Formal Verification of a JavaCard Virtual Machine with Frama-C

Adel Djoudi[1 [0000−0002−8238−6490]], Martin Hána[2 [0000−0001−8977−5950]], and Nikolai Kosmatov[3 [0000−0003−1557−2813]]

[1]Thales Digital Identity & Security, Meudon, France
[2]Thales Digital Identity & Security, Prague, Czech Republic
[3]Thales Research & Technology, Palaiseau, France
firstname.lastname@thalesgroup.com

Abstract. Formal verification of real-life industrial software remains a challenging task. It provides strong guarantees of correctness, which are particularly important for security-critical products, such as smart cards. Security of a smart card strongly relies on the requirement that the underlying JavaCard virtual machine ensures necessary isolation properties. This case study paper presents a recent formal verification of a JavaCard Virtual Machine implementation performed by Thales using the Frama-C verification toolset. This is the first verification project for such a large-scale industrial smart card product where deductive verification is applied on the real-life C code. The target properties include common security properties such as integrity and confidentiality. The implementation contains over 7,000 lines of C code. After a formal specification in the ACSL specification language, over 52,000 verification conditions were generated and successfully proved. We present several issues identified during the project, illustrate them by representative examples and present solutions we used to solve them. Finally, we describe proof results, some lessons learned and desired tool improvements.

1 Introduction

Safety and security of critical software has become a major concern today. Formal software verification is able to rigorously demonstrate the absence of bugs and security flaws. It provides strong guarantees of correctness, which are particularly important for security-critical products, such as smart cards. However, formal verification of real-life industrial software still remains a challenging task.

Security of a smart card strongly relies on a set of isolation properties that must be ensured by the underlying JavaCard virtual machine. According to [20], "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 corresponding access rules are usually implemented by a specific access control mechanism ensuring necessary isolation, that is called a *firewall*.

This case study paper presents a recent formal verification of a JavaCard Virtual Machine (JCVM) implementation performed by Thales using the Frama-C verification platform [14]. It was realized for a Common Criteria EAL6 certification (for which the certificate was recently issued). Target properties include common security properties—such as integrity and confidentiality—that ensure the required access control rules. The perimeter of verification includes the majority of functions of the JCVM with over 7,000 lines of C code. They were formally annotated in the ACSL specification language [4], and verified using the WP and MetAcsl plugins of Frama-C. Overall, more than 52,000 proof goals (verification conditions) were generated and successfully proved. As far as we know, this is the first verification project for such a large-scale industrial smart card product where deductive verification is applied on the real-life C code.

Contributions. The contributions of the paper include a presentation of the formal verification case study fully realized in the industrial context. It took approximately 3 person-years. We present our specification and verification approach, emphasize some of the issues faced during the project and briefly explain how they were solved. The issues include bit-related operations, heterogeneous pointer casts, existential quantifiers, prover scalability and failures of automatic proof, expressing and efficient proof of global (e.g. security) properties, and code maintenance. While most solutions are not new, their accurate combination was essential for the proof. We also describe some extensions and improvements of Frama-C—some of which were key for the success of this project—identified during the project and implemented by its developers. Finally, we present our proof results, some lessons learned and the desired tool improvements. The source code of smart card products is highly sensitive and cannot be shared. To illustrate the implementation structure, a subset of properties and our verification approach, we use a toy example. Due to space limitations, some other specific aspects of this work (in particular, related to stack verification and the Common Criteria evaluation methodology) will be described in future publications.

Outline. Sections 2 and 3 provide necessary background on Frama-C and JCVM. Our memory modeling approach using a companion ghost model is presented in Sect. 4. Section 5 describes the usage of lemmas and scripts to help the proof. Security property specification and verification with MetAcsl are presented in Sect. 6. Section 7 discusses code organization and maintenance. Our proof results, lessons learned and expected future improvements are presented in Sect. 8. Finally, Sections 9 and 10 present related work and a conclusion.

2 Frama-C Verification Platform

This section briefly presents the verification tools used in this project. We assume that the reader is familiar with basic notions of contracts and deductive verification.

Frama-C [14] is a program verification platform for C code developed by CEA List with participation of INRIA. It offers several analyzers organized as plugins around a common kernel. Developed since 2008, Frama-C has been used in several academic and industrial projects and has a large community of developers and users. Today, thanks to an active development, rigorous validation process and a wide range of applications throughout the world, Frama-C is considered as a state-of-the-art verification toolset for C programs. Frama-C uses ACSL (ANSI C Specification Language) [4], a formal specification language for C programs. It allows the user to specify annotations as typed first-order logic formulas. In particular, a precondition (ACSL clause **`requires`**) of a function f defines a property expected to hold before any invocation of f. It is assumed on entry during the verification of f, but must be proved before any call of f during the verification of a caller. A postcondition (**`ensures`** clause) of f states a property that must hold after the function call. It must be proved during the verification of f but is assumed after the call to f in a caller. An **`assigns`** clause in the contract of f specifies the memory locations that can be modified by f. A **`loop invariant`** clause in a loop contract specifies a property that must hold before the loop and after every loop iteration. Local properties (ACSL clause **`assert`**) must hold at the point where they are inserted.

The WP plugin of Frama-C is dedicated to modular deductive verification of C code. It takes as input a C program and a (partial) formal specification expressed by ACSL annotations and tries to prove that the program respects the provided annotations. To do that, WP generates proof goals (or proof obligations, or verification conditions) that are then either proved by WP itself (thanks to its internal formula simplification engine called Qed [8]) or sent via the Why3 tool [12] to external provers (SMT solvers). In this work we use Alt-Ergo [7], chosen in agreement with the certification authority. If all proof goals generated for a given program with ACSL annotations are proved, the program is guaranteed to respect the given ACSL specification. In addition to the specified properties, to ensure that the program under verification represents no risk of provoking undefined behaviors (also called runtime errors), WP relies on the RTE plugin of Frama-C to generate additional assertions to exclude the risk of undefined behaviors. Their proof is an essential step, both for the soundness of verification and to avoid security vulnerabilities due to undefined behaviors. Finally, one can rely on the MetAcsl plugin [23–25] to verify some security properties. It allows the user to specify global properties and translates them into local assertions (for instance, for each reading or writing operation) that can then be verified by other tools (like WP).

3 Overview of JavaCard Virtual Machine

This section briefly presents JavaCard Virtual Machine (whose detailed understanding is not mandatory to follow the paper).

JavaCard Virtual Machine (JCVM) [22] is often part of a smart card architecture and plays an essential role in it: it executes JavaCard bytecode. Its main functional goal is to provide application interoperability, when the same JavaCard bytecode can be run without re-compilation on several smartcard architectures. JavaCard bytecode contains a sequence of opcodes, each defining a particular operation, e.g. allocation of a new object of a given class, arithmetic operation on the Java stack, writing into a heap object a value read from the

Java stack, etc. Opcodes are read iteratively inside the main *dispatch loop*, which calls functions implementing particular opcodes.

We can distinguish 3 main memory locations managed by JCVM: Java stack, data heap and code area. Inside the code area, immutable information of the package(s) is stored during loading of application to the card, and is never modified later. It contains mainly package classes, including bytecode of their methods.

The data heap is used for storage of application data, namely class instances, arrays and (static) class variables. Three different types of memory are used to store heap data, depending on its life cycle: *transient deselect* data erased during owning application deselection, *transient reset* data erased only when the smart card is reset, and *persistent data* preserved anytime. The Java stack is a central location for data processing. In particular, its subpart, the operand stack, serves to realize arithmetic operations and for data exchange with the heap memory.

Next to functional aspects, JCVM also ensures security goals essential to the smart card ecosystem. As applications of different vendors can be loaded on the same smart card, it is crucial to guarantee isolation, in particular concerning the heap data. It is done by an oncard software component, JavaCard firewall [21].

A unique *context* value is assigned to each JavaCard binary (*CAP file*) during loading to the card. In general, the firewall blocks access to the data of another CAP file, except well-defined exceptions such as global arrays, ArrayViews or class variables. For simplicity, we ignore those exceptions in the examples in this paper. As a natural implementation choice, for each object on the heap (except class variables), the object owner context is stored inside the object header.

For illustration, we can list some examples of (simplified) security goals, which refine the general objective of CAP file data isolation, as defined in [20] (and quoted in Sect. 1), and distinguish confidentiality and integrity aspects.

$\mathcal{G}^{\text{head}}_{\text{integ}}$: Already allocated objects' headers cannot be modified during a VM run.

$\mathcal{G}^{\text{data}}_{\text{integ}}$: Elements of a (persistent or transient reset) array can be modified only if the accessing context is the owner of the accessed object.

$\mathcal{G}^{\text{data}}_{\text{conf}}$: Elements of a (persistent or transient reset) array can be read only if the accessing context is the owner of the accessed object.

