Formal Verification of PKCS#1 Signature Parser using Frama-C

Martin Hána[1][0000−0001−8977−5950], Nikolai Kosmatov[2][0000−0003−1557−2813], Virgile Prevosto[3][0000−0002−7203−0968], and Julien Signoles[3][0000−0001−9266−0820]

[1]Thales Cybersecurity and Digital Identity, Prague, Czechia
[2]Thales Research and Technology, cortAIx Labs, Palaiseau, France
[3]Université Paris-Saclay, CEA, List, Palaiseau, France
[1,2]firstname.lastname@thalesgroup.com
[3]firstname.lastname@cea.fr

Abstract. Message parsing represents a complex security-critical problem. It has been demonstrated by numerous real-world exploits on parsers, e.g. on PKCS#1 (Public-Key Cryptography Standard) v1.5 signature, X.509 certificate chain, or infamously on a TLS extension during the Heartbleed attack. In this case study, we perform formal verification of a PKCS#1 v1.5 signature parser using Frama-C, where the verification of the parser is realized for the first time directly over the actual implementation in C. This brings highest guarantees of security and functional properties, while leaving developers the flexibility to adapt the code to the project's specific requirements. We present the proven properties, our verification approach and results. In particular, this work rules out applications of any variants of Bleichenbacher's signature forgery and ensures that we are able to detect potential parser incompatibilities. This work opens the door to future extensions to other protocols, for example, for parsing DER ASN.1 encoding of X.509 certificates and CRLs (Certificate Revocation Lists).

1 Introduction

As increasingly many aspects of public and private life are digitalized and moved into cyberspace, digital security becomes a major concern. The persistance of some families of bugs shows that code review and testing are not sufficient to reliably clean up the code even from publicly known vulnerabilities. One of the most risky attack vectors is related to the *message parsing problem*. Indeed, parsers are very often involved in processing of input data on external interfaces, where attackers have some level of control. Hence, parsers must apply a *rigorous defensive approach* and provide a high assurance that any malicious input is detected. It motivates the application of formal verification, capable to provide strong security guarantees.

An example of a long-lasting parsing problem—PKCS#1 v1.5 signature forging—was originally described by Bleichenbacher [28] in 2006, while exploitable variants have been reported in popular open-source libraries in 2021, 15 years later [57]! During PKCS#1 v1.5 signature verification, due to a complex structure of the signed message, a dedicated parser must be applied to extract necessary data (such as the hash of the message and the hash function identifier)

from the signature. An unsecured implementation of the parser opens the door to signature forgery. This case study paper addresses this problem.

Goals and approach. Our main goal is to demonstrate that it is achievable to perform formal verification of a PKCS#1 v1.5 signature parser [47] directly on the parser's C code. We formally specify *functional and security properties* of the parser and prove them in the Frama-C verification platform [5, 41]. These properties exclude any variant of Bleichenbacher attack or unauthorized memory access. Next to that, they ensure *parser compatibility* (with small limitations, discussed later in Sect. 5), i.e., the guarantee that correctly formatted messages are not refused by the parser. To illustrate that, we provide several buggy variants of the parser, for which—as expected—the proof with our specification fails.

Applying formal verification directly on the C implementation bridges the gap between a—typically, more abstract—formal model and the concrete code of the parser executed in practice, thus avoiding any non-trusted or laborious steps (such as code generation or refinement). In addition, it brings further practical advantages. For example, the code can be iteratively optimized or customized, as long as it is proved that each new modification retains all necessary properties and thus does not introduce vulnerabilities. On the other hand, it is often more difficult to verify real-life code than a more abstract high-level model. Further discussion about pros and cons of deductive verification on the source code level can be found in [23].

We select PKCS#1 v1.5 signature as a good representative of the TLV (Tag-Length-Value triple) tree structure, frequently used in parsers. In a TLV, the tag field defines the type of the data, and its length field defines the size of its value field. The PKCS#1 v1.5 signature has a non-trivial complexity yet is simple enough to tackle all methodological questions of parser verification. For the same reason, we develop a representative PKCS#1 v1.5 parser in C, for which we perform the proof, while keeping in mind the need of extension to other TLV-based protocols. This parser is capable of parsing the message structure specified in [47] almost completely (it is not compliant only to the fact that the NULL field can be *optional*). The annotated code of the verified parser and its buggy variants are available in a companion artifact [33].

The proposed specification approach of the message structure carefully combines *inductive predicates*—defined in a generic way—and a separate *ghost model* used in the inductive predicates and encoding a concrete TLV-based tree structure to be parsed. This separation appears to be practical and user-friendly. The proposed inductive predicates allow for a generic specification of different TLVs inside the same message and are expected to remain suitable for other message structures. It is an important benefit: such predicates are difficult to write correctly for a non-expert. The proposed ghost encoding of the TLV structure should be updated to parameterize the specification for other TLV structures. Closer to regular C data structures, it can be more easily written by engineers who are not experts in formal verification. It is another advantage of our approach.

Contributions. This verification case study presents the following contributions:

– a formal specification and proof—directly over the C code—of security and functional properties for a representative PKCS#1 v1.5 signature parser;
– an illustration by examples that the proposed specification rules out various variants of Bleichenbacher signature forgery [28] and ensures compatibility;
– a verification methodology based on the definition of inductive predicates, capable to conveniently express complex relations between particular TLVs in a generic way;
– an innovative usage of *ghost code* (see Sect. 2) to specify the target TLV structure and the application of cryptographic operations on concrete input data;
– a report of verification effort, results, faced difficulties and used workarounds.

Outline. Section 2 introduces Frama-C and the ACSL specification language. Section 3 presents some parser attacks and expected security and compatibility properties to guard against them. Section 4 details PKCS#1 v1.5 signature and known attacks. Section 5 describes our verification approach and proven properties. Section 6 provides a proof report. Finally, Sect. 7 presents related work and a conclusion.

2 Frama-C Verification Framework

Frama-C [5, 41] is a state-of-the-art modular verification framework for C code developed and maintained by CEA List. Its modular structure allows application and collaboration of various analyzers—implemented as plugins—and eases the introduction of new ones. This combination of plugins offers a large variety of verification and analysis approaches. In our work we use the WP and RTE plugins, which perform, resp., *deductive verification* (based on *weakest precondition* calculus) and generation of annotations whose validity implies the absence of *runtime errors.*

Deductive verification with WP is conducted in a modular way: each C function is proved to respect its function contract (and additional annotations in the function body), specified in ACSL (ANSI/ISO C Specification Language [4]) using typed first-order logic formulas. A *function contract* includes preconditions (**`requires`** clauses) and postconditions (**`ensures`** clauses). In addition, variables and memory locations the function is allowed to modify are listed in **`assigns`** clauses. Loops require additional annotations (**`loop invariant`** and **`loop assigns`** clauses). The reader can find more information on ACSL in [8, 12]. To show that the given C program respects the behavior specified by its annotations, the WP plug-in [9] generates *proof goals* (also called *proof obligations* or *verification conditions*). Such goals are mostly proven automatically, either by internal formula simplifier Qed [19] or external SMT solvers, but in some cases manual intervention is required to help the simplifier and the solvers by creating a *proof script*, a sequence of applications of predefined proof tactics. Examples of tactics include splitting a composed formula into simpler ones, unfolding a definition, instantiating a universally quantified formula with a concrete value (e.g. an index), rewriting bit-level operations, reasoning by case, etc.

Sometimes, it is also possible to circumvent the (time-consuming) development of such a proof script by adding assertions (using an **`assert`** clause), which must hold at a precise program point inside the function and can act as intermediate lemmas for the proof of complex goals. One example is to state an assertion that is useful to prove a predicate by explicitly providing a hypothesis. In some cases, **`assert`** clauses can also be generated by other plugins. In particular, the RTE plug-in [32] automatically emits **`assert`** clauses that are necessary to prove the absence of runtime errors (also known as *undefined behaviors*, as defined in the C standard [35], such as invalid memory accesses or some kinds of arithmetic overflows). Indeed, they must be avoided to ensure the soundness of the verification and exclude potential security vulnerabilities they can enable (notably via invalid memory accesses such as *buffer overflows*). WP generates proof goals for all assertions.

As illustrated in [10], there is a risk to introduce logical inconsistencies into specifications that are assumed, e.g. in entry-point function preconditions, environment hypotheses or postconditions of stub functions. WP can generate additional assertions to *try to detect* potential inconsistencies leading to logical contradictions. While very useful, this feature cannot guarantee the absence of inconsistencies, thus such specifications still must be carefully reviewed.

Verification can often benefit from *ghost code* [11, 22, 27], that is, additional C code added in annotations and used for verification only. It can create ghost C structures, possibly referring to structures of the original C code. Frama-C ensures non-interference of the ghost world into the non-ghost world: the semantics of the C code is not modified by ghost code (in particular, ghost instructions can read the content of C variables, but not modify it).

Our decision to use Frama-C is due to its capacity to successfully verify industrial C code and the fact that it is currently the only tool for C code verification recognized by ANSSI, the French Common Criteria certification body, as an acceptable formal verification technique for the highest levels of certification [23].

3 Parser Security and Related Requirements

Parsers are generally complex and dangerous software. Indeed, parsed messages can contain a lot of interdependencies between particular message elements. The attacker's control—full or even partial—over the parsed message can allow them to lead the processing system to unintended states and achieve practical exploits. This control can depend on the used cryptographic protections, the distance between the vulnerable parser and external interfaces, and other factors.

The most frequent parsing bugs are related to memory safety. All parsing steps must therefore carefully check that they access only valid memory of all involved buffers, in particular if this access is dependent on input data.

Numerous real-world exploits have been reported in last decades due to memory safety issues. For example, a simple overflow can lead to a segmentation error and a system crash, which can be exploited by attackers for denial-of-service attacks [16]. A buffer overflow during a reading operation, which relies on an incorrect buffer length from a malformed input message, can copy sensitive data

from the victim's memory and return it to the attacker, like in the HeartBleed vulnerability [2]. A buffer overflow during a writing operation can allow attackers to rewrite system metadata and get control over the subsequent execution [1].

Even when a message is cryptographically protected, a wrong interplay between the parser and the cryptographic module can lead to a loss of protection, for example, with an acceptance of unauthenticated data. Mechanisms of such a wrong cooperation differ. For example, the parser can return different data than what was previously authenticated by the module [29]. For the case of PKCS#1 v1.5 signature, RSA cryptographic strength is dependent on added padding that enforces modular exponentiation, a key operation used in RSA. The security of the whole signature scheme is thus dependent on the security of its message parser. Many bugs also come from arithmetic overflows [46], which can lead the program to unintended states and trigger some of the previous issues.

Let us state general requirements that will be refined and formalized below for the PKCS#1 v1.5 case ("sec" and "comp" stand for security and compatibility).

$\mathcal{R}_{\text{mem}}$**:** Parser processing must be memory-safe. All memory accesses must be done to valid memory.

$\mathcal{R}_{\text{arith}}$**:** Parser processing must be free of arithmetic overflows, except well-justified cases.

$\mathcal{R}_{\text{sec}}$**:** If the parser accepts a message (and extracts its data), it must enforce all checks necessary to ensure it has correct format and content.

$\mathcal{R}_{\text{comp}}$**:** If the parser refuses an input message, it must be based on a check, clearly showing that the message's format or content is incorrect.

4 PKCS#1 Signature Parser and Its Security

PKCS#1 (Public-Key Cryptography Standard) [47] specifies the usage of RSA algorithm [54], in particular for creation and verification of signatures. PKCS#1 signatures are used to protect X.509 certificates [18]. To enforce security, *padding schemes* (defining how additional padding should be used) are prescribed to be applied with an RSA operation. Although PKCS#1 v1.5 is an older signature padding scheme[1], it is still widely used for backward-compatibility. For example, the newest version of TLS [53] still mandates its support for certificate signature verification. Next to that, it is also still a valid option within SSH [58] and IPSec protocols [40, 48]. For those reasons, it still remains a target for attackers today.

4.1 TLV-based Message Structure

TLV-based structure. We denote the byte length of a data structure v by $\text{len}(v)$. The structure of messages manipulated by parsers is often based on *Tag-Length-Value triples*, or *TLVs*. A TLV $\theta = (t, l, v)$ contains three consecutive fields: a tag t used to identify the type of the TLV, a length field l containing the byte length of its value field, and a value field v containing the value. In other words,

[1] for new applications, it is superseded by PKCS#1-PSS.

