (as mentioned in Section VIII-A).

[2]The term "program" refers here to the C code part of the model, without the ACSL annotations and metaproperties.

*b) Limitations and Points of Attention:* The limitations of the bottom-up approach are caused by the code complexity which is directly transferred to the model making the whole proof too complex to be fully reviewed by the evaluator. However, deductive verification ensures that properties are correctly propagated between functions in the callgraph. If a property is verified in the contract of a caller, deductive verification ensures that the contracts of the callees are complete enough to verify the property at the caller level. Therefore, the evaluator can have confidence in the proof, as long as the properties are correctly defined, and the program is correctly modeled in terms of code transformations, ghost code, and hypotheses. Hence, the main points of attention are the following.

- *Properties:* The bottom-up approach may make the properties more complex to evaluate because they directly rely on the implementation. Metaproperties, which are defined globally and whose number is limited, can be reviewed more easily but their statement remains as complex as the low-level annotations.
- *Code transformations:* In some cases, the real-life code complexity cannot be fully supported by the existing tools, and requires manual transformations of the code. While some bugs in code transformations can be detected by the logical part of the model, some other bugs can lead to missing states and be hard to detect. An exhaustive manual review of the code remains difficult. Therefore, code transformations should rigorously follow a precise methodology even for the (apparently) simple cases.
- *Ghost code:* Companion ghost code increases the complexity of the model, but also helps the evaluator to understand how the proof is conducted. The evaluator should detect non-companion ghost code that is used to create new concepts in the model either to express properties that cannot be directly expressed with the concepts of the implementation, or to simplify the model. The evaluator needs to check that these new concepts are valid and consistent with the implementation.
- *Hypotheses:* Hypotheses can be introduced as preconditions of the entry point function, or contracts of stubs (whose code is not provided so their contracts are admitted). Stubs can include functions from the implementation removed for simplification, or new functions, specifically declared to introduce some local hypotheses at some points in the program. Local hypotheses make the understanding of the chain of reasoning and the detection of contradictory hypotheses more difficult. Additional tool-related hypotheses (like memory model assumptions in Frama-C/WP) also require specific attention of the evaluators.

### C. Conclusion

The bottom-up approach brings many benefits to the certification process in terms of model understanding and confidence in verified properties. It helps to reduce the gap between the formally proved properties and the implementation, and should facilitate the step from EAL6 to EAL7 for the developers. Modern verification tools like Frama-C/WP and Frama-C/MetAcsl are capable to deal with real-life code after only a limited number of code modifications. The application of bottom-up approaches can require some adjustments in the existing terminology and certification guidelines (e.g. [10]) released by certification bodies. Indeed, historically, they were designed with top-down approaches in mind, and their application on the bottom-up approach requires some clarifications. The existing evaluation methodology has to be extended with additional tasks for a careful analysis of properties, code transformations, ghost code and hypotheses.

## REFERENCES


[1] "Common criteria for information technology security evaluation. part 3: Security assurance components," CCMB-2017-04-003, Tech. Rep., 2017. [Online]. Available: https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf

[2] B. Beckert, D. Bruns, and S. Grebing, "Mind the gap: Formal verification and the common criteria (discussion paper)," in *Proc. of the 6th International Verification Workshop (VERIFY 2010)*, ser. EPiC Series in Computing, vol. 3. EasyChair, 2010, pp. 4–12.

[3] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Asp. Comput.*, pp. 1–37, 2015.

[4] A. Djoudi, M. Hána, and N. Kosmatov, "Formal verification of a JavaCard virtual machine with Frama-C," in *the 24th Int. Symp. on Formal Methods (FM 2021)*, vol. 13047. Springer, 2021, pp. 427–444.

[5] SGDN, "Application note: Target evaluation's security policies formal modelling," 587/SGDN/DCSSI/SDR - NOTE/12.1, Tech. Rep., 2008. [Online]. Available: https://www.ssi.gouv.fr/uploads/2014/11/NOTE-12-Modelisation-formelle-SPM-EN.pdf

[6] C. Lavatelli and G. Tétu, "Formal models for high assurance: why and how," in *International Common Criteria Conference*, 2020. [Online]. Available: https://www.internetoftrust.com/wp-content/uploads/2021/02/ICCC-2020-SPM_v1.0_20201118.pdf

[7] SGDN, "Remarques relatives à l'emploi des méthodes formelles (déductives) en sécurité des systèmes d'information," SGDN/DCSSI/SDS/LTI[1], Tech. Rep., 2008. [Online]. Available: https://www.ssi.gouv.fr/uploads/2014/11/ssi_formelle.pdf

[8] ANSSI-INRIA, "Requirements on the use of Coq in the context of common criteria evaluations," SGDN/ANSSI/SDE/DST/LSL, Tech. Rep., 2020. [Online]. Available: https://www.ssi.gouv.fr/uploads/2014/11/anssi-requirements-on-the-use-of-coq-in-the-context-of-common-criteria-evaluations-v1.0-en.pdf

[9] Y. Bertot, M. Dénès, V. Laporte, A. Fontaine, and T. Letan, "The use of Coq for common criteria evaluations," *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020), part of POPL 2020*, 2020.