There is an interplay between JCVM and another important security component of a smart card environment, Bytecode Verifier (BCV). As stated in [20], BCV must check any application prior to its execution to ensure its type security. On the other hand, (fully) defensive VMs ensure the discussed security properties even if the interpreted application is not verified by BCV. However, the toy example considered in this paper does not contain necessary sanity checks, so they rely on successful BCV checks. Thus, to enable its proof, we introduce hypotheses corresponding to particular BCV checks. For example, function `updateJPC` (modifying the Java program counter) will simply assume the updated code position stays within the method component of the CAP file.

The influence between JCVM and BCV is bidirectional. Indeed, BCV applies typed based simulation of bytecode, investigating all possible paths through the

```
 1 typedef unsigned char u1; typedef unsigned short u2; typedef unsigned int u4;
 2
 3 // === Code model and current Java context ===
 4 #define CODE_SIZE 10000
 5 u1  Code[CODE_SIZE], *JPC; // Java code area and Java program counter
 6 //@ ghost u4 gJPCOff;      // JPC offset in code area
 7 u1  JCC;                   // Current Java context
 8
 9 // === Heap model ===
10 #define SEGM_SIZE 10000
11 #define MAX_OBJS  500
12 u1 ObjHeader[SEGM_SIZE];   // Object headers area
13 //Header(8B),Bytes:Contents: 0:Owner,1:Flags,2-3:Class,4-5:BodyOff,6-7:BodySize
14 #define GET_OWN(addr)  ( *((u1*)addr + 0) )
15 #define GET_FLAG(addr)  ( *((u1*)addr + 1) )
16 #define GET_OFF(addr)  ( (u2)((*((u1*)addr + 4))*256 + *((u1*)addr + 5)) )
17 #define GET_SIZE(addr)  ( (u2)((*((u1*)addr + 6))*256 + *((u1*)addr + 7)) )
18 u1 PersiData[SEGM_SIZE];   // Persistent objects data area
19 u1 TransData[SEGM_SIZE];   // Transient  objects data area
20
21 /*@ ghost // === Companion ghost memory view ===
22   u4 gNumObjs;             // Number of allocated objects
23   u1 gIsTrans  [MAX_OBJS]; // Nonzero for transient object
24   u4 gHeadStart[MAX_OBJS]; // Start offset of object header
25   u4 gDataStart[MAX_OBJS]; // Start offset of object data
26   u4 gDataEnd  [MAX_OBJS]; // End offset of object data
27   u4 gCurObj;  */          // Currently considered object number
28
29 /*@ // === Validity predicates ===
30 predicate valid_code_model = 0 <= gJPCOff < CODE_SIZE &&
31   JPC == &Code[gJPCOff];
32 predicate valid_heap_model =
33   0 <= gNumObjs <= MAX_OBJS &&
34 // headers of allocated objects are within ObjHeader segment
35   (\forall integer i; 0 <= i < gNumObjs ==>
36     0 <= gHeadStart[i] <= SEGM_SIZE - 8 ) &&
37 // no overlapping between headers (each header has 8 bytes)
38   (\forall integer i,j; 0 <= i < j < gNumObjs ==>
39     (gHeadStart[i] >= gHeadStart[j]+8 || gHeadStart[j] >= gHeadStart[i]+8) ) &&
40 // IsTrans[i] encodes if i-th object's transient bit is set
41   (\forall integer i; 0 <= i < gNumObjs ==>
42     ( gIsTrans[i] <==> (GET_FLAG(ObjHeader+gHeadStart[i]) & 0x08) ) ) &&
43 // data of allocated objects is within a data segment
44   (\forall integer i; 0 <= i < gNumObjs ==>
45     gDataStart[i] == GET_OFF(ObjHeader+gHeadStart[i]) &&
46     gDataEnd[i] == gDataStart[i] + GET_SIZE(ObjHeader+gHeadStart[i]) - 1 &&
47     0 <= gDataStart[i] < gDataEnd[i] < SEGM_SIZE ) &&
48 // no overlapping between persistent object data
49   (\forall integer i,j; 0<=i<j<gNumObjs && !gIsTrans[i] && !gIsTrans[j] ==>
50     (gDataStart[i] > gDataEnd[j] || gDataStart[j] > gDataEnd[i]) ) &&
51 // no overlapping between transient object data
52   (\forall integer i,j; 0 <= i < j < gNumObjs && gIsTrans[i] && gIsTrans[j] ==>
53     (gDataStart[i] > gDataEnd[j] || gDataStart[j] > gDataEnd[i]) ); */
54
55 // Lines 56-66 give declarations of functions updateJPC, get_u1, get_u4, get_gu4.
```

Fig. 1. Illustrative example of JCVM: code and heap modeling.

code and checking its type safety [19]. As a hypothesis, BCV relies on the opcode specification [22] and its effect on the memory managed by JCVM (e.g. number of slots popped and pushed on the Java stack). It is therefore mandatory to check that JCVM respects this specification.

4 Memory Modeling and Companion Ghost Model

To illustrate our verification approach, we use a toy example of a JCVM, split into Fig. 1–5, where we omit some less important fragments or empty lines. It is strongly simplified to fit the paper, and intentionally modified to avoid revealing real-life code features. It is of course too simple to provoke proof issues we faced on real-life code, but sufficient to explain when they occur and how we address them. We consider one JCVM run, in which allocated objects cannot be deleted, but new objects can be allocated. Figure 4 presents the dispatch loop that reads the next opcode and calls the relevant opcode function. An opcode function and a simple firewall function are shown in Fig. 3. We detail all components of the example below. In this section, we explain Fig. 1 and the C code of Fig. 3.

Line 1 in Fig. 1 defines unsigned integer types with 1, 2 and 4 bytes. Lines 3–7 show a simple code model, where Java program counter `JPC` will be assumed to refer inside the `Code` array as specified by the code model validity predicate on lines 30–31. This predicate will be maintained by most functions in our example. Here, *to facilitate the automatic proof, we avoid an existentially quantified offset* by introducing the offset `gJPCOff` as a *ghost* variable (i.e. used only in ACSL annotations). We start the names of ghost variables with a `g`.

Lines 9–27 show a simplified model of the heap, where we model only persistent and transient reset objects. We consider three separate memory segments: for objects headers, persistent object data and transient (reset) object data (cf. lines 12, 18–19). A header contains the object's owner context (1 byte), flags (1 byte), class reference (2 bytes), followed by the start offset of the object data (body) and its size, each over 2 bytes (cf. line 13). Macros on lines 14–17 extract some of these fields. The number of allocated objects is specified as a ghost variable `gNumObjs`, and the allocated objects are supposed be numbered starting from 0. For the *i*-th object, the offset of its header is modeled by a ghost array element `gHeadStart[i]`, while `gDataStart[i]` and `gDataSize[i]` contain the offset and size of its body in one of the data segments. The ghost array element `gIsTrans[i]` is nonzero iff the *i*-th object has transient data.

The heap model validity predicate specifies first the value interval for the number of allocated objects (line 33). Lines 34–39 state that headers are within the bounds of the segment and do not overlap. Similarly, lines 43–53 state that object bodies are within the bounds of data segments and—when in the same segment—do not overlap. Thus, *we precisely model the heap memory using a companion ghost model*, some parts of which are not readily available in the C code. The heap validity predicate is maintained by most functions, including new object allocation.

Optimized code often uses bits, e.g. to encode various flags. Let us consider here only one bit: the *transient bit*, obtained from the flag byte with mask `0x08`. If this bit is set, the object data is located in the transient segment (with the offset and size given in the header), otherwise in the persistent segment (with the offset and size given in the header). Its usage is well illustrated by function `bastore` (see lines 110–119 in Fig. 3). It writes a given value into a given object at a given offset. After calling the firewall to check the access, it tests

```
67 /*@ // === A security property: object headers remain intact ===
68 predicate object_headers_intact{L1, L2} =
69   \forall integer i, off; 0 <= i < \at(gNumObjs,L1) &&
70     \at(gHeadStart[i],L1) <= off < \at(gHeadStart[i],L1) + 8 ==>
71     \at(ObjHeader[off],L1) == \at(ObjHeader[off],L2);
72
73 // === Memory footprint predicate and lemma example ===
74 predicate mem_model_footprint_intact{L1,L2} =
75   \at(gNumObjs,L1) <= \at(gNumObjs,L2) &&
76   ( \forall integer i; 0 <= i < \at(gNumObjs,L1) ==>
77     \at(gIsTrans[i],L1) == \at(gIsTrans[i],L2) &&
78     \at(gHeadStart[i],L1) ==\at(gHeadStart[i],L2) &&
79     \at(gDataStart[i],L1) ==\at(gDataStart[i],L2) &&
80     \at(gDataEnd[i],L1) ==\at(gDataEnd[i],L2) );
81
82 lemma vhm_preserved{L1,L2}: mem_model_footprint_intact{L1,L2} &&
83   object_headers_intact{L1,L2} && valid_heap_model{L1} &&
84   \at(gNumObjs,L1) == \at(gNumObjs,L2) ==> valid_heap_model{L2}; */
```

Fig. 2. Examples of a security property, a footprint-related predicate and a lemma.

the transient bit to choose and write the target memory location before moving the program counter to a next opcode (using the function `updateJPC`, omitted here). The firewall function (see lines 93–97 in Fig. 3) allows the access if the current context is the object owner and the destination offset is in the bounds (cf. $\mathcal{G}^{\text{data}}_{\text{integ}}$, $\mathcal{G}^{\text{data}}_{\text{conf}}$ in Sec. 3). In the real-life code, several bits can be manipulated within the same function, leading to complex proof goals.