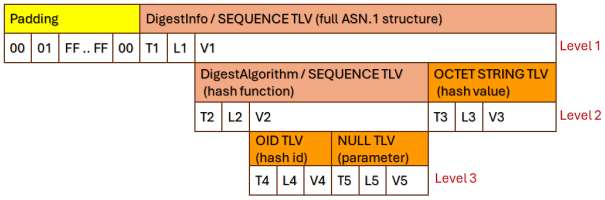

Fig. 1: PKCS#1 v1.5 signature: an ASN.1 structure with 5 nested TLVs preceded by a padding.

we have $l = \text{len}(v)$. The byte length of the whole TLV θ is the sum of the lengths of its fields, that is, $\text{len}(\theta) = \text{len}(t) + \text{len}(l) + \text{len}(v) = \text{len}(t) + \text{len}(l) + l$.

TLVs can be nested. Following [36], we say that a TLV is *constructed* if its value field contains a sequence of one or several TLVs. Otherwise, a TLV is called *primitive*. A TLV structure creates a tree, where constructed and primitive TLVs correspond, resp., to parent nodes and to leaves. Using this tree-based terminology we can speak of a TLV *level* (i.e., a depth in the tree) and define a parent-child relation between TLVs.

Example of TLVs. Consider the signature structure illustrated in Fig. 1, which is used by the target parser as detailed below. For the moment, we ignore the padding bytes on the left. At the upper level—Level 1—TLV $\theta_1 = (T_1, L_1, V_1)$ (of type DigestInfo/SEQUENCE in the standard) is constructed. It contains in its value field V_1 two other TLVs at Level 2: a TLV $\theta_2 = (T_2, L_2, V_2)$ (of type DigestAlgorithm/SEQUENCE in the standard) and a TLV $\theta_3 = (T_3, L_3, V_3)$ with a string of bytes (of type OCTET STRING in the standard). It means that V_1 consecutively stores the fields of θ_2 and θ_3. The latter is primitive. θ_2 is constructed again and contains two TLVs at Level 3, $\theta_4 = (T_4, L_4, V_4)$ (of type Object Identifier, or OID, in the standard) and $\theta_5 = (T_5, L_5, V_5)$ (of type NULL in the standard) containing a value V_5 of length $L_5 = 0$, i.e., an empty value with no bytes.

Correct message format and content. Following [15], we informally define two properties. A message has a *correct format* if it has the expected TLV structure (with tags specified in [47]), and its constructed TLVs verify property $\mathcal{P}_{\text{tree}}$:

$\mathcal{P}_{\text{tree}}$**:** For a constructed TLV $\theta = (t, l, v)$ whose value contains a sequence of n TLVs θ_i $(1 \leq i \leq n)$, the length of the value field of θ is the sum of lengths of its children θ_i, that is, $l = \sum_{i=1}^{n} \text{len}(\theta_i)$.

A message has a *correct content* if all its primitive TLVs have expected content, i.e. expected values of its length and value fields.

4.2 Algorithm Description

Overview. PKCS#1 v2.2 specification defines PKCS#1 v1.5 signature scheme as a *signature with appendix*. As depicted in Fig. 2, during the signature creation

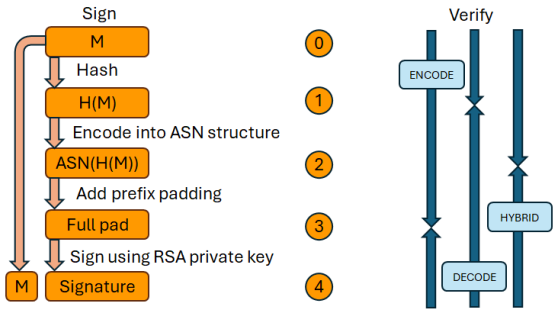

Fig. 2: PKCS#1 v1.5: creating a signature for message M (on the left) and three approaches for verification of the signature of message M (on the right).

procedure, the input message M is hashed, encoded, padded, and signed. The signature is the result of the RSA private key operation on a fully padded encoded message. This signature is then attached to the input message. Both are later required for signature verification, which can be done in several ways.

Signature creation. First, the hash operation (going from Step 0 to 1) provides a hash $H(M)$. It is then encoded into a so-called ASN.1 structure (Step 1 to 2). This is the nested TLVs structure shown in Fig. 1, which contains two main pieces of information: the OID identifier of the used hash function (θ_4) and the resulting hash value $H(M)$ of the input message (θ_3). An additional tag (θ_5) can host a hash parameter. For the SHA family of hashes, it should be NULL.

Next, a padding is added as a prefix to the ASN.1 structure (Step 2 to 3). It consists of two fixed bytes 0x00 and 0x01, followed by a variable-length sequence of padding bytes 0xFF, and finished by a delimiter byte 0x00, which separates it from the ASN.1 structure. The implementation of the resulting padded message is shown in Fig. 1. The number of padding bytes is chosen to achieve the message size required by the following RSA operation, that is, the size of RSA modulus.[2]

Correct signatures must satisfy two properties. First, the OID TLV must correspond to the ID of a hash function in the standard, and known to the parser. Second, the size of the hash value must correspond to this hash function.

Signature verification. Because of the deterministic nature of PKCS#1 v1.5 signature padding, it is possible to apply different approaches to signature verification, as depicted in Fig. 2. First, the Encoding approach—the main procedure presented in [47]—does not actually involve any parsing. Given an input message M and a hash function, M is re-encoded (Steps 0 to 3) and then compared byte-by-byte with the result of the RSA operation (using a public key to reverse from signature to padded message, Step 4 to 3). Omitting the parsing steps avoids a difficult situation, where all variants of Bleichenbacher's attack were reported.

However, the specification [47] also allows an alternative, the Decoding approach. There, the padded message content is parsed to extract the message hash

[2] this size defines the strength of an RSA operation, e.g. 1024 bits or 2048 bits.

(Steps 4 to 1). If the extracted hash is equal to the re-computed one (Step 0 to 1), the signature is accepted. To be secure, the parser must check for correctness of the message format and content. Compared to the previous one, this approach avoids allocating space for the re-encoded message, which has the RSA modulus size.

As reported in [57], many libraries use a Hybrid approach. They parse only the padding (cf. Fig. 1) and then extract the entire ASN.1 structure (Steps 4 to 2). The latter is not parsed, but instead compared, as in the Encoding approach, against the re-encoded structure (Steps 0 to 2). This approach seems a reasonable trade-off. Parsing of ASN.1, which is structurally complex, is avoided, resulting in a higher security assurance. At the same time, the ASN.1 structure is relatively small compared to modulus size, thus reducing memory space overhead.

The different approaches of the signature verification also impact the proof. If the Decoding approach is used, $\mathcal{P}_{\text{sec}}$ and $\mathcal{P}_{\text{comp}}$ are deduced from applied parsing steps. For the Encoding approach, the same properties would have to be proven based on a successfully matched re-encoded padded message (Step 3).[3] In this work, we focus on the Decoding approach, which contains complex parsing steps and is therefore the more bug prone and the most relevant for formal verification.

DER encoding and unique binary representation. One possible way to encode TLVs at binary level is defined by *DER (Distinguished Encoding Rules)* [36]. We also use it in our parser. The latest standard [47] mandates the use of DER for the creation of new signatures. DER ensures a *unique binary representation* of the data in the TLVs. When DER is used in PKCS#1 v1.5 to encode ASN.1, each of the tag and length fields of its TLVs is stored in *exactly one byte*.

Backward-compatibility. As it often happens in practice, evolution of software impacts compatibility. To ensure backward-compatibility, the standard [47] contains two requirements, which break the uniqueness of binary representation of the message: an *optional* presence of the NULL TLV[4] and support of another encoding, *BER (Basic Encoding Rules)* [36]. For simplicity, we do not support these requirements in this work (see also Sect. 4.4).

4.3 Signature Security

As already mentioned in Sect. 3, RSA security depends on the padding scheme. If the parser does not strictly check correct format and content of the padded message (State 3 of Fig. 2), it may lead to *universal signature forgery*, meaning that the attacker is able to create another actor's signature for *any* given message so that the resulting signature will be accepted by signature verification.

Since Bleichenbacher's original attack [28] reported in 2006, many variants have been found for the Decoding and Hybrid approaches. The attack is possible

[3] This approach actually exploits the uniqueness of the padded message for particular hash function and input message M, which is discussed in the next sections.

[4] This requirement is ignored by many open-source libraries [57] including OpenSSL v3.5, see `https://openssl-library.org/source`.

under the following two conditions. First, the signature must be verified using a public key of a specific small size—typically with public key exponent equal to 3. While rarely seen in practice currently [57], it can still be found in sensitive parts of the cryptographic ecosystem. For example, in 2021, Debian trust-anchor certificate bundle (ca-certificate) contained two certificates with such keys [57]. Four years later, they are still present on Ubuntu LTS 22.04.4.

Second, the verification procedure must ignore a sufficient number of bytes inside a padded message. Using the ignored bytes, an attacker can manage to compute a fake signature such that a public key operation on it—that is, taking a power of 3 *without modulo computation* in this case due to a small value—matches the data inside the checked bytes, such as the input message's hash. Note that it is impossible to avoid modulo computation if no byte inside a padded message is ignored and all are checked to some concrete values. Such a computation is then considered infeasible without knowledge of the private key.

Bleichenbacher's attack variants differ in locations that the buggy parser ignores. The original report [28] found parsers that ignored trailing bytes at the end of the padded message, i.e., after a correct ASN.1 structure. In [42, 49], the length field in the NULL TLV was not checked and its value skipped. Another variant [34] ignored bytes in an extended form of the length field due to an arithmetic overflow. In yet another variant [15], padding bytes were not checked.

We can now formulate a security property, which refines $\mathcal{R}_{\text{sec}}$ for the PKCS#1 v1.5 parser and gives a sufficient condition to rule out such attacks.

$\mathcal{P}_{\text{uniq}}$**:** The parser accepts a padded message (depicted at State 3 in Fig. 2) only if it checks each of its bytes for a unique correct value determined by input message M and hash function according to the PKCS#1 v1.5 specification.

4.4 Target Parser Implementation

For the sake of simplicity, we verify a representative PKCS#1 v1.5 signature parser that we implemented in C. It follows the Decoding approach, which is more bug-prone and thus more relevant for verification. Three main functions perform the parsing. Function `pars_PKCS1_lev1` is the parser's entry point. It removes the padding and realizes checks for the TLV at Level 1 (cf. Fig. 1). It expects two buffers (see line 649 of Fig. 7): the padded message (`pad_msg` of size `pad_msg_sz`), which is the output of an RSA operation on signature (State 3 in Fig 2), and the input message M (`in_msg` of size `in_msg_sz`). Its callee, function `pars_PKCS1_lev2`, handles the nested TLVs (Levels 2 and 3) and makes all semantic checks for the primitive TLVs' content. Repetitive steps for parsing a TLV are encapsulated in function `read_one_tlv`. The value returned by the parser either indicates that all checks pass, or provides an error code.

Our verification perimeter does not include the RSA operation, but includes a call to a hash computation (modeled by one of the three stub functions `stub_shaX`). This allows us to extend our properties to check that a proper hash function was indeed applied. As it is common for open-source libraries, the chosen hash function is indicated by an input parameter (`expect_hash_ind`,

```
 6 typedef unsigned char u1;
 7 typedef unsigned int uint;

70 /*@ ghost
71 const uint g_tlv_1[4] = {1, TAG_SEQ, CONSTR, 1};
72 const uint g_tlv_2[7] = {2, TAG_SEQ, CONSTR, 2, TAG_OCT,  PRIM, 0};
73 const uint g_tlv_3[7] = {2, TAG_OID, PRIM,   0, TAG_NULL, PRIM, 0};
74
75 \ghost const uint* const g_tlv_spec[3] =
76   { &g_tlv_1[0], &g_tlv_2[0], &g_tlv_3[0] };
77
78 u1* g_tlv_1_p[1]; u1* g_tlv_2_p[2]; u1* g_tlv_3_p[2];
79
80 u1* \ghost * const g_tlv_p[3] =
81   { &g_tlv_1_p[0], &g_tlv_2_p[0], &g_tlv_3_p[0] };  */
```

Fig. 3: Basic types and a ghost model of the TLV structure of ASN.1. Each `g_tlv_`N contains the specification of one TLV sequence.

see lines 649–650 of Fig. 7), expected to be a valid index in the list of supported hashes (see line 597). It allows the parser to check the hash function used inside the signature (as stored in the OID TLV) to any declaration external to PKCS#1 (e.g., the hash function declaration in the field of a X.509 certificate).