[10] SGDN, "Note d'application : Modélisation formelle des politiques de sécurité d'une cible d'évaluation," 587/SGDN/DCSSI/SDR - NOTE/12.1, Tech. Rep., 2008. [Online]. Available: https://www.ssi.gouv.fr/uploads/2014/11/NOTE-12-modelisation-formelle.pdf

[11] Oracle, "Java Card system – open configuration protection profile, version 3.1," Oracle, Tech. Rep., 2020. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/ReportePP/pp0099V2b_pdf.pdf;jsessionid=6C3F5A7FB5FA0D928A1C310C1C0EF1CE.internet462?__blob=publicationFile&v=1

[12] X. Leroy, "Bytecode verification on Java smart cards," *Software: Practice and Experience*, vol. 32, no. 4, pp. 319–340, apr 2002.

[13] Oracle, "Java Card Platform: Runtime Environment Specification, Classic Edition, Version 3.1," Oracle, Oracle, Tech. Rep., feb 2021. [Online]. Available: https://docs.oracle.com/javacard/3.1/related-docs/JCCRE/JCCRE.pdf

[14] ——, "Java Card Platform: Virtual Machine Specification, Classic Edition, Version 3.1," Oracle, Oracle, Tech. Rep., feb 2021. [Online]. Available: https://docs.oracle.com/javacard/3.1/related-docs/JCVMS/JCVMS.pdf

[15] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language*, 2021. [Online]. Available: https://www.frama-c.com/download/acsl.pdf

[16] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall, "Methodology for specification and verification of high-level properties with MetAcsl," in *9th IEEE/ACM International Conference on Formal Methods in Software Engineering (FormaliSE 2021)*. IEEE, 2021, to appear.

[17] BSI, "Application notes and interpretation of the scheme (AIS), AIS 34, version 3," Bundesamt für Sicherheit in der Informationstechnik (BSI), Bundesamt für Sicherheit in der Informationstechnik (BSI), Tech. Rep., 2009. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_34_pdf.html

[18] B. Chetali and Q.-H. Nguyen, "About the world-first smart card certificate with EAL7 formal assurances," *The 9th International Common Criteria Conference*, 2008.

[19] ——, "Industrial use of formal methods for a high- level security evaluation," in *Proc. of the 15th International symposium on Formal Methods (FM 2008)*, 2008.

[20] D. Bolignano, "Formal proof of a secure OS full trusted computing base (invited paper)," in *Proc. of the 2nd International Workshop on Enabling Trust through OS Proof and Beyond (Entropy 2019), co-located the 4th IEEE European Symposium on Security and Privacy (EuroS&P 2019)*, 2019.

[21] G. Barthe, G. Dufay, L. Jakubiec, B. P. Serpette, and S. M. de Sousa, "A formal executable semantics of the JavaCard platform," in *10th European Symposium on Programming on Programming Languages and Systems, (ESOP 2001), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2001)*, ser. LNCS, vol. 2028. Springer, 2001, pp. 302–319.

[22] I. A. Siveroni, "Operational semantics of the Java Card Virtual Machine," *The Journal of Logic and Algebraic Programming*, vol. 58, no. 1-2, pp. 3–25, jan 2004.

[23] M. Éluard, T. Jensen, and E. Denne, "An operational semantics of the Java Card firewall," in *Smart Card Programming and Security, International Conference on Research in Smart Cards (E-smart 2001)*, ser. LNCS. Springer, 2001, vol. 2140, pp. 95–110.

[24] Q.-H. Nguyen and B. Chetali, "Certifying native java card API by formal refinement," in *The 7th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications (CARDIS 2006)*, ser. LNCS. Springer, 2006, vol. 3928, pp. 313–328.

[25] W. Mostowski, "Fully verified Java Card API reference implementation," in *Proceedings of the 4th Interenational Verification Workshop in connection with CADE-21. CEUR Workshop Proceedings, Vol-259*, 2007.

[26] C. Marché, C. Paulin-Mohring, and X. Urbain, "The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML," *The Journal of Logic and Algebraic Programming*, vol. 58, no. 1-2, pp. 89–106, jan 2004.

[27] J. Andronick, B. Chetali, and C. Paulin-Mohring, "Formal verification of security properties of smart card embedded source code," in *International Symposium of Formal Methods (FM 2005)*, ser. LNCS, vol. 3582. Springer, 2005, pp. 302–317.

[28] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, jun 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1743546.1743574http://web1.cs.columbia.edu/$\sim$junfeng/09fa-e6998/papers/sel4.pdf

[29] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *Proc. of the 13th ACM conference on Computer and communications security (CCS 2006)*. ACM Press, 2006, pp. 346–355.

[30] S. P. Kaluvuri, M. Bezzi, and Y. Roudier, "A quantitative analysis of Common Criteria certification practice," in *TrustBus 2014: Trust, Privacy, and Security in Digital Business*, 2014, pp. 132–143. [Online]. Available: https://www.eurecom.fr/fr/publication/4438/download/rs-publi-4438.pdf

[31] "The common criteria - certified products list - statistics." [Online]. Available: https://www.commoncriteriaportal.org/products/stats/

[32] P. Heck, M. Klabbers, and M. van Eekelen, "A software product certification model," *Software Quality Journal*, vol. 18, no. 1, pp. 37–55, 2010. [Online]. Available: https://link.springer.com/article/10.1007/s11219-009-9080-0

[33] F. Tuong and B. Wolff, "Deeply integrating C11 code support into Isabelle/PIDE," *Electronic Proceedings in Theoretical Computer Science*, vol. 310, pp. 13–28, dec 2019. [Online]. Available: https://arxiv.org/pdf/1912.10630.pdf

[34] S. Bezzecchi, P. Crisafulli, C. Pichot, and B. Wolff, "Making agile development processes fit for V-style certification procedures," *arXiv preprint arXiv:1905.06604*, may 2019. [Online]. Available: http://arxiv.org/abs/1905.06604