Straightforward specification of the code with bit-related operations does not scale well in our case study: automatic proof fails for many properties over the real-life code when numerous bits are involved, thus requiring extra assertions or interactive scripts. *To overcome proof scalability issues due to bit-level operations, we duplicate the bit-level information by boolean ghost variables and maintain their equivalence.* That is why we encode the transient bit of the *i*-th object as a ghost array element `gIsTrans[i]`, as specified by lines 40–42. By expressing annotations using the resulting ghost variables (like on lines 49, 52, or as we will see later in Fig. 5) rather than the transient bit, we provide the provers with a parallel, companion view of bit-level information. It enhances their capacity of automatic proof in our project.

Heterogeneous pointer casts present another difficulty faced in our project. The definitions of macros of lines 16–17 in real-life code would use such casts:

```
#define GET_OFF(addr)  ( (u2)(*(u2*)(addr + 4)) )    // Before rewriting of casts
#define GET_SIZE(addr) ( (u2)(*(u2*)(addr + 6)) )    // Before rewriting of casts
```

To allow the proof with the Typed memory model of WP, we rewrite such casts equivalently as shown on lines 16–17. The equivalence of rewriting can be checked even by an exhaustive enumeration. The Typed memory model of WP [3] is both sound and efficient, but unable to support heterogeneous pointer casts. Lower-level models are either unsound or unable to reason efficiently on our case study. Introducing ghost variables to store the resulting casted values (cf. lines 45–46) and using those ghost variables in annotations (cf. line 91) also had a positive effect on the automatic proof.

Overall, the companion ghost model in our project has a twofold role: it allows us to conveniently express memory-related properties and facilitates au-

```
 86 /*@
 87   requires vhm: valid_heap_model;
 88   requires 0 <= gCurObj < gNumObjs && ObjRef == gHeadStart[gCurObj];
 89   assigns \nothing;
 90   ensures \result <==> ( GET_OWN(ObjHeader+ObjRef) == JCC &&
 91     gDataStart[gCurObj] + DestOff <= gDataEnd[gCurObj] );
 92 */
 93 u1 firewall(u4 ObjRef, u4 DestOff){
 94   if (GET_OWN(ObjHeader+ObjRef) == JCC && DestOff < GET_SIZE(ObjHeader+ObjRef))
 95     return 1;
 96   return 0;
 97 }
 98
 99 /*@
100   requires vhm: valid_heap_model;
101   requires vcm: valid_code_model;
102   admit requires 0 <= gCurObj < gNumObjs && ObjRef == gHeadStart[gCurObj];
103   assigns  PersiData[0..(SEGM_SIZE-1)],TransData[0..(SEGM_SIZE-1)],JPC,gJPCOff;
104   assigns  JPC \from &Code[0]; // possible base address
105   ensures  vhm: valid_heap_model;
106   ensures  vcm: valid_code_model;
107   ensures  oh:  object_headers_intact{Pre,Post};
108   ensures  mmf: mem_model_footprint_intact{Pre,Post};
109 */
110 void bastore(u4 ObjRef, u4 DestOff, u1 Val)
111 {
112   if( ! firewall(ObjRef,DestOff) )                       // Check access and
113     return;                                              // exit if forbidden
114   if( GET_FLAG(ObjHeader+ObjRef) & 0x08 )                // If trans. bit set,
115     TransData[GET_OFF(ObjHeader+ObjRef) + DestOff] = Val;// write to trans.body
116   else                                                   // Otherwise
117     PersiData[GET_OFF(ObjHeader+ObjRef) + DestOff] = Val;// write to pers.body
118   updateJPC();
119 }
```

Fig. 3. `firewall` and `bastore` functions with their ACSL contracts.

tomatic reasoning for bit-level operations and heterogeneous casts rewritten with arithmetic operations.

5 Predicates, Lemmas and Scripts

This section details Fig. 2 and 4, as well as function contracts in Fig. 3.

The predicate on lines 68–71 of Fig. 2 states that the object headers of allocated objects do not change between labels (program points) `L1,L2`. The predicate on lines 74–80 states that the companion model does not change between labels `L1,L2` for objects that existed at label `L1`, but new objects can have been allocated. As we said, we do not consider object deletion.

The C code of the dispatch loop in Fig. 4 reads the next opcode and chooses the opcode function to be called. The code of `bastore` function was presented in Sec. 4. We assume that `baload` is a similar function for reading a value, and `other_opcode` illustrates other opcodes. For simplicity, the Java stack and Java reference resolution are not modeled in this example, and the necessary arguments of `bastore` and `baload` are read as non-deterministic values (lines 185–186, 188–189). Here again, *to avoid an existential quantifier, we use a ghost variable* `gCurObj` to represent the index of the object in our companion model. We assume for simplicity as a precondition of `bastore` (cf. line 102 in Fig. 3)

```
120 //Lines 121-170 contain functions baload, other_opcode and a contract of main_loop
    ...
171 void main_loop(){
172 /*@
173   loop invariant vhm: valid_heap_model;
174   loop invariant vcm: valid_code_model;
175   loop invariant oh:  object_headers_intact{LoopEntry,Here};
176   loop invariant mmf: mem_model_footprint_intact{LoopEntry,Here};
177   loop invariant no:  gNumObjs >= \at(gNumObjs,LoopEntry);
178   loop assigns gNumObjs, ObjHeader[0..(SEGM_SIZE-1)],
179     gIsTrans[0..(MAX_OBJS-1)], gHeadStart[0..(MAX_OBJS-1)],
180     gDataStart[0..(MAX_OBJS-1)], gDataEnd[0..(MAX_OBJS-1)], gCurObj, JCC,
181     PersiData[0..(SEGM_SIZE-1)], TransData[0..(SEGM_SIZE-1)], JPC, gJPCOff;
182 */
183   while(1){
184     if(*JPC == 1)                             // Assume code 1 is for BASTORE
185       /*@ ghost gCurObj=get_gu4(); */         // Assume arbitrary object index and
186       bastore(get_u4(),get_u4(),get_u1()); // header offset, body offset, value
187     else if(*JPC == 2)                        // Assume code 2 is for BALOAD
188       /*@ ghost gCurObj=get_gu4(); */         // Assume arbitrary object index and
189       baload(get_u4(),get_u4());              // header offset, body offset
190     else if(*JPC == 3)                        // Assume code 3 is for exit
191       return;
192     else                                      // Other opcodes
193       other_opcode();
194   }
195 }
```

Fig. 4. The dispatch loop and its ACSL contract.

that the object reference is a valid object, therefore, it has an index in the companion model. The **admit** keyword[1] indicates that this annotation is assumed without proof. Heap and code validity are both pre- and postconditions (lines 100–101, 105–106). Line 103 indicates variables that the function is allowed to modify. Line 107 ensures security property $\mathcal{G}^{\text{head}}_{\text{integ}}$ of Sec. 3. Line 104 indicates base address(es) of memory locations pointer `JPC` can be assigned to refer to. This information is needed for a recent alias analysis in WP for pointers modified inside the function (see [3, Sec. 3.6]). Line 108 is explained below. The contract of `firewall` is straightforward.

The loop contract of the dispatch loop is similar to the contract of `bastore`, but also allows modifications of current context `JCC`, an allocation of new objects (line 177) and, therefore, modifications of the companion ghost model (lines 178–180). For simplicity, in properties on lines 175–177 we compare the state after each iteration (label `Here`, taken by default) to the start of the loop (label `LoopEntry`), i.e. to the objects allocated before the loop. To cover all objects allocated before the current iteration, similar properties comparing the start and the end of an iteration can be specified in the loop body.

The dispatch loop iterations can modify a large part of the memory in the real-life code, that decreases the capacity of automatic proof. To facilitate the proof, we introduce the memory model footprint preservation property for previously allocated objects (lines 73–80). It is used to state several preservation lemmas for complex properties (like on lines 82–84 in Fig. 2). They facilitate the automatic proof for the real-life code: non-modification of some variables is easier to prove

[1] It was recently added in the 23.0beta and 23.0 releases of Frama-C; it should be removed if an earlier release is used.

automatically than more complex properties, and helps to automatically deduce more complex properties using lemmas.

When automatic proof does not work, the interactive proof editor of WP [3] is very helpful to indicate some first proof steps—that can be recorded in a proof script—to help the automatic prover to finish the proof. For instance, for the lemma in Fig. 2, Alt-Ergo cannot perform the proof. The preservation of the loop invariant `mmf` in the dispatch loop is another unproved goal. Proof scripts can help to finish the proof. Typically, a script in our project includes unfolding some predicate definitions, splitting some proof goals and instantiating universally quantified goals with specific values.