As it is essential for secure parsing programming, we chose a clear parsing pattern for our code, i.e., it parses all TLVs of the same sequence and checks $\mathcal{P}_{\text{tree}}$ property with respect to its parent before starting to parse its children. Similarly, before starting to read a new TLV—in functions `read_one_tlv` and `pars_PKCS1_lev1`—we check that the read indices are inside the entire parsed message (for Levels 2 and 3, it is done indirectly by checking the inclusion inside the value field of the parent TLV).

Our parser supports three hash functions (SHA256, SHA512, and, for backward compatibility, SHA1). Adding new functions is straightforward. The complete code is provided in the companion artifact. For lack of space, only essential (slightly simplified) parts are presented in Figs. 3–7, in which we preserve the line numbers of the complete file for convenience of the readers.

5 Formal ACSL Properties and Verification Approach

Correct message format (incl. $\mathcal{P}_{\text{tree}}$). To describe TLV-based structures, our methodology strongly relies on *inductive predicate* definitions in ACSL. Indeed, due to the structural nature of $\mathcal{P}_{\text{tree}}$, it is convenient to inductively define the correctness of a parsing step, as well as to maintain information about the already (correctly) parsed elements of the structure in the form of an inductive predicate. An interesting feature of the approach is that those predicates are *generic*, and are parameterized by a concrete TLV structure, defined as a *ghost model*.

Figure 3 presents the ghost model of the ASN.1 structure (cf. Fig. 1). The ghost models of the sequences of TLVs at Levels 1–3 are encoded, resp., in arrays `g_tlv_`N with $N \in \{1, 2, 3\}$ (see lines 71–73 in Fig. 3). The first byte indicates the number of TLVs in the sequence. The next bytes store the descriptions for each TLV of the sequence. Each description is composed of three bytes. The first

```
86 inductive valid_tlv(u1* start, integer prev_pars, integer size_pars,
87   integer tlv_num_pars, integer tlv_id, boolean with_children)
88 {
89 case empty: \forall u1* start, integer tlv_id;
90   valid_tlv(start, 0, 0, 0, tlv_id, \true);
91
92 case new_tlv_child_not_incl:
93   \forall u1* start, integer prev_pars, integer size_pars,
94     integer tlv_num_pars, integer tlv_id;
95   \let tlv_cur = g_tlv_spec[tlv_id];
96   0 <= tlv_num_pars < tlv_cur[0] &&
97   valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true) &&
98   start[size_pars] == (u1)tlv_cur[1 + tlv_num_pars*3]
99   ==>
100  valid_tlv(start, size_pars, size_pars + 2 + start[size_pars + 1],
101    tlv_num_pars+1, tlv_id, \false);
...
113 case last_tlv_constr_incl_child:
114   \forall u1* start, integer prev_pars, integer size_pars,
115     integer tlv_num_pars, integer tlv_id, integer nest_prev_pars;
116   \let tlv_cur = g_tlv_spec[tlv_id];
117   \let tlv_link = tlv_cur[3 + (tlv_num_pars-1)*3];
118   1 <= tlv_num_pars &&
119   valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \false) &&
120   (u1)tlv_cur[2 + (tlv_num_pars-1)*3] == CONSTR &&
121   valid_tlv(start + prev_pars + 2, nest_prev_pars, start[prev_pars + 1],
122     g_tlv_spec[tlv_link][0], tlv_link, \true)
123   ==>
124   valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true); }
```

Fig. 4: Inductive predicate for correct message format (`start`: parsing beginning, `prev_pars`: previous number of consumed (parsed) bytes, `size_pars`: number of consumed bytes, `tlv_num_pars`: number of parsed TLVs, `tlv_id`: TLV index inside `g_tlv_spec`, `with_children`: child TLV sequence checked).

and the second bytes indicate, resp., the expected tag and TLV type (constructed or primitive). For a primitive TLV, the third byte contains a dummy value (here, 0, cf. lines 72, 73). For a constructed TLV, the third byte is used to find the ghost model of the child sequence of TLVs. It is indicated indirectly,[5] as an index in the ghost array `g_tlv_spec` (cf. lines 75–76) that stores pointers to the ghost models of TLV sequences. For instance, index 2 on line 72 refers to the sequence of Level 3 since the element of index 2 in `g_tlv_spec` is `&g_tlv_3[0]`. The `\ghost` keyword indicates that the pointed arrays are also ghost. As shown in Fig. 1, each TLV level contains one sequence of TLVs in our case[6]. Lines 78–81 of Fig. 3 will be explained below.

The `valid_tlv` inductive predicate defined in Fig. 4 states that a given TLV sequence has a correct format (so far). It takes 6 arguments. First, parameter `start` points to a location where the parsing of the TLV sequence begins. The number of already consumed bytes and already parsed TLVs are given, resp., in

[5] An alternative way of encoding was suggested by a reviewer. The investigation of such alternative approaches is left for future work.

[6] In general, there will be n sequences of TLVs at Level $k+1$ if at Level k we have n TLVs having a child sequence.

size_pars and tlv_num_pars. The ghost model of the target TLV sequence is indicated as its index within g_tlv_spec by tlv_id. The last two parameters are more technical, related to the current stage of the parsing. Parameter with_children indicates whether child TLV sequences were already resolved (i.e., checked) for the last parsed TLV, as described below. Lastly, prev_pars is the number of bytes consumed before parsing the last parsed TLV.

For example, consider the TLV structure of Fig. 1 and assume θ_1 starts at p. When the parsing of TLV θ_1 is correctly finished at Level 1, before checking the child sequence, we have (i) valid_tlv(p,0,2+L1,1,0,\false), and (ii) valid_tlv(p,0,2+L1,1,0,\true) after checking the child sequence. The latter step requires a correct format of the sequence at Level 2, leading after parsing θ_2 (and its child sequence) to (iii) valid_tlv(p+2,0,2+L2,1,1,\true) and finally to (iv) valid_tlv(p+2,2+L2,4+L2+L3,2,1,\true) after parsing θ_3. Recall that the ghost models of Levels $1, 2$ are, resp., at indices $0, 1$ in g_tlv_spec. To respect $\mathcal{P}_{\text{tree}}$, L_1 must be equal to $4 + L_2 + L_3$.

Each induction step is defined by one case. Case empty initializes the induction for a starting pointer and a ghost model, with no parsed bytes and no parsed TLVs (lines 89–90 in Fig. 4). Next, case new_tlv_child_not_incl (lines 92–101) parses the next TLV in the sequence. The tag and the number of TLVs are checked against the ghost model (lines 95–96, 98), the size of the new TLV is added to the number of parsed bytes and the number of parsed TLVs is incremented (lines 100–101). After this step, with_children is false, indicating that for the last parsed TLV, the child sequence has still to be checked.

To deduce a full validity (i.e., with with_children set to true) for the last parsed TLV, we introduce two other cases, resp., for a constructed and a primitive TLV. Case last_tlv_constr_incl_child is applied if the last parsed TLV θ is constructed and its child sequence has not yet been checked (lines 119–120). We identify the ghost model of this child sequence (line 117), and check that the child TLVs are correct with respect to this model, including their own children if any (lines 121–122). Property $\mathcal{P}_{\text{tree}}$ is then ensured for θ and its children. The case for a primitive TLV is trivial (switching with_children to true), and is omitted here.

In the previous example, step last_tlv_constr_incl_child is used to deduce (ii) from (i) and (iv). The reader can check by a step-by-step application that it works only if $L_1 = 4 + L_2 + L_3$, thus ensuring $\mathcal{P}_{\text{tree}}$.

Correct message content. According to [47] (and using the notation of Fig. 1), a valid message content must satisfy the following properties for primitive TLVs. (These properties are specific for the considered message structure.)

- $\mathcal{P}_1$**:** The parsed OID (fields L_4 and V_4) corresponds to the indicated hash function (parameter expect_hash_ind in the entry point function).
- $\mathcal{P}_2$**:** The length of NULL TLV (field L_5) is equal to 0.
- $\mathcal{P}_3$**:** The length of the parsed hash value (L_3) corresponds to the parsed OID.
- $\mathcal{P}_4$**:** The function used for hash re-computation corresponds to the parsed OID.
- $\mathcal{P}_5$**:** Hash re-computation is applied on input message M (parameter in_msg in the entry point function).

```
193 predicate prim_tlv_oid_param(integer exp_hash_ind) =
194   \let oid_len = *(g_tlv_p[2][0]+1);                                  // Reads L4
195   \let oid_val_p = g_tlv_p[2][0]+2;                                   // Points to V4
196   \let param_len = *(g_tlv_p[2][1]+1);                                // Reads L5
197   (oid_len == OID_list[exp_hash_ind][0]) &&                           // En-
198   same_content(OID_list[exp_hash_ind]+1, oid_val_p, oid_len) && //   sures P1
199   param_len == 0;                                                     // Ensures P2
```

Fig. 5: $\mathcal{P}_1$ and $\mathcal{P}_2$: content properties related to OID and NULL TLVs.

$\mathcal{P}_6$**:** The parsed hash value (V_3) is equal to the re-computed hash value.

Since our inductive predicate `valid_tlv` focuses on the correct format in a generic way and does not trace particular TLVs, we cannot use it to identify, e.g., θ_4 for validating $\mathcal{P}_1$ when looking at the message as a whole. Similarly, as we can see for example for $\mathcal{P}_3$, there are dependencies between different TLVs, sometimes from different TLV levels. We detail here only $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_5$.

To express these properties, we introduce a ghost array `g_tlv_p` (see lines 78–81 of Fig. 3), whose structure and size are similar to `g_tlv_spec`. During parsing, we store—via ghost code—pointers to the beginning of each primitive TLV at the corresponding locations of this array. Once parsing is completed successfully, it allows us to refer to any primitive TLV. For instance, `g_tlv_p[2][0]` stores a pointer to the first TLV (second index 0) in the TLV sequence at Level 3 (first index 2), that is, to θ_4. Added ghost code is simple—it stores a pointer to each primitive TLV into the corresponding array element. It is however crucial to capture all primitive TLVs. To avoid any mistake, we introduce a second inductive predicate `prim_gh_set`. Structurally, it mimics `valid_tlv` presented above, with two important differences: it does not check again the expected tag, and it requires `g_tlv_p` to be correctly assigned with pointers to primitive TLVs. Maintaining `prim_gh_set` together with `valid_tlv` ensures that the message was checked for correct format and all pointers to primitive TLVs were correctly set, being readily available for content verifications.

Figure 5 shows a predicate to check $\mathcal{P}_1$ and $\mathcal{P}_2$. On lines 194–196, the OID and NULL TLVs (θ_4 and θ_5) are accessed. To get the OID length L_4 we first find θ_4, the first TLV at level 3, using pointer `g_tlv_p[2][0]`. This gives us the address of T_4. We increment it by 1 to get the address of L_4, that we dereference to get its value. Using our list `OID_list` of IDs of supported hash functions, line 197 checks that L_4 is the expected length of the ID, while line 198 verifies that V_4 contains the expected ID. On line 199 we then check L_5 to be 0.

Ghost code is also used to store necessary information for the proof of $\mathcal{P}_4$ and $\mathcal{P}_5$. Let us detail $\mathcal{P}_5$ here. It requires to check that the hash function is applied indeed on the *original* message. To keep a copy of the message, we introduce a ghost array `g_backup_mes_to_hash` and a stub function `g_make_backup` (see Fig. 6). Its contract ensures that the function copies the content of the input into the ghost array. We call it at the very beginning of the entry point function (line 652 of Fig. 7). To verify $\mathcal{P}_5$ at a hash computation, typically realized by a crypto library call, it is sufficient to add a precondition checking that the content of its input array is identical to the one saved in `g_backup_mes_to_hash`. The

```
227 /*@ ghost
228 u1 g_backup_mes_to_hash[INT_MAX];
229 uint g_backup_len;
230
231 /@ requires \valid_read(msg + (0 .. msg_sz-1));
232   terminates \true; exits \false;
233   assigns g_backup_mes_to_hash[0 .. msg_sz-1];
234   ensures same_content(msg, &g_backup_mes_to_hash[0], msg_sz);
235   ensures g_backup_len == msg_sz; @/
236 void g_make_backup(u1* msg, int msg_sz); */
```

Fig. 6: Ghost array and function for creating a memory snapshot.

strong separation[7] of the non-ghost world from the ghost one enforced by Frama-C [4] excludes unintended modifications of `g_backup_mes_to_hash` in the C code, thus making this approach consistent.