Another issue was related to bit-level lemmas, not proved with Alt-Ergo, e.g.:

```
/*@ lemma dn: \forall u1 c; (c & 0x04)==0 && (c & 0x08)!=0 ==> (c & 0x0C)==0x08;*/
```

On our request, WP developers added new tactics so that now such lemmas are successfully proved after a few clics in the interactive proof editor of WP.

One scalability issue we met was related to the simplification engine Qed of WP: it could take about 40 min per property because of a very high number of branches (for 185 opcodes) in the dispatch loop. The solution we used was to deactivate some Qed simplifications (with option `-wp-no-pruning`) and to rewrite a long dispatch loop equivalently with shorter functions. *Another kind of code transformation was necessary to rewrite longjmp/setjmp instructions* present in the code but not yet supported by Frama-C. Apart from these two cases of transformations and a minor rewriting for heterogeneous pointer casts (see Sec.4), *the real-life code was proved as is, without other code transformations.*

Overall, a careful combination of preservation properties, lemmas and interactive proof scripts helped us to successfully finish the proof.

6 Verification of Security Properties with MetAcsl

We saw that some security properties like $\mathcal{G}^{\text{head}}_{\text{integ}}$ can be specified in ACSL as an invariant property maintained by relevant functions and directly proved by WP. For other properties, like $\mathcal{G}^{\text{data}}_{\text{integ}}$ and $\mathcal{G}^{\text{data}}_{\text{conf}}$, it is not possible. Confidentiality properties cannot be currently verified by WP because there is no way supported by the tool to specify which variables (or memory locations) can be read and under which precise conditions. But even for an integrity property $\mathcal{G}^{\text{data}}_{\text{integ}}$, it is not easy to specify that modifications can only occur when allowed. The current context of the smart card can be changed (under certain conditions, that must of course be specified and verified as well). Hence $\mathcal{G}^{\text{data}}_{\text{integ}}$ cannot be specified as preservation of values during the dispatch loop: object data *can be modified* if the current context `JCC` was legally changed to the object owner. As various involved variables (in this example, object data and `JCC`) are changed in different functions under different specific conditions, it is extremely difficult to achieve a global view of what is really specified and verified. To solve this issue, we use the recent metaproperty-based approach and the MetAcsl plugin [23–25].

Figure 5 shows two metaproperties expressing $\mathcal{G}^{\text{data}}_{\text{integ}}$ and $\mathcal{G}^{\text{data}}_{\text{conf}}$ for persistent objects. Lines 198, 203 provide a name, the set of target functions (here, all functions) and the context—the situations in which it must apply. The first

```
197 /*@ // === Metaproperties: persistent object data written/read only by owner ===
198 meta \prop,\name(meta_persi_objects_integrity),\targets(\ALL),\context(\writing),
199   ( \forall integer i; 0 <= i < gNumObjs && !gIsTrans[i] &&
200   ObjHeader[gHeadStart[i] + 0] != JCC ==>
201   \separated(\written,PersiData+(gDataStart[i]..gDataEnd[i])) );
202
203 meta \prop,\name(meta_persi_objects_confident),\targets(\ALL),\context(\reading),
204   ( \forall integer i; 0 <= i < gNumObjs && !gIsTrans[i] &&
205   ObjHeader[gHeadStart[i] + 0] != JCC ==>
206   \separated(\read,PersiData+(gDataStart[i]..gDataEnd[i])) ); */
```

Fig. 5. Metaproperties for persistent object data integrity/confidentiality.

metaproperty has a writing context and applies whenever a variable is written. It means that whenever a variable (or memory location) is written, the predicate on lines 199–201 must hold, where `\written` refers to the written location. In other words, the written location must be *separated* (that is, disjoint) from the data of any persistent object if the current context is not its owner, as required by $\mathcal{G}^{\text{data}}_{\text{integ}}$. Similarly, the second metaproperty states that every read location must be separated (that is, disjoint) with the data of any persistent object if the current context is not its owner, as required by $\mathcal{G}^{\text{data}}_{\text{conf}}$. Metaproperties for transient objects are expressed similarly.

MetAcsl translates metaproperties into assertions at each relevant program point. For example, for the first metaproperty, an assertion of the provided predicate will be added before each writing operation, where `\written` will be replaced by the address of the written location. Those assertions can then be verified by WP. If all assertions are proved, the metaproperty is proved. Notice that these metaproperties directly ensure the security properties for all currently allocated objects (not only those allocated before the loop as for a loop invariant) since the predicate is inserted and evaluated at each relevant program location.

On our request, MetAcsl developers added an extremely useful feature: to translate a metaproperty into checks rather than asserts. In ACSL, the proof of a **`check`** is attempted, but it is not kept in the proof context for the following properties, contrary to an **`assert`**, that is proved and kept in the context. *Thanks to the translation of metaproperties into checks that do not overload proof contexts, the metaproperty-based approach scales very well,* despite a great number of generated annotations.

7 Specification Architecture and Effort

Maintenance issue. ACSL annotations may become pervasive, difficult to track and to maintain, especially when the verification scope is meant to be extended. They require a *careful organization to ensure specification traceability and maintainability.* Function contracts are placed in header files with function declarations, security properties are grouped in a separate header file, etc.

Macros to define common contracts. We leverage the pre-processing of C macros to organize a large part of function and loop contracts as macros. They are used to define common properties that occur in several contracts. For instance, one macro is used to gather all postconditions that apply to several opcode functions. Other examples of macros are some common preconditions, or common assigns

				User-provided ACSL					MetAcsl			RTE	
				Manual effort			Auto. prep.		Man.	Auto. trans.		Auto. gen.	
Code subset	$\#_{\text{opc}}$	$\#_{\text{fun}}$	$\#_{\text{C}}$	$\#_{\text{ghost}}$	$\#^{man}_{\text{ACSL}}$	$\frac{\#^{man}_{\text{ACSL}}}{\#_{\text{C}}}$	$\#^{\text{pp}}_{\text{ACSL}}$	$\frac{\#^{\text{pp}}_{\text{ACSL}}}{\#_{\text{C}}}$	$\#_{\text{P}}$	$\#^{\text{meta}}_{\text{ACSL}}$	$\frac{\#^{\text{meta}}_{\text{ACSL}}}{\#_{\text{C}}}$	$\#^{\text{rte}}_{\text{ACSL}}$	$\frac{\#^{\text{rte}}_{\text{ACSL}}}{\#_{\text{C}}}$
Bastore	1	11	540	70	2,814	**5.21**	2,770	**5.12**	29	57,362	106.22	514	0.95
Sample 1	4	19	783	94	3,153	4.08	3,526	4.50	29	84,698	108.17	734	0.93
Sample 2	11	36	1,201	97	3,939	3.27	5,787	4.81	34	117,191	97.57	897	0.74
All	185	391	7,014	162	**12,432**	**1.77**	**35,480**	**5.05**	**36**	**396,603**	56.54	2,290	0.32

Table 1. Specification effort for the real-life code.

clauses. *Macros reduce redundancy in specifications and facilitate updates and maintenance.* Note that specific clauses can still be added to opcode function contracts if required.

Macros to reduce the VM to particular opcodes. We realize a rich set of macros to select a consistent minimal part of the C code and ACSL annotations for verification of properties for some code subsets: one opcode or a sample of opcodes. On a large project, running the proof on such a code subset is handy for getting faster results for a subset of opcodes and for proof debugging purposes.

Inventory of ACSL annotations. We distinguish several kinds of ACSL annotations in our project, depending whether they have been manually written by the user or automatically generated by MetAcsl, RTE or WP plugins:

– User-provided annotations: ACSL predicates and lemmas, function contracts, loop contracts, proof-guiding assertions and metaproperties.
– Automatically generated annotations, produced
  • by MetAcsl plugin according to user-defined metaproperties (cf. Sec. 6);
  • by RTE plugin to prevent undefined behaviors (cf. Sec. 2);
  • by WP plugin to detect ACSL specification inconsistencies, called *smoke tests* (optional) [3, Sec. 2.3.5]. Basically, they check if `false` is provable.

Target JCVM code and code subsets. We verify the JCVM code that interprets 185 standard opcodes of the JavaCard platform [22]. To show our proof results, we consider four incrementally increasing subsets of C code: (i) the smallest subset required to interpret the bastore opcode; (ii) a subset required to interpret 4 opcodes (Sample 1); (iii) a subset for 11 opcodes (Sample 2); (iv) the whole code with all 185 opcodes (All). Columns $\#_{\text{opc}}$, $\#_{\text{fun}}$ and $\#_{\text{C}}$ of Table 1 show, respectively, the number of opcodes, functions and lines of C code in each subset. The whole code (All) submitted to our deductive verification contains 7,014 lines of C code. In addition to the 391 functions that are fully proved, it contains 23 stub functions (only specified in ACSL but not verified) to delimit the considered verification scope. They mainly include some specific functions exploring the hierarchy of classes and interfaces, functions for particular exception handler operations, and memory address resolution functions.