Entry point function contract. Figure 7 shows the contract for the entry point function, `pars_PKCS1_lev1`. Several predicates are introduced to deal with the padding shown in Fig. 1. Predicate `prefix_prop` expresses the presence of the two fixed prefix bytes (lines 557–558). Then, `padding_prop` requires a variable length sequence containing (at least `MIN_FF_COUNT`) times 0xFF and finished by a delimiter byte (at offset `pad_bound`). Predicate `asn_prop` states that the Level 1 ASN.1 TLV is correctly formatted (using `valid_tlv`) and occupies the entire space between the delimiter and the end of the padded message. As we already explained, it is valid only if all its TLV descendants were correctly parsed too. Predicate `ghost_set_prop` ensures that referenced ghost pointers were set correctly. Regarding message content properties, $\mathcal{P}_1$–$\mathcal{P}_4$ and $\mathcal{P}_6$ are regrouped in predicate `prim_tlv` (not detailed here, used on line 625), while $\mathcal{P}_5$ is ensured at another location.

$\mathcal{R}_{\text{sec}}$ *and* $\mathcal{P}_{\text{uniq}}$. As we can see on lines 614–625 of Fig.7, if the parser accepts the message, the conjunction of all predicates above must hold. We recall there are 3 primitive TLVs within ASN.1 structure: θ_3, θ_4 and θ_5. One can easily see that properties $\mathcal{P}_1$–$\mathcal{P}_6$ specify all bytes of fields L_i, V_i for $3 \leq i \leq 5$[8] by making them uniquely determined by the input message and expected hash function. Indeed, for example on lines 197–198 of Fig. 5, we specify all bytes of L_4 and V_4 when we match parsed OID against parser list of supported OIDs. Inductive predicate `valid_tlv` then provides unique specification for all TLV tags. Next to that, due to $\mathcal{P}_{\text{tree}}$, lengths of constructed TLVs are uniquely determined by the contained primitive TLVs. Based on this analysis of our properties, we see that the proved parser indeed *accepts only one padded message* for each input message and expected hash function and so it fulfills $\mathcal{P}_{\text{uniq}}$.[9] As explained in Sect. 4.3, this guarantees its security against Bleichenbacher's attack family.

[7] Indeed, the C code cannot see ghost variables at all, while the ghost code can only read non-ghost variables but cannot modify them.

[8] V_5 is absent as L_5 is equal to 0.

[9] An explicit specification and verification of $\mathcal{P}_{\text{uniq}}$ in Frama-C may be performed using the RPP plug-in [13], which allows users to express *relational properties* between two

```
556 /*@ ghost uint pad_bound;*/
557 /*@ predicate prefix_prop(u1* pad_msg) =
558   pad_msg[OFF_B1] == EXP_B1 && pad_msg[OFF_B2] == EXP_B2;
559 predicate padding_prop(u1* pad_msg, integer pad_msg_sz, integer pad_bound)=
560   2+ MIN_FF_COUNT <= pad_bound < pad_msg_sz &&
561   (\forall integer padd_off; 2 <= padd_off < pad_bound ==>
562     pad_msg[padd_off] == EXP_FF) && pad_msg[pad_bound] == EXP_DELIM;
563 predicate asn_prop(u1* pad_msg, integer pad_msg_sz, integer pad_bound,
564   integer tlv_num) =
565   2+ MIN_FF_COUNT <= pad_bound < pad_msg_sz &&
566   \exists integer prev_pars; valid_tlv(pad_msg + pad_bound + 1, prev_pars,
567     pad_msg_sz - pad_bound - 1, tlv_num, 0, \true);
568 predicate ghost_set_prop(u1* pad_msg, integer pad_msg_sz, integer pad_bound,
569   integer tlv_num)=
570   2+ MIN_FF_COUNT <= pad_bound < pad_msg_sz &&
571   \exists integer prev_pars; prim_gh_set(pad_msg + pad_bound + 1, prev_pars,
572     pad_msg_sz - pad_bound -1, tlv_num, 0, \true); */
...
595 requires \valid(pad_msg + (0 .. pad_msg_sz-1));
596 requires \valid(in_msg + (0 .. in_msg_sz - 1));
597 requires 0 <= expect_hash_ind < SUPP_OID_COUNT;
...
614 // ======= SECURITY PART =======
615 behavior sec_format:
616   ensures prefix: \result == PASS_OK ==> prefix_prop(pad_msg);
617   ensures pad: \result == PASS_OK ==>
618     padding_prop(pad_msg, pad_msg_sz, pad_bound);
619   ensures asn: \result == PASS_OK ==>
620     asn_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
621
622 behavior sec_prim_tlv:
623   ensures ghost_set: \result == PASS_OK ==>
624     ghost_set_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
625   ensures prim_tlv:  \result == PASS_OK ==> prim_tlv(expect_hash_ind);
626
627 // ======= COMPATIBILITY (ACCEPTANCE) PART =======
628 behavior acc_format:
629   ensures prefix:
630     \result == ERR_B1||\result == ERR_B2 ==> !prefix_prop(pad_msg);
631   ensures pad:
632     \result == ERR_FF_COUNT||\result == ERR_DELIM ==>
633     \forall uint pad_bound; !padding_prop(pad_msg, pad_msg_sz, pad_bound) ||
634     !asn_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
635   ensures asn:
636     \result == ERR_LEN||\result == ERR_TAG||\result == ERR_TLV_LEN_CONSIST ==>
637     \forall uint pad_bound; !padding_prop(pad_msg, pad_msg_sz, pad_bound) ||
638     !asn_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
639
640 behavior acc_prim_tlv:
641   ensures ghost_set:
642     \result == ERR_HASH_SZ||\result == ERR_HASH_OID||\result == ERR_HASH_VAL||
643     \result == ERR_NULL_SZ ==>
644     ghost_set_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
645   ensures prim_tlv:
646     \result == ERR_HASH_SZ||\result == ERR_HASH_OID||\result == ERR_HASH_VAL||
647     \result == ERR_NULL_SZ ==> prim_tlv_neg(expect_hash_ind);
648 */
649 int pars_PKCS1_lev1(u1* pad_msg, int pad_msg_sz, u1* in_msg, int in_msg_sz,
650   uint expect_hash_ind)
651 {
652   //@ ghost g_make_backup(in_msg, in_msg_sz);
```

Fig. 7: Entry point parser contract, where `pad_msg` and `pad_msg_sz` denote the padded message and its size, `in_msg` and `in_msg_sz`, the input message M and its size, and `expect_hash_ind` is the index of the used hash function.

$\mathcal{R}_{\text{comp}}$. We formalize our properties for $\mathcal{R}_{\text{comp}}$ on lines 627–647 of Fig.7. The specification is split based on the error type: for example, if an error is returned for insufficient space during the parsing of a new TLV, an unexpected tag or an inconsistent TLV structure (with regard to $\mathcal{P}_{\text{tree}}$), it implies a falsification of `valid_tlv` (lines 635–638). In case an error indicates a wrong content, we prove a correct setting of ghost pointers and a falsification of content properties (lines 640–647). An additional reason to split `prim_gh_set` and `valid_tlv` into two separate predicates—despite their similar structure—is the following. Predicate `prim_gh_set` can be falsified also by a buggy ghost code, (i.e. a mistake made during proof development). On the other hand, $\mathcal{R}_{\text{comp}}$ requires that the parser returns an error code only based on a wrong message format or content. Inside our specification, we therefore cannot allow the parser to refuse the message based on a failure of `prim_gh_set`.

$\mathcal{R}_{\text{mem}}$ and $\mathcal{R}_{\text{arith}}$. $\mathcal{R}_{\text{mem}}$ and $\mathcal{R}_{\text{arith}}$ are checked by annotations automatically generated by RTE (see Sect. 3). We specify a precondition of the entry point function, claiming memory validity of both input buffers (lines 595–596 of Fig.7). Regarding $\mathcal{R}_{\text{arith}}$, as explained in [32], some arithmetic overflows (e.g. for char or unsigned integers) fall into the category of implementation-defined or even well-defined behavior and thus are not checked by RTE by default. In our verification, we also activate the corresponding checks to avoid any overflows.

Current limitations and perspectives of extension. While this case study already targets a representative parser, our approach may require extensions for the verification of other PKCS#1 v1.5 parser implementations or different TLV-based protocol. We currently consider the Decoding approach, with 1-byte fields both for a tag and a length field, and a mandatory presence of a NULL TLV. Extensions to support other choices are left as future work.

6 Proof Results

Summary of results. We use Frama-C v. 29.0 (Copper) with external SMT prover Alt-Ergo 2.5.4. on a VM running Ubuntu 24.04 under VirtualBox (running on a host PC under Windows 10 with Intel(R) Core(TM) i7 CPU @ 2.70GHz) with 4 processors and 8 GB dedicated to the VM. The verified parser (155 lines of C code) includes seven C functions (including three stubs) annotated with 364 lines[10] of ACSL, giving a 2.35 spec-to-code ratio. To show the absence of runtime errors, we use the RTE plugin, which automatically generates 52 asserts.

Overall, the WP plugin generates and successfully proves 434 proof goals. Among them, reachability analysis proves 4 goals. The internal WP simplifier,

runs of the parser. It could be used to prove that if the parser accepts two padded messages for a given message M and a given hash function, then these two padded messages are necessarily identical. This extension is left for future work.

[10] We apply cloc utility. It considers ACSL annotations as other C comments. We thus removed classical C comments to count ACSL. Empty lines in C code are ignored.

Qed, discharges 251 goals. Alt-Ergo solves 176 goals. 3 goals require manually created WP proof scripts. The whole proof (with additional 60 consistency checks activated by `-wp-smoke-tests`) takes 2 min 8 s, with up to 14.2 s for one proof goal. We estimate the total effort to perform the verification case study (incl. creating a suitable methodology for a new kind of target code, identifying properties, annotating code and creating necessary proof scripts) as 6 person-months. Acquiring necessary expertise in cybersecurity and proof is not included in it.

Selected difficulties. The created proof scripts instantiate the most complex induction steps of `valid_tlv` (see line 113 in Fig. 4) and `prim_gh_set`, which check the nested TLV structure. Next to that, we use several **`assert`** clauses, which further help us to instantiate other steps of these inductive predicates. It is for example necessary for each TLV sequence to initialize induction.

In our work, we experienced a specific difficulty with proofs involving inductive predicates. Indeed, in some cases, an assignment to an unrelated memory location would cause previously established instances of the inductive predicate to be "forgotten" by the prover. We worked around this issue by adding **`assert`** clauses for initial induction steps after the writing. This is not a universal solution and another methodology might be needed in other cases.

7 Related Work and Conclusion

Related work. Various approaches using formal methods were already applied on parsers. Symbolic execution and adaptive combinatorial testing were used to generate test suites and to discover a significant number of bugs in PKCS#1 v1.5 signature verification of public cryptographic libraries [15, 57]. Formal verification was deployed [21, 57] using Coq and Agda, providing test oracles and a reference implementation of PKCS#1 v1.5 signature and X.509 certificate validation logic. A formally proven PKCS#1 v1.5 signature verification procedure was further extracted to OCaml source code [57]. However, the proof was not performed for efficient real-life parsers. Our approach is on the contrary suitable for proving optimized parsers in C. Ramananandro et al. developed an automatic parser generator, formally proven in F*, and extracted C code using a non-verified tool [52]. Although it can serve to generate a big amount of parser code automatically [55], some validation logic is not expressible and the corresponding code must be added manually [21]. Li et al. used ACL2 formal language and developed a formally verified PDF parser and a semantic validator [45]. However, the correspondence between a formally verified ACL2 code and an efficient C code was not proved.

More generally, this work is related to other verification case studies [31]. Various verification tools were applied to verify real-life code, including KeY [7, 20], VerCors [3, 50], Frama-C [22, 24, 26], SPARK [17, 25], VCC [43], Dafny [14, 44], VeriFast [37, 51] and many others. Each new case study contributes to enhance verification tools by identifying their limitations and to push further the frontiers of what is achievable for formal verification.

Conclusion and future work. We have proposed a *methodology for formal verification* of a PKCS#1 v1.5 parser in Frama-C, and successfully *verified a representative parser* written in C. The proven properties include *security*, protecting against any variant of *Bleichenbacher's attack* and ensuring *memory safety*, and *compatibility*, i.e. non-rejection of a correct signature. To the best of our knowledge, it is the first time that formal verification is applied directly on the C implementation of such a parser. We proposed an original specification approach that relies on *inductive predicates*—defined in a generic way—and a separate *ghost model* encoding a concrete TLV-based structure to be parsed. We believe that this separation brings benefits for future extensions and industrial applications. The inductive predicates—more difficult to write—can be reused, whereas the ghost model should be adapted for another structure, that can be done more easily by non-experts.

Future work includes extensions to support additional features (see the end of Sect. 5), and formal verification of real-life open-source C parsers. A comparison of our parser with other similar parsers (in particular, regarding their performances) is another interesting work perspective. We believe that our methodology also provides a basis for future extensions to other TLV-based formats, like X.509 certificates [18]. A study of applications of Large Language Models (LLMs) to generate (candidate) specifications for parsers is another future work direction, which becomes popular today [6, 30, 38, 39, 56]. Finally, an easy integration of the proposed methodologies into industrial workflows should remain an important point of attention.

Data availability statement. The companion artifact [33] contains the annotated code of the parser, examples of its buggy versions, proof scripts, and a virtual machine (with all necessary tools installed), ready to reproduce the proof.

Acknowledgments. Parts of this work have been funded by the French National Research Agency (ANR) through PEPR Cyber Secureval (Grant ANR-22-PECY-0005), EMASS (Grant ANR-22-CE39-0014), CoMeMov (Grant ANR-22-CE25-0018), VeDySec (Grant ANR-24-ASM2-0001), and the European Union's Horizon Europe projects SecOPERA (Grant 101070599) and VASSAL (Grant 101160022). We would also like to thank the Frama-C team for their support and the reviewers for their insightful comments.

References

1. CVE-2002-0639 (2002), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0639
2. Heartbleed bug (2014), https://www.heartbleed.com
3. Armborst, L., Bos, P., van den Haak, L.B., Huisman, M., Rubbens, R., Sakar, Ö., Tasche, P.: The VerCors verifier: A progress report. In: Proc. of the 36th International Conference on Computer Aided Verification (CAV 2024). LNCS, vol. 14682, pp. 3–18. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_1

4. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
5. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. Commun. ACM (2021). https://doi.org/10.1145/3470569
6. Beckert, B., Klamroth, J., Pfeifer, W., Röper, P., Teuber, S.: Towards combining the cognitive abilities of large language models with the rigor of deductive progam verification. In: Proc. of the 12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2024). LNCS, vol. 15222, pp. 242–257. Springer (2024). https://doi.org/10.1007/978-3-031-75387-9_15
7. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally verifying an efficient sorter. In: Proc. of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024). LNCS, vol. 14570, pp. 268–287. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_15
8. Blanchard, A.: Introduction to C program proof with Frama-C and its WP plugin (2020), https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf
9. Blanchard, A., Bobot, F., Baudin, P., Correnson, L.: Formally Verifying That a Program Does What It Should: The Wp plug-in. In: Guide to Software Verification with Frama-C. Core Components, Usages, and Applications, pp. 187–261. Springer (2024). https://doi.org/10.1007/978-3-031-55608-1_4
10. Blanchard, A., Correnson, L., Djoudi, A., Kosmatov, N.: No smoke without fire: Detecting specification inconsistencies with Frama-C/WP. In: Proc. of the 18th International Conference on Tests and Proofs (TAP 2024), co-located with the 26th International Symposium on Formal Methods (FM 2024). LNCS, vol. 15153, pp. 65–83. Springer (Sep 2024). https://doi.org/10.1007/978-3-031-72044-4_4
11. Blanchard, A., Loulergue, F., Kosmatov, N.: Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In: Proc. of the 10th NASA Formal Methods Symposium (NFM 2018). LNCS, vol. 10811, pp. 37–53. Springer (2018). https://doi.org/10.1007/978-3-319-77935-5
12. Blanchard, A., Marché, C., Prevosto, V.: Formally Expressing What a Program Should Do: The ACSL language. In: Guide to Software Verification with Frama-C. Core Components, Usages, and Applications, pp. 3–80. Springer (2024). https://doi.org/10.1007/978-3-031-55608-1_1
13. Blatter, L., Kosmatov, N., Prevosto, V., Robles, V.: Chapter 10: Specification and verification of high-level properties. In: Guide to Software Verification with Frama-C. Core Components, Usages, and Applications, pp. 457–486. Springer (2024). https://doi.org/10.1007/978-3-031-55608-1_10
14. Cassez, F., Fuller, J., Quiles, H.M.A.: Deductive verification of smart contracts with Dafny. In: Proc. of the 27th International Conference on Formal Methods for Industrial Critical Systems (FMICS 2022). LNCS, vol. 13487, pp. 50–66. Springer (2022). https://doi.org/10.1007/978-3-031-15008-1_5
15. Chau, S.Y., Yahyazadeh, M., Chowdhury, O., Kate, A., Li, N.: Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification. In: Proc. of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019). The Internet Society (2019). https://doi.org/10.14722/ndss.2019.23430

16. Cisco Talos: WolfSSL library X509 Certificate Text Parsing Code Execution Vulnerability (2017), https://talosintelligence.com/vulnerability_reports/TALOS-2017-0293
17. Cluzel, G., Georgiou, K., Moy, Y., Zeller, C.: Layered formal verification of a TCP stack. In: Proc. of the IEEE Secure Development Conference (SecDev 2021). pp. 86–93. IEEE (2021). https://doi.org/10.1109/SecDev51306.2021.00028
18. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile (2008). https://doi.org/10.17487/RFC5280, RFC 5280
19. Correnson, L.: Qed. Computing What Remains to Be Proved. In: Proc. of the 6th International Symposium on NASA Formal Methods (NFM 2014). LNCS, vol. 8430, pp. 215–229. Springer (2014). https://doi.org/10.1007/978-3-319-06200-6_17
20. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. Formal Aspects Comput. **35**(3), 18:1–18:26 (2023). https://doi.org/10.1145/3594729
21. Debnath, J., Jenkins, C., Sun, Y., Chau, S.Y., Chowdhury, O.: ARMOR: A formally verified implementation of X.509 certificate chain validation. In: Proc. of the IEEE Symposium on Security and Privacy (SP 2024). pp. 1462–1480. IEEE (2024). https://doi.org/10.1109/SP54263.2024.00220
22. Djoudi, A., Hána, M., Kosmatov, N.: Formal Verification of a JavaCard Virtual Machine with Frama-C. In: Proc. of the 24th International Symposium on Formal Methods (FM 2021). LNCS, vol. 13047, pp. 427–444. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_23, long version available at https://nikolai-kosmatov.eu/publications/djoudi_hk_fm_2021.pdf
23. Djoudi, A., Hána, M., Kosmatov, N., Kříženecký, M., Ohayon, F., Mouy, P., Fontaine, A., Féliot, D.: A Bottom-Up Formal Verification Approach for Common Criteria Certification: Application to JavaCard Virtual Machine. In: Proc. of the 11th European Congress on Embedded Real-Time Systems (ERTS 2022) (2022), https://hal.science/hal-03695829
24. 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
25. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Proc. of the 9th International Symposium on NASA Formal Methods (NFM 2017). LNCS, vol. 10227, pp. 68–83 (2017). https://doi.org/10.1007/978-3-319-57288-8_5
26. 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
27. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods in System Design **48**(3), 152–174 (Jun 2016). https://doi.org/10.1007/s10703-016-0243-x

28. Finney, H.: Bleichenbacher's RSA signature forgery based on implementation error (2006), https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/
29. Forristal, J.: Android: One Root to Own Them All (2013), https://www.youtube.com/watch?v=mCF5kaCt4NI
30. Granberry, G., Ahrendt, W., Johansson, M.: Specify what? Enhancing neural specification synthesis by symbolic methods. In: Proc. of the 19th International Conference on Integrated Formal Methods (iFM 2024). LNCS, vol. 15234, pp. 307–325. Springer (2024). https://doi.org/10.1007/978-3-031-76554-4_19
31. 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
32. Herrmann, P., Signoles, J.: RTE Runtime Error Annotation Generation (2024), https://frama-c.com/download/frama-c-rte-manual.pdf
33. Hána, M., Kosmatov, N., Prevosto, V., Signoles, J.: Formal Verification of PKCS#1 Signature Parser using Frama-C. Companion Artifact for the Paper Submitted to iFM 2025 (Aug 2025). https://doi.org/10.5281/zenodo.16919839
34. Intel Security: BERserk Vulnerability – Part 2: Certificate Forgery in Mozilla NSS (2014), https://bugzilla.mozilla.org/attachment.cgi?id=8499825
35. ISO/IEC JTC 1/SC 22: ISO/IEC 9899:1999 Programming languages — C. Standard 9899:1999, International Standard Organisation (1999), https://www.iso.org/standard/29237.html
36. ITU: X.690: Information Technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). Tech. rep. (Feb 2021), https://www.itu.int/rec/T-REC-X.690/
37. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. of the Third International Symposium on NASA Formal Methods (NFM 2011). LNCS, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
38. Janßen, C., Richter, C., Wehrheim, H.: Can ChatGPT support software verification? In: Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2024). LNCS, vol. 14573, pp. 266–279. Springer (2024). https://doi.org/10.1007/978-3-031-57259-3_13
39. Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Leveraging LLMs for program verification. In: Proc. of the 24th Conference on Formal Methods in Computer-Aided Design (FMCAD 2024). pp. 107–118. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_16
40. Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., Kivinen, T.: Internet Key Exchange Protocol Version 2 (IKEv2) (2014). https://doi.org/10.17487/RFC7296, RFC 7296
41. Kosmatov, N., Prevosto, V., Signoles, J. (eds.): Guide to Software Verification with Frama-C. Core Components, Usages, and Applications. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1
42. Kühn, U., Pyshkin, A., Tews, E., Weinmann, R.P.: Variants of Bleichenbacher's Low-exponent Attack on PKCS# 1 RSA Signatures. In: Sicherheit 2008: Sicherheit, Schutz und Zuverlässigkeit. Konferenzband der 4.

Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI) (2008), https://download.hrz.tu-darmstadt.de/pub/FB20/Dekanat/Publikationen/CDC/sigflaw.pdf
43. Leinenbach, D., Santen, T.: Verifying the microsoft Hyper-V hypervisor with VCC. In: Proc. of the Second World Congres on Formal Methods (FM 2009). LNCS, vol. 5850, pp. 806–809. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_51
44. Leino, K.R.M.: Program Proofs. The MIT Press (2023)
45. Li, L.W., Eakman, G., Garcia, E.J.M., Atman, S.: Accessible formal methods for verified parser development. In: Proc. of the IEEE Security and Privacy Workshops (SP Workshops 2021). pp. 142–151. IEEE (2021). https://doi.org/10.1109/SPW53761.2021.00028
46. Mitre: CWE-190: Integer overflow or wraparound, https://cwe.mitre.org/data/definitions/190.html
47. Moriarty, K., Kaliski, B., Jonsson, J., Rusch, A.: PKCS #1: RSA Cryptography Specifications, Version 2.2 (2016). https://doi.org/10.17487/RFC8017, RFC 8017
48. Nir, Y., Kivinen, T., Wouters, P., Migault, D.: Algorithm Implementation Requirements and Usage Guidance for the Internet Key Exchange Protocol Version 2 (IKEv2) (2017). https://doi.org/10.17487/RFC8247, RFC 8247
49. Oiwa, Y., Kobara, K., Watanabe, H.: A new variant for an attack against RSA signature verification using parameter field. In: Proc. of the 4th European Workshop on Theory and Practice of Public Key Infrastructure (EuroPKI 2007). LNCS, vol. 4582, pp. 143–153. Springer (2007). https://doi.org/10.1007/978-3-540-73408-6_10
50. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Proc. of the 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
51. Philippaerts, P., Mühlberg, J., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: Industrial case studies. Sci. Comput. Program. **82**, 77–97 (2014). https://doi.org/10.1016/J.SCICO.2013.01.006
52. Ramananandro, T., Delignat-Lavaud, A., Fournet, C., Swamy, N., Chajed, T., Kobeissi, N., Protzenko, J.: EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In: Proc. of the 28th USENIX Conference on Security Symposium (SEC 2019) (2019), https://www.microsoft.com/en-us/research/wp-content/uploads/2019/05/20190601everparse.pdf
53. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 (2018). https://doi.org/10.17487/RFC8446, RFC 8446
54. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). https://doi.org/10.1145/359340.359342
55. Swamy, N., Ramananandro, T., Rastogi, A., Spiridonova, I., Ni, H., Malloy, D., Vazquez, J., Tang, M., Cardona, O., Gupta, A.: Hardening Attack Surfaces with Formally Proven Binary Format Parsers. In: Proc. of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022) (2022). https://doi.org/10.1145/3519939.3523708
56. Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S., Tian, C.: Enchanting program specification synthesis by large language models using static analysis and program verification. In: Proc. of the 36th International Conference on

Computer Aided Verification (CAV 2024). LNCS, vol. 14682, pp. 302–328. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_16
57. Yahyazadeh, M., Chau, S.Y., Li, L., Hue, M.H., Debnath, J., Ip, S.C., Li, C.N., Hoque, E., Chowdhury, O.: Morpheus: Bringing The (PKCS) One To Meet the Oracle. In: Proc. of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS 2021). pp. 2474–2496 (2021). https://doi.org/10.1145/3460120.3485382
58. Ylonen, T., C. Lonvick, E.: The Secure Shell (SSH) Transport Layer Protocol (2006). https://doi.org/10.17487/RFC4253, RFC 4253

A Appendix: Supplementary Material

This appendix partly describes the companion artifact. It will be included in the long version of the paper published online. It includes the annotated parser code, proof scripts and a few buggy versions of the parser, which illustrate the effectiveness of the proof to detect security and compatibility issues.