User-provided ACSL annotations. The core of our formal specification consists of manually written ACSL annotations. We show separately the amount (in lines of code (loc)) of ghost code and other annotations (except metaproperties), resp., in columns $\#_{\text{ghost}}$ and $\#_{\mathsf{ACSL}}^{man}$ of Table 1. The amount of ghost code is relatively small (162 loc for All) compared to the project size. It mainly contains declaring and updating ghost variables. Column $\#_{\mathsf{ACSL}}^{\text{pp}}$ shows the amount of ACSL annotations after preprocessing of the macros defining common contracts. We observe that the ratio of expanded ACSL annotations with respect to the C code (column $\#_{\mathsf{ACSL}}^{\text{pp}}/\#_{\text{C}}$) is between 4.5 and 5.12 for all subsets. However the ratio of user-provided ACSL annotations with respect to the C code (column $\#_{\mathsf{ACSL}}^{man}/\#_{\text{C}}$) shrinks drastically from 5.05 for Bastore to 1.77 for All. Indeed, macros help to reduce the number of lines of user-provided ACSL annotations from $35,480$ to $12,432$ for the whole C code. Hence, *the benefit of macros monotonically increases with the increase of redundant ACSL annotations for larger code subsets,* and the ratio of manually written annotations for the whole code becomes very reasonable. Macros save a lot of effort and enhance the readability and traceability of ACSL annotations. Notice though that this observation can be specific to our project, where some groups of opcodes have similar contracts.

MetAcsl annotations. Column $\#_P$ in Table 1 gives the number of metaproperties (that varies since some of them cover different sets of functions). Column $\#_{\mathsf{ACSL}}^{\text{meta}}$ shows the number of lines in annotations automatically generated from them by MetAcsl. Despite a very high number of ACSL annotations generated ($396,603$ loc for All, which is 56.54 times the original C code size), we had to write only 36 metaproperties with approximately 480 lines of ACSL. Note that the ratio of the size of generated ACSL annotations w.r.t the original code ($\#_{\mathsf{ACSL}}^{\text{meta}}/\#_{\text{C}}$) decreases from 106.22 for Bastore to 56.54 for All, in particular, since All includes many simple, short opcode functions, for which MetAcsl generates less annotations.

RTE annotations. The ratio of the size of generated RTE annotations w.r.t. the whole C code ($\#_{\mathsf{ACSL}}^{\text{rte}}/\#_{\text{C}}$) is 0.32 (cf. Table 1), which is smaller compared to user-provided annotations (5.05) and MetAcsl-generated annotations (56.54). Is decreases for the same reason as for metaproperties.

8 Proof Results and Lessons Learned

8.1 Proof results

Table 2 depicts proof results obtained by running Frama-C 22.0 (Titanium) on an Ubuntu virtual machine. It was used under VirtualBox on a host desktop PC running Windows 10 with Intel(R) core(TM) i7 CPU @ 2.00GHz processor and 32.0 GB RAM. 8 processors and 24GB were allocated to the virtual machine. Frama-C was run with option `-wp-par 8` to optimally use the 8 allocated processors and a timeout for provers set to 100 seconds (option `-wp-timeout 100`). We used the Alt-Ergo solver, version 2.3.2. Overall, 52,198 proof goals have been proven within 3h28m07s.

		User-provided ACSL	MetAcsl	RTE	Total	
Code subset	Prover	#Goals	#Goals	#Goals	#Goals	Time
Bastore	Qed	1,019	3,304	106	4,429 (**77.92%**)	0h47m45s
	Script	78	131	1	210 (3.69%)	0h11m12s
	SMT	305	590	148	1,043 (18.35%)	0h17m23s
	All	1,402 (24.67%)	4,025 (**70.81%**)	255 (4.48%)	**5,684**	**0h49m37s**
Sample 1	Qed	1,491	5,037	120	6,648 (**79.76%**)	1h00m49s
	Script	111	149	7	267 (3.20%)	0h13m41s
	SMT	437	784	199	1,420 (17.03%)	0h28m24s
	All	2,039 (24.46%)	5,970 (**71.63%**)	326 (3.91%)	**8,335**	**0h59m59s**
Sample 2	Qed	2,413	6,884	126	9,423 (**79.43%**)	1h04m33s
	Script	144	257	20	421 (3.55%)	0h18m15s
	SMT	682	1,088	249	2,019 (17.01%)	0h37m01s
	All	3,239 (27.30%)	8,229 (**69.36%**)	395 (3.33%)	**11,863**	**1h09m47s**
All	Qed	18,925	22,361	168	41,454 (**79.42%**)	2h58m15s
	Script	330	212	30	572 (1.1%)	0h44m48s
	SMT	4,683	4,588	902	10,173 (19.49 %)	2h36m18s
	All	23,938 (**45.85%**)	27,435 (**52.55%**)	1,117 (2.13%)	**52,198**	**3h28m07s**

Table 2. Proof results for the real-life code.

Results per prover. The internal simplifier engine Qed of WP proves most proof goals (around 79%) with an average time of 257ms per proved goal. The maximum time spent by Qed to prove one goal is 10.9s. The SMT solver is able to discharge around 20% of proof goals with an average time of 974ms per proved goal. The maximum time spent to prove one goal is 1m2s. Last but not least, scripts prove the rest of the goals with an average time of 4s699ms. The majority of scripts are necessary, but some scripts are introduced for time saving purpose, when a script-based proof is faster and more stable with a fixed timeout of 100s.

Results per annotation kind. As discussed in Sect. 7, most lines of ACSL are generated by MetAcsl. Each ACSL annotation is usually encoded on several lines and several proof goals may be generated by WP for each ACSL annotation. This partially explains the difference of proportions between Tables 1 and 2: in Table 1, the ratio of the number of generated ACSL lines for MetAcsl w.r.t. the number of preprocessed user-provided ACSL lines ranges between 20 times for Bastore subset (57,362 vs 2,770) and 11 times for All (396,603 vs 35,480). However, in Table 2, the ratio of the number of generated goals for MetAcsl w.r.t. the number of goals for user-provided ACSL ranges between 3 times for Bastore subset (4,025 vs 1,402) and 1.1 times for All (27,435 vs 23,938). For the same reasons as explained in Sect. 7, the percentage of MetAcsl goals decreases from 70.81% for Bastore subset to 52.55% for All.

Overall results scalability. Results of Table 2 show that the proof scales well with an increasing number of proof obligations. Whereas the number of proof goals increases ten times from bastore subset to all opcodes subset, the proof time increased only four times. This is thanks to parallelisation of goal proofs in Frama-C. The distribution of proved goals over provers and ACSL annotation kinds is given in Table 2. Although scripts prove only 1% of goals for the whole program, they are very important to achieve a complete proof. They are also important to get a complete proof verdict in a reasonable time while the specification task is in progress and the verification engineers wait for the proof results. Without the 572 scripts, proved in 44m48s, 16 extra hours will be necessary to get a complete proof verdict with a timeout of 100s set for external provers.

8.2 Lessons Learned

Successful industrial application. Our application of deductive verification on a large industrial C program shows that formal verification of real-life industrial code has become feasible today. The proof of real-life code in our project requires a careful combination of several ingredients: companion ghost code, preservation properties, lemmas and proof scripts. 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. The majority of proof goals (almost 99%) are proved automatically by the Qed simplification engine of WP and an automatic SMT solver. The remaining goals are successfully proved with proof scripts. MetAcsl proved to offer a convenient and efficient technique for specification and verification of security-related properties. An efficient support from the tool developers during the whole project was essential for its success. Some anomalies were reported and fixed, and several new features were requested and implemented. Examples of such features include the implementation of check-and-forget versions of all annotations (i.e. verified but not kept in the proof context), their usage for annotations generated by MetAcsl, as well as precise generation of memory model hypotheses necessary for a sound proof [3, Sec. 3.6].

Further improvements. Creating and updating proof scripts in WP is a time-consuming task. Scripts are very sensitive to specification changes and require to be updated accordingly. Designing and applying custom, project-specific strategies—possible in Frama-C—would at least partially address this issue and save efforts. Another issue we faced during this project is related to properties mixing casts and arithmetic operations between different integer types. Lemmas allowing to prove such properties should be either systematically activated in the tool or made applicable on request. Further improvements in the tool seem to be necessary to perform a proof of large programs, in particular, mixing complex logic properties and low-level operations. One future work direction is the development of collaborative memory models, capable to reason with different memory models on various parts of code (e.g. with and without low-level operations) and to soundly combine the results. Integrating more abstract levels of reasoning into source-code based deductive verification is also an interesting work perspective. Another work direction concerns a deeper proof parallelization. In our case study, doubling the number of processors dedicated to the proof computation from 8 to 16 cores does not seem to bring any benefit on the proof efficiency today since some parts of WP are not parallelized. A more efficient proof parallelization would facilitate industrial applications of the tool. Finally, scalability issues of the Qed simplification engine on very long functions—that we avoided by a code rewriting—should be further investigated.