A.1 The Complete PKCS#1 v1.5 Parser Code

Listing 1.1 gives the complete code of the target PKCS#1 v1.5 parser including ACSL annotations. The scope and signatures of the included functions are discussed in Sect. 4.4. The parser was successfully proven using Frama-C 29.0 (Copper) and SMT solver Alt-Ergo 2.5.4. The full command launching the proof is given in the beginning of the file. Three Frama-C **`ensures`** are not proven automatically and require additional proof scripts. They are used for instantiation of the two inductive steps, which handle constructed TLV child, namely `gh_last_tlv_constr_incl_child` and `last_tlv_constr_incl_child`). To trigger this instantiation inside the interactive Frama-C/WP GUI, we put additional asserts at the end of function `pars_PKCS1_lev1`. The related proof scripts are part of the artifact.

Listing 1.1: Complete annotated code of the PKCS#1 v1.5 parser.

```
// proven with:
// frama-c -wp -wp-rte -wp-timeout 30 -wp-session folder_with_scripts
// -wp-smoke-tests -wp-check-memory-model -warn-unsigned-overflow
// -warn-signed-downcast -warn-unsigned-downcast -wp-prover=script,Alt-Ergo
#include "limits.h"
typedef unsigned char u1;
typedef unsigned int uint;

// Parsing errors                // Unexpected:
#define ERR_B1              0x01 //First prefix byte
#define ERR_B2              0x02 //Second prefix byte
#define ERR_FF_COUNT        0x03 //Less than minimum of pad bytes
#define ERR_DELIM           0x04 //Delimiter between pad and ASN
#define ERR_TAG             0x05 //TLV Tag
#define ERR_LEN             0x06 //TLV Length
#define ERR_HASH_SZ         0x08 //Hash length
#define ERR_TLV_LEN_CONSIST 0x09 //Nesting TLV property
#define ERR_HASH_OID        0x0A //Hash identifier
#define ERR_NULL_SZ         0x0B //Hash paramater size
#define ERR_HASH_VAL        0x0C //Hash value
//Hash crypto call error
#define ERR_INTER_BAD_CRYPT 0xF2
// Nominal parsing case
#define PASS_OK             0
// Parsing offsets and lengths
#define OFF_B1              0 // Prefix padding byte 1 offset
#define OFF_B2              1 // Prefix padding byte 2 offset
#define LEN_TAGLEN          2 // Total length of 2 fields: TAG + LEN
// Expected values during parsing
#define EXP_B1              0x00 // Prefix padding byte 1
#define EXP_B2              0x01 // Prefix padding byte 2
#define EXP_FF              0xFF // Padding 0xFF sequence
#define EXP_DELIM           0x00 // Padding delimiter
#define MIN_FF_COUNT        0x08 // Minimal size of FF sequence
#define TAG_SEQ             0x30 // ASN.1 SEQUENCE Tag
#define TAG_OCT             0x04 // ASN.1 OCTET_STRING Tag
#define TAG_OID             0x06 // ASN.1 OID Tag
#define TAG_NULL            0x05 // ASN.1 NULL Tag
#define ASN_LEN_NULL        0x00 // ASN.1 NULL Length
// Supported hash functions
#define SUPP_OID_COUNT      3
#define MAX_HASH_SIZE       64
#define SHA256_SIZE         32
#define SHA512_SIZE         64
#define SHA1_SIZE           20

enum hash_functions {SHA256_INDEX, SHA512_INDEX, SHA1_INDEX};

// Element 0 contains size of OID, followed by its value
const u1 SHA256_OID[10] ={0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x01};
const u1 SHA512_OID[10] ={0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x03};
const u1 SHA1_OID[7]    ={0x05,0x2B,0x0E,0x03,0x02,0x1A};

const u1* const OID_list[SUPP_OID_COUNT] =
  {(u1*)&SHA256_OID[0], (u1*)&SHA512_OID[0], (u1*)&SHA1_OID[0]};

u1 hash_res[MAX_HASH_SIZE]; // Working buffer for hash computation

const int hash_len_for_oid[SUPP_OID_COUNT]={SHA256_SIZE, SHA512_SIZE, SHA1_SIZE};

#define OFF_COUNT           0
#define SPEC_ITEM_SZ        3
#define OFF_TAG             1
#define OFF_TYPE            2
#define OFF_LINK            3
#define PRIM                1
```

```
#define CONSTR              0

// ======= TLV structure specification =======
/*@ ghost
const uint g_tlv_1[4] = {1, TAG_SEQ, CONSTR, 1};
const uint g_tlv_2[7] = {2, TAG_SEQ, CONSTR, 2, TAG_OCT,  PRIM, 0};
const uint g_tlv_3[7] = {2, TAG_OID, PRIM,   0, TAG_NULL, PRIM, 0};

\ghost const uint* const g_tlv_spec[3] =
  { &g_tlv_1[0], &g_tlv_2[0], &g_tlv_3[0] };

u1* g_tlv_1_p[1]; u1* g_tlv_2_p[2]; u1* g_tlv_3_p[2];

u1* \ghost * const g_tlv_p[3] =
  { &g_tlv_1_p[0], &g_tlv_2_p[0], &g_tlv_3_p[0] };  */

// Correct structure of message with respect to TLV specification g_tlv_spec
/*@
// ======= Inductive predicate for correct message format =======
inductive valid_tlv(u1* start, integer prev_pars, integer size_pars,
  integer tlv_num_pars, integer tlv_id, boolean with_children)
{
case empty: \forall u1* start, integer tlv_id;
  valid_tlv(start, 0, 0, 0, tlv_id, \true);

case new_tlv_child_not_incl:
  \forall u1* start, integer prev_pars, integer size_pars,
    integer tlv_num_pars, integer tlv_id;
  \let tlv_cur = g_tlv_spec[tlv_id];
  0 <= tlv_num_pars < tlv_cur[OFF_COUNT] &&
  valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true) &&
  start[size_pars] == (u1)tlv_cur[OFF_TAG + tlv_num_pars*SPEC_ITEM_SZ]
  ==>
  valid_tlv(start, size_pars, size_pars + 2 + start[size_pars + 1],
    tlv_num_pars+1, tlv_id, \false);

case last_tlv_prim_incl_child:
  \forall u1* start, integer prev_pars, integer size_pars,
    integer tlv_num_pars, integer tlv_id;
  \let tlv_cur = g_tlv_spec[tlv_id];
  1 <= tlv_num_pars &&
  valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \false) &&
  (u1)tlv_cur[OFF_TYPE + (tlv_num_pars-1)*SPEC_ITEM_SZ] == PRIM
  ==>
  valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true);

case last_tlv_constr_incl_child:
  \forall u1* start, integer prev_pars, integer size_pars,
    integer tlv_num_pars, integer tlv_id, integer nest_prev_pars;
  \let tlv_cur = g_tlv_spec[tlv_id];
  \let tlv_link = tlv_cur[OFF_LINK + (tlv_num_pars-1)*SPEC_ITEM_SZ];
  1 <= tlv_num_pars &&
  valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \false) &&
  (u1)tlv_cur[OFF_TYPE + (tlv_num_pars-1)*SPEC_ITEM_SZ] == CONSTR &&
  valid_tlv(start + prev_pars + 2, nest_prev_pars, start[prev_pars + 1],
    g_tlv_spec[tlv_link][OFF_COUNT], tlv_link, \true)
  ==>
  valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true); }

// Supplementary predicate
// To ease instantiation of prim_gh_set manually asserted in the code
predicate prim_tlv_stored(u1* start, integer tlv_id, integer tlv_seq_id,
    integer prev_pars) =
  \let tlv_cur = g_tlv_spec[tlv_seq_id];
  \let tlv_p_curr = g_tlv_p[tlv_seq_id];
  (u1)tlv_cur[OFF_TYPE + (tlv_id-1)*SPEC_ITEM_SZ] == PRIM &&
  tlv_p_curr[tlv_id-1] == &start[prev_pars];

```

```
135 // Ghost code correctly store pointers to all primitive TLVs
136 inductive prim_gh_set (u1* start, integer prev_pars, integer size_pars,
137   integer tlv_num_pars, integer tlv_id, boolean with_children)
138 {
139 case gh_empty: \forall u1* start, integer tlv_id;
140   prim_gh_set (start, 0, 0, 0, tlv_id, \true);
141
142 case gh_new_tlv_child_not_incl:
143   \forall u1* start, integer prev_pars, integer size_pars,
144     integer tlv_num_pars, integer tlv_id;
145   \let tlv_cur = g_tlv_spec[tlv_id];
146   0 <= tlv_num_pars < tlv_cur[OFF_COUNT] &&
147   prim_gh_set(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true)
148   ==>
149   prim_gh_set(start, size_pars, size_pars + 2 + start[size_pars + 1],
150     tlv_num_pars+1, tlv_id, \false);
151
152 case gh_last_tlv_prim_incl_child:
153   \forall u1* start, integer prev_pars, integer size_pars,
154     integer tlv_num_pars, integer tlv_id;
155   \let tlv_cur = g_tlv_spec[tlv_id];
156   1 <= tlv_num_pars &&
157   prim_gh_set(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \false) &&
158   prim_tlv_stored(start, tlv_num_pars, tlv_id, prev_pars)
159   ==>
160   prim_gh_set(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true);
161
162 case gh_last_tlv_constr_incl_child:
163   \forall u1* start, integer prev_pars, integer size_pars,
164     integer tlv_num_pars, integer tlv_id, integer nest_prev_pars;
165   \let tlv_cur = g_tlv_spec[tlv_id];
166   \let tlv_link = tlv_cur[OFF_LINK + (tlv_num_pars-1)*SPEC_ITEM_SZ];
167   1 <= tlv_num_pars &&
168   prim_gh_set(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \false) &&
169   (u1)tlv_cur[OFF_TYPE + (tlv_num_pars-1)*SPEC_ITEM_SZ] == CONSTR &&
170   prim_gh_set(start + prev_pars + 2, nest_prev_pars, start[prev_pars + 1],
171     g_tlv_spec[tlv_link][OFF_COUNT], tlv_link, \true)
172   ==>
173   prim_gh_set(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true);
174 }*/
175
176 /*@ predicate same_content(u1* f, u1* s, int len) =
177   \forall integer i; 0<= i < len ==> (f[i] == s[i]);
178 */
179
180 // Stores index of used hash function
181 /*@ ghost int g_hash_app; */
182
183 /*  Primitive TLVs content
184     Refers pointers to TLVs stored inside g_tlv_p structure
185 P1. Parsed OID corresponds to inputted hash function
186 P2. Length of NULL TLV is equal to 0
187 P3. Parsed hash value length corresponds to parsed OID
188 P4. Function used for hash re-computation corresponds to parsed OID
189 P5. Hash re-computation is applied on input message M
190 P6. Parsed hash value is equal to re-computed hash value*/
191 /*@
192 // Properties P1 and P2
193 predicate prim_tlv_oid_param(integer exp_hash_ind) =
194   \let oid_len = *(g_tlv_p[2][0]+1);                          // Reads L4
195   \let oid_val_p = g_tlv_p[2][0]+2;                           // Points to V4
196   \let param_len = *(g_tlv_p[2][1]+1);                        // Reads L5
197   (oid_len == OID_list[exp_hash_ind][0]) &&                   // En-
198   same_content(OID_list[exp_hash_ind]+1, oid_val_p, oid_len) && //   sures P1
199   param_len == 0;                                             // Ensures P2
200
201 // Properties P3 + P6,
202 // P4  (together with ghost code related to
```

```
// g_hash_app)
predicate prim_tlv_hash(integer exp_hash_ind) =
  \let hash_len = *(g_tlv_p[1][1]+1); \let hash_value_p = g_tlv_p[1][1]+2;
  hash_len == hash_len_for_oid[exp_hash_ind] &&
  g_hash_app == exp_hash_ind &&
  same_content(hash_value_p, &hash_res[0], hash_len);

// prim_tlv_hash_neg is not logical negation of prim_tlv_hash
predicate prim_tlv_hash_neg(integer exp_hash_ind) =
  \let hash_len = *(g_tlv_p[1][1]+1); \let hash_value_p = g_tlv_p[1][1]+2;
  !(hash_len == hash_len_for_oid[exp_hash_ind]) ||
  (g_hash_app == exp_hash_ind &&
  !same_content(hash_value_p, &hash_res[0], hash_len));

predicate prim_tlv(uint exp_hash_ind) =
  0<= exp_hash_ind < SUPP_OID_COUNT &&
  prim_tlv_oid_param(exp_hash_ind) && prim_tlv_hash(exp_hash_ind);

predicate prim_tlv_neg(uint exp_hash_ind) =
  0<= exp_hash_ind < SUPP_OID_COUNT ==>
  !prim_tlv_oid_param(exp_hash_ind) || prim_tlv_hash_neg(exp_hash_ind);
*/

// Stores input data snapshot for later comparison
/*@ ghost
u1 g_backup_mes_to_hash[INT_MAX];
uint g_backup_len;

/@ requires \valid_read(msg + (0 .. msg_sz-1));
  terminates \true; exits \false;
  assigns g_backup_mes_to_hash[0 .. msg_sz-1];
  ensures same_content(msg, &g_backup_mes_to_hash[0], msg_sz);
  ensures g_backup_len == msg_sz; @/
void g_make_backup(u1* msg, int msg_sz); */

/*
Params:  arr1: input array 1 start
         arr2: input array 2 start
         len : length to compare
Returns: 0 if arr1 and arr2 has the same prefix of len bytes
         1 otherwise */
/*@
  requires \valid_read(arr1 + (0 .. len-1));
  requires \valid_read(arr2 + (0 .. len-1));
  requires len >=0;
  assigns \nothing;
  ensures (\result == (int)0) <==> same_content(arr1, arr2, len);
*/
int cmp_arr(const u1* arr1, const u1* arr2, int len)
{ /*@ loop assigns i;
      loop invariant 0<= i <=len;
      loop invariant \forall integer j; 0<=j< i ==> arr1[j] == arr2[j];
      loop variant len-i; */
  for (int i = 0; i < len; i++){
    if (arr1[i] != arr2[i])
      return 1; }
  return 0;
}
#define NO_HASH_APP  -1

/* For all hash stubs
Params:  msg        : input message to hash
         msg_sz     : input size
         out_hash   : output for hash
Returns: hash size */

// Precondition using same_content checks that hash is applied on
// (entire) input message of pars_PKCS1_lev1 - in_msg. Property P5.
```

```
271 /*@
272   requires \valid_read(msg + (0 ..msg_sz-1));
273   requires msg_sz >= 0;
274   requires same_content(msg, &g_backup_mes_to_hash[0], msg_sz);
275   requires g_backup_len == msg_sz;
276   requires g_hash_app == NO_HASH_APP;
277   terminates \true;
278   exits \false;
279   assigns out_hash[0 .. 31], g_hash_app;
280   ensures g_hash_app == SHA256_INDEX;
281   ensures \result == (int)32;
282 */
283 int stub_sha256(u1* msg, int msg_sz, u1* out_hash);
284
285 /*@
286   requires \valid_read(msg + (0 ..msg_sz-1));
287   requires msg_sz >= 0;
288   requires same_content(msg, &g_backup_mes_to_hash[0], msg_sz);
289   requires g_backup_len == msg_sz;
290   terminates \true;
291   exits \false;
292   assigns out_hash[0 .. 63], g_hash_app;
293   ensures g_hash_app == SHA512_INDEX;
294   ensures \result == (int)64;
295 */
296 int stub_sha512(u1* msg, int msg_sz, u1* out_hash);
297
298 /*@
299   requires \valid_read(msg + (0 ..msg_sz-1));
300   requires msg_sz >= 0;
301   requires same_content(msg, &g_backup_mes_to_hash[0], msg_sz);
302   requires g_backup_len == msg_sz;
303   terminates \true;
304   exits \false;
305   assigns out_hash[0 .. 19], g_hash_app;
306   ensures g_hash_app == SHA1_INDEX;
307   ensures \result == (int)20;
308 */
309 int stub_sha1(u1* msg, int msg_sz, u1* out_hash);
310
311 /*Reads Tag and Length of next TLV
312 Params:  msg            : start of TLV sequence
313          OUTPUT:num_read: already consumed bytes of msg
314          num_to_read    : size of TLV sequence in bytes
315          exp_tag        : next expected Tag
316          OUTPUT:tlv_size: size of read TLV
317 Returns: ERR_LEN        : there is not enough space for next TLV
318          ERR_TAG        : if read Tag is not expected
319          0              : otherwise */
320 /*@
321   requires \valid_read(msg + (0 .. num_to_read - 1));
322   requires \valid(num_read);
323   requires \valid(tlv_size);
324   requires \separated(num_read, tlv_size, msg + (0 .. num_to_read - 1));
325   requires num_to_read >= 0;
326   requires 0 <= *num_read <= INT_MAX - 2;
327   assigns *num_read, *tlv_size;
328   ensures \result == 0 || \result == ERR_LEN || \result == ERR_TAG;
329   ensures \result == 0 ==> \old(*num_read) + LEN_TAGLEN <= num_to_read;
330   ensures \result == 0 ==> msg[\old(*num_read)] == exp_tag;
331   ensures \result == 0 ==> *num_read == \old(*num_read) + 2;
332   ensures \result == 0 ==> *tlv_size == msg[\old(*num_read) + 1];
333   ensures \result == ERR_LEN ==> \old(*num_read) + LEN_TAGLEN > num_to_read;
334   ensures \result == ERR_TAG ==> msg[\old(*num_read)] != exp_tag;
335 */
336 int read_one_tlv(u1* msg, int* num_read, int num_to_read, uint exp_tag,
337   int *tlv_size)
338 {
```

```
  // there is enough place for TAG and LEN fields
  if (*num_read + LEN_TAGLEN > num_to_read)
  {  return ERR_LEN;}

  if(msg[(*num_read)++]!= exp_tag)
  {  return ERR_TAG;}

  *tlv_size = msg[(*num_read)++];
  return 0; }

/* Parse TLV level 2 and 3,
i.e. DigestInfo and DigestAlgorithm Values
Params:  dig_inf        : message to parse, DigestInfo Value
         dig_inf_sz     : size of dig_inf
         msg_to_hash    : message to hash
         msg_sz         : size of msg_to_hash
         expect_hash_ind: hash to be used for match
Returns: ERR_LEN        : there is not enough space for next TLV
         ERR_TAG        : read Tag is not expected
         ERR_TLV_LEN_CONSIST: sizes of parent and children TLVs are not
           consistent
         ERR_HASH_SZ    : parsed hash size does not match parsed hash
           function
         ERR_HASH_OID   : parsed hash function is not expected
         ERR_NULL_SZ    : NULL TLV does not have 0 size
         ERR_HASH_VAL   : recomputed hash does not match parsed hash
         PASS_OK        : otherwise */
/*@
requires \valid(dig_inf + (0 .. dig_inf_sz-1));
requires 0<= dig_inf_sz <= INT_MAX;
requires \valid(msg_to_hash + (0 .. msg_sz-1));
requires 0<= msg_sz <= INT_MAX;
requires \separated(dig_inf + (0 ..), hash_res + (0 ..), hash_len_for_oid + (0..),
  OID_list + (0..), SHA256_OID +(0 ..), SHA512_OID +(0 ..), SHA1_OID +(0 ..));

requires 0 <= expect_hash_ind < SUPP_OID_COUNT;

//check that we pass (entire) input message of pars_PKCS1_lev1
requires same_content(msg_to_hash, &g_backup_mes_to_hash[0], msg_sz);
requires g_backup_len == msg_sz;

requires g_hash_app == NO_HASH_APP;

assigns g_tlv_p[1][1], g_tlv_p[2][0..1], g_hash_app;
assigns hash_res[0..63];

ensures \result==ERR_LEN || \result==ERR_TAG || \result==ERR_TLV_LEN_CONSIST ||
  \result == ERR_HASH_SZ || \result == ERR_HASH_OID || \result == ERR_NULL_SZ ||
  \result == ERR_HASH_VAL || \result == PASS_OK;

//SECURITY PART
behavior sec_format:
  ensures format: \result == PASS_OK ==>
    \exists integer prev_pars; valid_tlv(dig_inf, prev_pars, dig_inf_sz,
    g_tlv_2[0], 1, \true);

behavior sec_prim_tlv:
  ensures ghost_set: \result == PASS_OK ==>
    \exists integer prev_pars; prim_gh_set(dig_inf, prev_pars, dig_inf_sz,
    g_tlv_2[0], 1, \true);
  ensures prim_tlv:(\result == PASS_OK) ==> prim_tlv(expect_hash_ind);

//COMPATIBILITY (ACCEPTANCE) PART
behavior acc_format:
  ensures format: \result == ERR_LEN || \result == ERR_TAG ||
    \result == ERR_TLV_LEN_CONSIST ==>
    (!(\exists integer prev_pars; valid_tlv(dig_inf, prev_pars, dig_inf_sz,
    g_tlv_2[0], 1, \true)));
```

```
407
408 behavior acc_prim_tlv:
409   ensures ghost_set: \result == ERR_HASH_SZ || \result == ERR_HASH_OID ||
410     \result == ERR_HASH_VAL || \result == ERR_NULL_SZ ==>
411     \exists integer prev_pars; prim_gh_set(dig_inf, prev_pars, dig_inf_sz,
412     g_tlv_2[0], 1, \true);
413   ensures prim_tlv: \result == ERR_HASH_SZ || \result == ERR_HASH_OID ||
414     \result == ERR_HASH_VAL || \result == ERR_NULL_SZ ==>
415     prim_tlv_neg(expect_hash_ind);
416 */
417 int pars_PKCS1_lev2(u1* dig_inf, int dig_inf_sz, u1* msg_to_hash, int msg_sz,
418   uint expect_hash_ind)
419 {
420   // Parsing of TLV Level 2
421   // ======= LEVEL 2 =======