9 Related Work

JavaCard related formal verification. A classical approach of applying formal verification on JavaCard platform consists in building a high-level formal model of target sub-modules. Several case studies have adopted this approach. An executable formal semantics of the JCVM and BCV is proposed in [2] with 15,000 lines of Coq scripts. Authors of [17] describe a refinement-based approach, using 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 a considerable effort to be justified because of the gap between the formal model and the source code. In our case, all specified features and properties are expressed as ACSL annotations directly on source code. An operational semantics of a language that models the JCVM behavior is proposed in [11,26]. It includes the basic structures needed to model object ownership and the JavaCard firewall. This is analogous to our formal specification. In addition, we perform a full proof of target security properties on a real-life JCVM implementation. Among tools devised and/or used for the purpose of providing formal guarantees about JavaCard platform security properties we can list: Key [16], KRAKATOA [15] and Caduceus [1].

Other success stories of deductive verification. Various verification case studies of real-life software have emerged in the last two decades, where code was annotated and verified, and often bugs were found [13]. A recent case study [18] presented formal verification of industrial safety-critical software for a traffic tunnel control system verification based on VerCors tool. Authors of [9] provide a feedback on their experience of using ACSL and Frama-C on a real-world example. Other case studies based on Frama-C present formal verification of kLIBC, a minimalistic C library [6], a unit-proof of almost 3315 C functions of an avionics software [5] and a verified RTE-free X.509 parser [10]. Proved properties tend to be shallower as the code becomes of a lower-level nature. In our work, we took on the challenge and managed to prove global critical security properties on large real-life C code.

10 Conclusion

In this paper, we have presented a formal verification case study fully realized in an industrial context for a certification purpose. It contributes to collect and publish best practices and specification patterns in formal verification. We believe this work will set up a new state of the art for applying deductive verification to prove global security properties directly on large security-critical code. We report detailed specification statistics and proof results that measure the specification effort and proof scalability. The reported lessons learned from this project open the door for further methodology and tool enhancements. As a future work, we plan to introduce deductive verification in a sustainable continuous integration process as both the code and its formal specification share the same codebase.

Acknowledgement. The authors thank the Frama-C team members for their reliable and efficient support. Many thanks to the anonymous reviewers for their helpful comments.

References

1. Andronick, J., Chetali, B., Paulin-Mohring, C.: Formal verification of security properties of smart card embedded source code. In: International Symposium of Formal Methods (FM 2005). LNCS, vol. 3582, pp. 302–317. Springer (2005). https://doi.org/10.1007/11526841_21
2. Barthe, G., Dufay, G., Jakubiec, L., Serpette, B.P., de Sousa, S.M.: 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). LNCS, vol. 2028, pp. 302–319. Springer (2001). https://doi.org/10.1007/3-540-45309-1_20
3. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual (2020), `https://frama-c.com/download/frama-c-wp-manual.pdf`
4. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.16 (2020), `http://frama-c.cea.fr/acsl.html`
5. Brahmi, A., Carolus, M.J., Delmas, D., Essoussi, M.H., Lacabanne, P., Lamiel, V.M., Randimbivololona, F., Souyris, J.: Industrial use of a safe and efficient formal method based software engineering process in avionics. In: Embedded Real Time Software and Systems (ERTS 2020) (2020)
6. Carvalho, N., da Silva Sousa, C., Sousa Pinto, J., Tomb, A.: Formal verification of kLIBC with the WP Frama-C plug-in. In: NASA Formal Methods (NFM 2014). LNCS, vol. 8430, pp. 343–358. Springer (2014). https://doi.org/10.1007/978-3-319-06200-6_29
7. Conchon, S., et al.: The Alt-Ergo automated theorem prover., `http://alt-ergo.lri.fr`
8. Correnson, L.: Qed. Computing what remains to be proved. In: NASA Formal Methods (NFM 2014). LNCS, vol. 8430, pp. 215–229. Springer (2014). https://doi.org/10.1007/978-3-319-06200-6_17
9. Dordowsky, F.: An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. Electronic Proceedings in Theoretical Computer Science **187**, 28–41 (2015). https://doi.org/10.4204/EPTCS.187.3
10. Ebalard, A., Mouy, P., Benadjila, R.: Journey to a RTE-free X.509 parser. In: Symposium sur la sécurité des technologies de l'information et des communications (SSTIC 2019) (2019), `https://www.sstic.org/media/SSTIC2019/SSTIC-actes/journey-to-a-rte-free-x509-parser/SSTIC2019-Article-journey-to-a-rte-free-x509-parser-ebalard_mouy_benadjila_3cUxSCv.pdf`
11. Éluard, M., Jensen, T., Denne, E.: An operational semantics of the Java Card firewall. In: Smart Card Programming and Security, pp. 95–110. Springer (2001). https://doi.org/10.1007/3-540-45418-7_9
12. Filliâtre, J., Paskevich, A.: Why3 – where programs meet provers. In: the 22nd European Symp. on Programming (ESOP 2013). LNCS, vol. 7792, pp. 125–128. Springer (2013)
13. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science – State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18

14. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. pp. 1–37 (2015). https://doi.org/10.1007/s00165-014-0326-7
15. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. The Journal of Logic and Algebraic Programming **58**(1-2), 89–106 (2004). https://doi.org/10.1016/j.jlap.2003.07.006
16. Mostowski, W.: Fully verified Java Card API reference implementation. In: 4th International Verification Workshop in connection with CADE-21. CEUR Workshop Proceedings, vol. 259. CEUR-WS.org (2007), `http://ceur-ws.org/Vol-259/paper12.pdf`
17. Nguyen, Q.H., Chetali, B.: Certifying native java API by formal refinement. In: 7th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications (CARDIS 2006), LNCS, vol. 3928, pp. 313–328. Springer (2006). https://doi.org/10.1007/11733447_23
18. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: 15th International Conference on Integrated Formal Methods (IFM 2019), LNCS, vol. 11918, pp. 418–436. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_23
19. Oracle: Java Card 2.2 Off-Card Verifier, Whitepaper. Tech. rep., Oracle (2002)
20. Oracle: Java Card System – Open Configuration Protection Profile, Version 3.1. Tech. rep., Oracle (2020), `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/ReportePP/pp0099V2b_pdf.pdf;jsessionid=6C3F5A7FB5FA0D928A1C310C1C0EF1CE.internet462?__blob=publicationFile&v=1`
21. Oracle: Java Card Platform: Runtime Environment Specification, Classic Edition, Version 3.1. Tech. rep., Oracle, Oracle (feb 2021), `https://docs.oracle.com/javacard/3.1/related-docs/JCCRE/JCCRE.pdf`
22. Oracle: Java Card Platform: Virtual Machine Specification, Classic Edition, Version 3.1. Tech. rep., Oracle, Oracle (feb 2021), `https://docs.oracle.com/javacard/3.1/related-docs/JCVMS/JCVMS.pdf`
23. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Gall, P.L.: MetAcsl: Specification and verification of high-level properties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019). LNCS, vol. 11427, pp. 358–364. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_22
24. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: Tame your annotations with MetAcsl: Specifying, testing and proving high-level properties. In: 13th International Conference on Tests and Proofs (TAP 2019), Held as Part of the Third World Congress on Formal Methods (FM 2019). LNCS, vol. 11823, pp. 167–185. Springer (2019). https://doi.org/10.1007/978-3-030-31157-5_11
25. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: 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). pp. 54–67. IEEE (2021). https://doi.org/10.1109/FormaliSE52586.2021
26. Siveroni, I.A.: Operational semantics of the Java Card Virtual Machine. The Journal of Logic and Algebraic Programming **58**(1-2), 3–25 (jan 2004). https://doi.org/10.1016/j.jlap.2003.07.003

A Appendix: Supplementary Material

A.1 Complete illustrative example

Figures 6, 7, 8, 9 give the complete illustrative example of the JCVM code annotated in ACSL. It was tested with Frama-C 23.0, Why3 1.4.0, Alt-Ergo 2.2.0. The commands to run the proof with or without MetAcsl are given at the end of the file.

Three proof goals remain unproven with these versions and require scripts. One of them is the lemma in Fig. 2. The second one is the preservation of the loop invariant `mmf` in the dispatch loop. Finally, lemma `dn` is the third goal. We prepared scripts to illustrate that they can be proved using proof scripts.

```
typedef unsigned char u1; typedef unsigned short u2; typedef unsigned int u4;

// === Code model and current Java context ===
#define CODE_SIZE 10000
u1  Code[CODE_SIZE], *JPC; // Java code area and Java program counter
//@ ghost u4 gJPCOff;      // JPC offset in code area
u1  JCC;                   // Current Java context

// === Heap model ===
#define SEGM_SIZE 10000
#define MAX_OBJS  500
u1 ObjHeader[SEGM_SIZE];   // Object headers area
//Header(8B),Bytes:Contents: 0:Owner,1:Flags,2-3:Class,4-5:BodyOff,6-7:BodySize
#define GET_OWN(addr)  ( *((u1*)addr + 0) )
#define GET_FLAG(addr) ( *((u1*)addr + 1) )
#define GET_OFF(addr)  ( (u2)((*((u1*)addr + 4))*256 + *((u1*)addr + 5)) )
#define GET_SIZE(addr) ( (u2)((*((u1*)addr + 6))*256 + *((u1*)addr + 7)) )
u1 PersiData[SEGM_SIZE];   // Persistent objects data area
u1 TransData[SEGM_SIZE];   // Transient  objects data area

/*@ ghost // === Companion ghost memory view ===
  u4 gNumObjs;             // Number of allocated objects
  u1 gIsTrans  [MAX_OBJS]; // Nonzero for transient object
  u4 gHeadStart[MAX_OBJS]; // Start offset of object header
  u4 gDataStart[MAX_OBJS]; // Start offset of object data
  u4 gDataEnd  [MAX_OBJS]; // End offset of object data
  u4 gCurObj;  */          // Currently considered object number

/*@ // === Validity predicates ===
predicate valid_code_model = 0 <= gJPCOff < CODE_SIZE &&
  JPC == &Code[gJPCOff];
predicate valid_heap_model =
  0 <= gNumObjs <= MAX_OBJS &&
// headers of allocated objects are within ObjHeader segment
  (\forall integer i; 0 <= i < gNumObjs ==>
    0 <= gHeadStart[i] <= SEGM_SIZE - 8 ) &&
// no overlapping between headers (each header has 8 bytes)
  (\forall integer i,j; 0 <= i < j < gNumObjs ==>
    (gHeadStart[i] >= gHeadStart[j]+8 || gHeadStart[j] >= gHeadStart[i]+8) ) &&
// IsTrans[i] encodes if i-th object's transient bit is set
  (\forall integer i; 0 <= i < gNumObjs ==>
    ( gIsTrans[i] <==> (GET_FLAG(ObjHeader+gHeadStart[i]) & 0x08) ) ) &&
// data of allocated objects is within a data segment
  (\forall integer i; 0 <= i < gNumObjs ==>
    gDataStart[i] == GET_OFF(ObjHeader+gHeadStart[i]) &&
    gDataEnd[i] == gDataStart[i] + GET_SIZE(ObjHeader+gHeadStart[i]) - 1 &&
    0 <= gDataStart[i] < gDataEnd[i] < SEGM_SIZE ) &&
// no overlapping between persistent object data
  (\forall integer i,j; 0<=i<j<gNumObjs && !gIsTrans[i] && !gIsTrans[j] ==>
    (gDataStart[i] > gDataEnd[j] || gDataStart[j] > gDataEnd[i]) ) &&
// no overlapping between transient object data
  (\forall integer i,j; 0 <= i < j < gNumObjs && gIsTrans[i] && gIsTrans[j] ==>
    (gDataStart[i] > gDataEnd[j] || gDataStart[j] > gDataEnd[i]) ); */
```

Fig. 6. Complete illustrative example of the JCVM code, part 1/4.

```
 54
 55 // Lines 56-66 give declarations of functions updateJPC, get_u1, get_u4, get_gu4.
 56 /*@
 57   requires vcm: valid_code_model;
 58   assigns  JPC, gJPCOff;
 59   assigns  JPC \from &Code[0],JPC;
 60   ensures  vcm: valid_code_model; */
 61 void updateJPC(void);
 62
 63 /*@ assigns \nothing; */  u1 get_u1(void);
 64 /*@ assigns \nothing; */  u4 get_u4(void);
 65 /*@ ghost  /@ assigns \nothing; @/ u4 get_gu4(void); */
 66
 67 /*@ // === A security property: object headers remain intact ===
 68 predicate object_headers_intact{L1, L2} =
 69   \forall integer i, off; 0 <= i < \at(gNumObjs,L1) &&
 70     \at(gHeadStart[i],L1) <= off < \at(gHeadStart[i],L1) + 8 ==>
 71     \at(ObjHeader[off],L1) == \at(ObjHeader[off],L2);
 72
 73 // === Memory footprint predicate and lemma example ===
 74 predicate mem_model_footprint_intact{L1,L2} =
 75   \at(gNumObjs,L1) <= \at(gNumObjs,L2) &&
 76   ( \forall integer i; 0 <= i < \at(gNumObjs,L1) ==>
 77     \at(gIsTrans[i],L1) == \at(gIsTrans[i],L2) &&
 78     \at(gHeadStart[i],L1) ==\at(gHeadStart[i],L2) &&
 79     \at(gDataStart[i],L1) ==\at(gDataStart[i],L2) &&
 80     \at(gDataEnd[i],L1) ==\at(gDataEnd[i],L2) );
 81
 82 lemma vhm_preserved{L1,L2}: mem_model_footprint_intact{L1,L2} &&
 83   object_headers_intact{L1,L2} && valid_heap_model{L1} &&
 84   \at(gNumObjs,L1) == \at(gNumObjs,L2) ==> valid_heap_model{L2}; */
 85
 86 /*@
 87   requires vhm: valid_heap_model;
 88   requires 0 <= gCurObj < gNumObjs && ObjRef == gHeadStart[gCurObj];
 89   assigns \nothing;
 90   ensures \result <==> ( GET_OWN(ObjHeader+ObjRef) == JCC &&
 91     gDataStart[gCurObj] + DestOff <= gDataEnd[gCurObj] );
 92 */
 93 u1 firewall(u4 ObjRef, u4 DestOff){
 94   if (GET_OWN(ObjHeader+ObjRef) == JCC && DestOff < GET_SIZE(ObjHeader+ObjRef))
 95     return 1;
 96   return 0;
 97 }
 98
 99 /*@
100   requires vhm: valid_heap_model;
101   requires vcm: valid_code_model;
102   admit requires 0 <= gCurObj < gNumObjs && ObjRef == gHeadStart[gCurObj];
103   assigns  PersiData[0..(SEGM_SIZE-1)],TransData[0..(SEGM_SIZE-1)],JPC,gJPCOff;
104   assigns  JPC \from &Code[0]; // possible base address
105   ensures  vhm: valid_heap_model;
106   ensures  vcm: valid_code_model;
107   ensures  oh:  object_headers_intact{Pre,Post};
108   ensures  mmf: mem_model_footprint_intact{Pre,Post};
109 */
110 void bastore(u4 ObjRef, u4 DestOff, u1 Val)
111 {
112   if( ! firewall(ObjRef,DestOff) )                       // Check access and
113     return;                                              // exit if forbidden
114   if( GET_FLAG(ObjHeader+ObjRef) & 0x08 )                // If trans. bit set,
115     TransData[GET_OFF(ObjHeader+ObjRef) + DestOff] = Val;// write to trans.body
116   else                                                   // Otherwise
117     PersiData[GET_OFF(ObjHeader+ObjRef) + DestOff] = Val;// write to pers.body
118   updateJPC();
119 }
```

Fig. 7. Complete illustrative example of the JCVM code, part 2/4.

```
120 //Lines 121-170 contain functions baload, other_opcode and a contract of main_loop
121 /*@
122   requires vhm: valid_heap_model;
123   requires vcm: valid_code_model;
124   admit requires 0 <= gCurObj < gNumObjs && ObjRef == gHeadStart[gCurObj];
125   assigns  JPC, gJPCOff;
126   assigns  JPC \from &Code[0]; // possible base address
127   ensures  vhm: valid_heap_model;
128   ensures  vcm: valid_code_model;
129   ensures  oh:  object_headers_intact{Pre,Post};
130   ensures  mmf: mem_model_footprint_intact{Pre,Post};
131 */
132 void baload(u4 ObjRef, u4 DestOff)
133 {
134   u1 Value;
135   if( ! firewall(ObjRef,DestOff) )                        // Check access and
136     return;                                               // exit if forbidden
137   if( GET_FLAG(ObjHeader+ObjRef) & 0x08 )                 // If transient flag
138     Value = TransData[GET_OFF(ObjHeader+ObjRef) + DestOff];// set transient body
139   else                                                    // Otherwise
140     Value = PersiData[GET_OFF(ObjHeader+ObjRef) + DestOff];// set persistent body
141   updateJPC();
142 }
143
144 /*@
145   requires vhm: valid_heap_model;
146   requires vcm: valid_code_model;
147   assigns  gNumObjs, ObjHeader[0..(SEGM_SIZE-1)],
148     gIsTrans[0..(MAX_OBJS-1)], gHeadStart[0..(MAX_OBJS-1)],
149     gDataStart[0..(MAX_OBJS-1)], gDataEnd[0..(MAX_OBJS-1)], gCurObj, JCC,
150     PersiData[0..(SEGM_SIZE-1)], TransData[0..(SEGM_SIZE-1)], JPC, gJPCOff;
151   assigns  JPC \from &Code[0]; // possible base address
152   ensures  gNumObjs >= \at(gNumObjs,Pre);
153   ensures  vhm: valid_heap_model;
154   ensures  vcm: valid_code_model;
155   ensures  oh:  object_headers_intact{Pre,Post};
156   ensures  mmf: mem_model_footprint_intact{Pre,Post};
157 */
158 void other_opcode(void);
159
160 /*@
161   requires vhm: valid_heap_model;
162   requires vcm: valid_code_model;
163   assigns  gNumObjs, ObjHeader[0..(SEGM_SIZE-1)],
164     gIsTrans[0..(MAX_OBJS-1)], gHeadStart[0..(MAX_OBJS-1)],
165     gDataStart[0..(MAX_OBJS-1)], gDataEnd[0..(MAX_OBJS-1)], gCurObj, JCC,
166     PersiData[0..(SEGM_SIZE-1)], TransData[0..(SEGM_SIZE-1)], JPC, gJPCOff;
167   assigns  JPC \from &Code[0]; // possible base address
168   ensures  vhm: valid_heap_model;
169   ensures  vcm: valid_code_model;
170   ensures  oh:  object_headers_intact{Pre,Post}; */
171 void main_loop(){
172 /*@
173   loop invariant vhm: valid_heap_model;
174   loop invariant vcm: valid_code_model;
175   loop invariant oh:  object_headers_intact{LoopEntry,Here};
176   loop invariant mmf: mem_model_footprint_intact{LoopEntry,Here};
177   loop invariant no:  gNumObjs >= \at(gNumObjs,LoopEntry);
178   loop assigns gNumObjs, ObjHeader[0..(SEGM_SIZE-1)],
179     gIsTrans[0..(MAX_OBJS-1)], gHeadStart[0..(MAX_OBJS-1)],
180     gDataStart[0..(MAX_OBJS-1)], gDataEnd[0..(MAX_OBJS-1)], gCurObj, JCC,
181     PersiData[0..(SEGM_SIZE-1)], TransData[0..(SEGM_SIZE-1)], JPC, gJPCOff;
182 */
183   while(1){
184     if(*JPC == 1)                           // Assume code 1 is for BASTORE
185       /*@ ghost gCurObj=get_gu4(); */       // Assume arbitrary object index and
186       bastore(get_u4(),get_u4(),get_u1()); // header offset, body offset, value
187     else if(*JPC == 2)                      // Assume code 2 is for BALOAD
188       /*@ ghost gCurObj=get_gu4(); */       // Assume arbitrary object index and
189       baload(get_u4(),get_u4());            // header offset, body offset
190     else if(*JPC == 3)                      // Assume code 3 is for exit
191       return;
192     else                                    // Other opcodes
193       other_opcode();
194   }
195 }
```

Fig. 8. Complete illustrative example of the JCVM code, part 3/4.

```

/*@ // === Metaproperties: persistent object data written/read only by owner ===
meta \prop,\name(meta_persi_objects_integrity),\targets(\ALL),\context(\writing),
  ( \forall integer i; 0 <= i < gNumObjs && !gIsTrans[i] &&
  ObjHeader[gHeadStart[i] + 0] != JCC ==>
  \separated(\written,PersiData+(gDataStart[i]..gDataEnd[i])) );

meta \prop,\name(meta_persi_objects_confident),\targets(\ALL),\context(\reading),
  ( \forall integer i; 0 <= i < gNumObjs && !gIsTrans[i] &&
  ObjHeader[gHeadStart[i] + 0] != JCC ==>
  \separated(\read,PersiData+(gDataStart[i]..gDataEnd[i])) ); */

/*@ // === Metaproperties: transient object data written/read only by owner ===
meta \prop,\name(meta_trans_objects_integrity),\targets(\ALL),\context(\writing),
  ( \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] &&
  ObjHeader[gHeadStart[i] + 0] != JCC ==>
  \separated(\written,TransData+(gDataStart[i]..gDataEnd[i])) );

meta \prop,\name(meta_trans_objects_confident),\targets(\ALL),\context(\reading),
  ( \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] &&
  ObjHeader[gHeadStart[i] + 0] != JCC ==>
  \separated(\read,TransData+(gDataStart[i]..gDataEnd[i])) );
*/

// A bit-related lemma not proved by Alt-Ergo,Z3,CVC4 but easily proved by script
/*@ lemma dn: \forall u1 c; (c & 0x04)==0 && (c & 0x08)!=0 ==> (c & 0x0C)==0x08;*/

/* Run the proof with RTE and with MetAcsl:

frama-c-gui code_with_casts.c -machdep x86_32  -meta -meta-checks -meta-no-simpl
     -meta-no-check-ext -meta-number-assertions -then-last -wp
     -wp-check-memory-model -wp-rte -wp-prover=script,alt-ergo -wp-smoke-tests

Run the proof with RTE without MetAcsl:

frama-c-gui code_with_casts.c -machdep x86_32  -wp -wp-check-memory-model -wp-rte
     -wp-prover=script,alt-ergo -wp-smoke-tests

The previous commands run the proof using the pre-recorded scripts and the
     Alt-Ergo solver.
To observe the proof results without scripts, replace -wp-prover=script,alt-ergo
     by -wp-prover=alt-ergo
*/
```

Fig. 9. Complete illustrative example of the JCVM code, part 4/4.

A.2 Illustrative example of annotations generated by MetAcsl

Figures 10 and 11, show the function `bastore` with annotations automatically generated by MetAcsl for metaproperties `meta_persi_objects_integrity` and `meta_trans_objects_confident`. The generated assertions are inserted in the form of checks, that can be seen as "check-and-forget" assertions, not preserved in the proof contexts. These checks are automatically proved. Similar checks are inserted for other metaproperties and into other functions.

```
void bastore(u4 ObjRef, u4 DestOff, u1 Val)
{
  u1 tmp;
  /*@ check meta_persi_objects_integrity: _1: meta:
        \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
          ObjHeader[gHeadStart[i] + 0] != JCC ==>
          \separated(&tmp, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
  /*@ check meta_persi_objects_confident: _1: meta:
        \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
          ObjHeader[gHeadStart[i] + 0] != JCC ==>
          \separated(&ObjRef, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
  /*@ check meta_persi_objects_confident: _2: meta:
        \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
          ObjHeader[gHeadStart[i] + 0] != JCC ==>
          \separated(&DestOff, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
  tmp = firewall(ObjRef,DestOff);
  /*@ check meta_persi_objects_confident: _3: meta:
        \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
          ObjHeader[gHeadStart[i] + 0] != JCC ==>
          \separated(&tmp, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
  if (! tmp) goto return_label;
  /*@ check meta_persi_objects_confident: _4: meta:
        \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
          ObjHeader[gHeadStart[i] + 0] != JCC ==>
          \separated(&ObjRef, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
  /*@ check meta_persi_objects_confident: _5: meta:
        \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
          ObjHeader[gHeadStart[i] + 0] != JCC ==> \separated(&ObjHeader[ObjRef]+1,
          &PersiData[gDataStart[i] .. gDataEnd[i]]); */
  if ((int)*(& ObjHeader[ObjRef] + 1) & 0x08)
    /*@ check meta_persi_objects_integrity: _2: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
              \separated(&TransData[(u4)((u4)((unsigned short)((int)((int)((int)*
                (&ObjHeader[ObjRef] + 4) * 256) + (int)*(&ObjHeader[ObjRef]
                + 5)))) + DestOff)], &PersiData[gDataStart[i] .. gDataEnd[i]] );*/
    /*@ check meta_persi_objects_confident: _6: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
            \separated(&ObjRef, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _7: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
            \separated(&DestOff, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _8: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
            \separated(&Val, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _9: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0]!=JCC ==> \separated(&ObjHeader[ObjRef]+4,
            &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _10: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0]!=JCC ==> \separated(&ObjHeader[ObjRef]+5,
            &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    TransData[(u4)((unsigned short)((int)*(& ObjHeader[ObjRef] + 4) * 256 +
      (int)*(& ObjHeader[ObjRef] + 5))) + DestOff] = Val;
```

Fig. 10. Function `bastore` with annotations automatically generated by MetAcsl for metaproperties `meta_persi_objects_integrity` and `meta_trans_objects_confident`, part 1/2.

```
  else
    /*@ check meta_persi_objects_integrity: _3: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
            \separated(&PersiData[(u4)((u4)((unsigned short)((int)((int)((int)*
              (&ObjHeader[ObjRef] + 4) * 256) + (int)*(&ObjHeader[ObjRef] + 5))))
              + DestOff)], &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _11: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
            \separated(&ObjRef, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _12: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
            \separated(&DestOff, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _13: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0] != JCC ==>
            \separated(&Val, &PersiData[gDataStart[i] .. gDataEnd[i]]); */
    /*@ check meta_persi_objects_confident: _14: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0]!=JCC ==> \separated(&ObjHeader[ObjRef]+4,
            &PersiData[gDataStart[i] .. gDataEnd[i]] ); */
    /*@ check meta_persi_objects_confident: _15: meta:
          \forall integer i; 0 <= i < gNumObjs && gIsTrans[i] == 0 &&
            ObjHeader[gHeadStart[i] + 0]!=JCC ==> \separated(&ObjHeader[ObjRef]+5,
            &PersiData[gDataStart[i] .. gDataEnd[i]] ); */
    PersiData[(u4)((unsigned short)((int)*(& ObjHeader[ObjRef] + 4) * 256 +
      (int)*(& ObjHeader[ObjRef] + 5))) + DestOff] = Val;
  updateJPC();
  return_label: return;
}
```

Fig. 11. Function `bastore` with annotations automatically generated by MetAcsl for metaproperties `meta_persi_objects_integrity` and `meta_trans_objects_confident`, part 2/2.