422
423   int seq_l2_len, off = 0;
424   int res = read_one_tlv(dig_inf, &off, dig_inf_sz, TAG_SEQ, &seq_l2_len);
425   if(res!=0) return res;
426
427   // Remember offset of OID's TAG field
428   int oid_off = off;
429
430   // Skip to OCTET_STRING TAG (contains hash) and store it into ghosts
431   off += seq_l2_len;
432   // Store pointer to primitive TLV inside ghost array
433   /*@ ghost g_tlv_p[1][1] = &dig_inf[off]; */
434   // To help instantiation of prim_gh_set
435   /*@ for sec_prim_tlv, acc_prim_tlv:
436   assert prim_tlv_stored(dig_inf, 2, 1, off); */
437
438   int hash_len;
439   res = read_one_tlv(dig_inf, &off, dig_inf_sz, TAG_OCT, &hash_len);
440   if(res!=0) return res;
441
442   // Remember offset of hash VALUE field
443   int hash_val_off = off;
444
445   //TLV LEVEL 2 nesting property
446   if (off + hash_len!= dig_inf_sz)
447   {
448     return ERR_TLV_LEN_CONSIST;
449   }
450
451   // If LEVEL 2 consistent, we continue to LEVEL 3
452   // ======= LEVEL 3 =======
453
454   u1* oid_start = &dig_inf[oid_off];
455   off = 0;
456
457   /*@ ghost g_tlv_p[2][0] = &oid_start[off]; */
458   /*@ for sec_prim_tlv, acc_prim_tlv:
459   assert prim_tlv_stored(oid_start, 1, 2, off);*/
460
461   int oid_len;
462   res = read_one_tlv(oid_start, &off, seq_l2_len, TAG_OID, &oid_len);
463   if(res!=0) return res;
464
465   int oid_val_off = off;
466   off += oid_len;
467
468   /*@ ghost g_tlv_p[2][1] = &oid_start[off]; */
469   /*@ for sec_prim_tlv, acc_prim_tlv:
470   assert prim_tlv_stored(oid_start, 2, 2, off); */
471
472   int null_size;
473   res = read_one_tlv(oid_start, &off, seq_l2_len, TAG_NULL, &null_size);
474   if(res!=0)
```

```
475     /*@ for acc_format: assert \forall integer prev_pars;
476       !valid_tlv(dig_inf,prev_pars,dig_inf_sz,g_tlv_2[0], 1,\true);*/
477     return res;
478
479   if(null_size + off !=  seq_l2_len)
480     {/*@ for acc_format: assert (\forall integer prev_pars;
481       !valid_tlv(dig_inf,prev_pars,dig_inf_sz,g_tlv_2[0], 1,\true));*/
482     return ERR_TLV_LEN_CONSIST;}
483
484   // Init inductive step for TLV level 2 and 3
485   // To confirm correct pointers setting if null_size is wrong
486   /*@ for sec_prim_tlv, acc_prim_tlv:
487   assert prim_gh_set(oid_start, 0, 0, 0, 2, \true); */
488   /*@ for sec_prim_tlv, acc_prim_tlv:
489   assert prim_gh_set(dig_inf, 0, 0, 0, 1, \true); */
490
491   if(null_size != 0)
492     return ERR_NULL_SZ;
493
494   // If TLV structure is OK, we continue to TLV content
495   // ======= PRIMITIVE TLVs CHECKS =======
496
497   if (oid_len == OID_list[expect_hash_ind][0] &&
498   !cmp_arr(&oid_start[oid_val_off], &OID_list[expect_hash_ind][1], oid_len))
499   {
500     if (hash_len != hash_len_for_oid[expect_hash_ind])
501     {
502       return ERR_HASH_SZ;
503     }
504
505     int hash_out_size =0;
506     // Re-hash input message
507     switch(expect_hash_ind){
508       case SHA256_INDEX:
509         hash_out_size = stub_sha256(msg_to_hash, msg_sz, hash_res);
510         break;
511       case SHA512_INDEX:
512         hash_out_size = stub_sha512(msg_to_hash, msg_sz, hash_res);
513         break;
514       case SHA1_INDEX:
515         hash_out_size = stub_sha1(msg_to_hash, msg_sz, hash_res);
516         break;
517     }
518
519     if(hash_out_size!= hash_len_for_oid[expect_hash_ind])
520       //@ assert dead1: \false;
521       return ERR_INTER_BAD_CRYPT;
522
523     // Due to stub_shaX writing, we move valid_tlv and prim_gh_set
524     // initial steps here. For prim_gh_set we repeat the init.
525     /*@ for sec_format: //initial step for Level 2
526     assert valid_tlv(dig_inf, 0, 0, 0, 1, \true); */
527     /*@ for sec_format: //initial step for Level 3
528     assert valid_tlv(oid_start, 0, 0, 0, 2, \true); */
529     /*@ for sec_prim_tlv, acc_prim_tlv: //initial step for Level 2
530     assert prim_gh_set(oid_start, 0, 0, 0, 2, \true); */
531     /*@ for sec_prim_tlv, acc_prim_tlv: //initial step for Level 3
532     assert prim_gh_set(dig_inf, 0, 0, 0, 1, \true); */
533
534     // compare re-computed and parsed hashes
535     if (cmp_arr(&dig_inf[hash_val_off], hash_res, hash_out_size))
536     //@ for acc_prim_tlv: assert prim_tlv_neg(expect_hash_ind);
537     /*@ for acc_prim_tlv: assert (\exists integer prev_pars;
538     prim_gh_set(dig_inf, prev_pars, dig_inf_sz, g_tlv_2[0],
539       1, \true)); */
540       return ERR_HASH_VAL;
541     else
542       // nominal case
```

```
543       return PASS_OK;
544   }
545   /*@ for acc_prim_tlv: assert (\exists integer prev_pars;
546     prim_gh_set(dig_inf,prev_pars,dig_inf_sz,g_tlv_2[0],1,\true));*/
547   //@ for acc_prim_tlv: assert prim_tlv_neg(expect_hash_ind);
548   return ERR_HASH_OID;
549 };
550
551 // prefix_prop    :  2 prefix bytes
552 // padding_prop   :  sequence of 0xFF and delimiter
553 // asn_prop       :  top level inductive predicate valid_tlv
554 // ghost_set_prop :  top level inductive predicate prim_gh_set
555 // Boundary between 0xFF padding and ASN.1
556 /*@ ghost uint pad_bound;*/
557 /*@ predicate prefix_prop(u1* pad_msg) =
558   pad_msg[OFF_B1] == EXP_B1 && pad_msg[OFF_B2] == EXP_B2;
559 predicate padding_prop(u1* pad_msg, integer pad_msg_sz, integer pad_bound)=
560   2+ MIN_FF_COUNT <= pad_bound < pad_msg_sz &&
561   (\forall integer padd_off; 2 <= padd_off < pad_bound ==>
562     pad_msg[padd_off] == EXP_FF) && pad_msg[pad_bound] == EXP_DELIM;
563 predicate asn_prop(u1* pad_msg, integer pad_msg_sz, integer pad_bound,
564   integer tlv_num) =
565   2+ MIN_FF_COUNT <= pad_bound < pad_msg_sz &&
566   \exists integer prev_pars; valid_tlv(pad_msg + pad_bound + 1, prev_pars,
567     pad_msg_sz - pad_bound - 1, tlv_num, 0, \true);
568 predicate ghost_set_prop(u1* pad_msg, integer pad_msg_sz, integer pad_bound,
569   integer tlv_num)=
570   2+ MIN_FF_COUNT <= pad_bound < pad_msg_sz &&
571   \exists integer prev_pars; prim_gh_set(pad_msg + pad_bound + 1, prev_pars,
572     pad_msg_sz - pad_bound -1, tlv_num, 0, \true); */
573
574 /*Parse prefix padding and TLV level 1
575 Params:  pad_msg        : message to parse, DigestInfo TLV
576          pad_msg_sz     : size of pad_msg
577          in_msg         : message to hash
578          in_msg_sz      : size of in_msg
579          expect_hash_ind: hash to be used for match
580 Returns: ERR_B1         : first byte has unexpected value
581          ERR_B2         : second byte has unexpected value
582          ERR_FF_COUNT   : sequence of 0xFF bytes is shorter than 8
583          ERR_DELIM      : there is unexpected delimiter
584          ERR_LEN        : there is not enough space for next TLV
585          ERR_TAG        : read Tag is not expected
586          ERR_TLV_LEN_CONSIST: sizes of parent and children TLVs are not
587            consistent
588          ERR_HASH_SZ    : parsed hash size doesn't match parsed hash
589            function
590          ERR_HASH_OID   : parsed hash function is not expected
591          ERR_NULL_SZ    : NULL TLV doesn't have 0 size
592          ERR_HASH_VAL   : recomputed hash doesn't match parsed hash
593          PASS_OK        : otherwise */
594 /*@
595 requires \valid(pad_msg + (0 .. pad_msg_sz-1));
596 requires \valid(in_msg + (0 .. in_msg_sz - 1));
597 requires 0 <= expect_hash_ind < SUPP_OID_COUNT;
598 requires 0 <= pad_msg_sz <= INT_MAX- 255;
599 requires 0 <= in_msg_sz <= INT_MAX;
600 requires \separated(pad_msg + (0 ..), in_msg + (0 ..), hash_res + (0 ..),
601   hash_len_for_oid + (0..), OID_list + (0..), SHA256_OID +(0 ..),
602   SHA512_OID +(0 ..), SHA1_OID +(0 ..));
603
604 assigns pad_bound, g_tlv_p[0][0], g_tlv_p[1][0..1], g_tlv_p[2][0..1], g_hash_app,
605   g_backup_mes_to_hash[0 .. in_msg_sz-1];
606 assigns hash_res[0..63];
607
608 ensures \result == ERR_B1 ||\result == ERR_B2 || \result == ERR_FF_COUNT ||
609   \result == ERR_DELIM || \result == ERR_LEN || \result == ERR_TAG  ||
610   \result == ERR_TLV_LEN_CONSIST  || \result == ERR_HASH_SZ ||
```

```
611   \result == ERR_HASH_OID || \result == ERR_NULL_SZ || \result == ERR_HASH_VAL ||
612   \result == PASS_OK;
613
614 // ======= SECURITY PART =======
615 behavior sec_format:
616   ensures prefix: \result == PASS_OK ==> prefix_prop(pad_msg);
617   ensures pad: \result == PASS_OK ==>
618     padding_prop(pad_msg, pad_msg_sz, pad_bound);
619   ensures asn: \result == PASS_OK ==>
620     asn_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
621
622 behavior sec_prim_tlv:
623   ensures ghost_set: \result == PASS_OK ==>
624     ghost_set_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
625   ensures prim_tlv:  \result == PASS_OK ==> prim_tlv(expect_hash_ind);
626
627 // ======= COMPATIBILITY (ACCEPTANCE) PART =======
628 behavior acc_format:
629   ensures prefix:
630     \result == ERR_B1||\result == ERR_B2 ==> !prefix_prop(pad_msg);
631   ensures pad:
632     \result == ERR_FF_COUNT||\result == ERR_DELIM ==>
633     \forall uint pad_bound; !padding_prop(pad_msg, pad_msg_sz, pad_bound) ||
634     !asn_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
635   ensures asn:
636     \result == ERR_LEN||\result == ERR_TAG||\result == ERR_TLV_LEN_CONSIST ==>
637     \forall uint pad_bound; !padding_prop(pad_msg, pad_msg_sz, pad_bound) ||
638     !asn_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
639
640 behavior acc_prim_tlv:
641   ensures ghost_set:
642     \result == ERR_HASH_SZ||\result == ERR_HASH_OID||\result == ERR_HASH_VAL||
643     \result == ERR_NULL_SZ ==>
644     ghost_set_prop(pad_msg, pad_msg_sz, pad_bound, g_tlv_1[0]);
645   ensures prim_tlv:
646     \result == ERR_HASH_SZ||\result == ERR_HASH_OID||\result == ERR_HASH_VAL||
647     \result == ERR_NULL_SZ ==> prim_tlv_neg(expect_hash_ind);
648 */
649 int pars_PKCS1_lev1(u1* pad_msg, int pad_msg_sz, u1* in_msg, int in_msg_sz,
650   uint expect_hash_ind)
651 {
652   //@ ghost g_make_backup(in_msg, in_msg_sz);
653   //@ ghost g_hash_app = NO_HASH_APP;
654
655   int dig_siz;
656   if(pad_msg_sz < 2)
657     return ERR_LEN;
658
659   int off = OFF_B1;
660   if (pad_msg[off++] != EXP_B1)
661     return ERR_B1;
662   if (pad_msg[off++] != EXP_B2)
663     return ERR_B2;
664
665   int ff_start = off;
666   /*@
667     loop assigns off;
668     loop invariant \at(off, LoopEntry) <= off <= pad_msg_sz;
669     loop invariant \forall integer i; (\at(off, LoopEntry) <= i < off) ==>
670       (pad_msg[i] == EXP_FF);
671     loop variant pad_msg_sz - off;
672   */
673   while ((off < pad_msg_sz) && (pad_msg[off] == EXP_FF))
674     off++;
675   if ((off - ff_start < MIN_FF_COUNT) || (off >= pad_msg_sz - 2))
676   { return ERR_FF_COUNT; }
677
678   if (pad_msg[off++] != EXP_DELIM)
```

```
679     return ERR_DELIM;
680
681   // Store boundary between 0xFF padding and ASN.1
682   /*@ ghost pad_bound = off -1;*/
683
684   if (pad_msg[off++] != TAG_SEQ)
685     return ERR_TAG;
686
687   dig_siz = pad_msg[off++];
688
689   if (dig_siz + off != pad_msg_sz)
690     return ERR_TLV_LEN_CONSIST;
691
692   // Parse DigestInfo Value (TLV level 2 and 3)
693   int ret = pars_PKCS1_lev2(&pad_msg[off], dig_siz, in_msg, in_msg_sz,
694     expect_hash_ind);
695
696   // Initial inductive step for Level 1
697   /*@ assert valid_tlv(pad_msg + pad_bound + 1, 0, 0, 0, 0, \true); */
698   // used for script for sec_asn (instantiation of inductive step)
699   /*@ for sec_format: assert
700   \forall u1* start, integer prev_pars, integer size_pars,
701     integer tlv_num_pars, integer tlv_id, integer nest_prev_pars;
702   \let tlv_cur = g_tlv_spec[tlv_id];
703   \let tlv_link = tlv_cur[OFF_LINK + (tlv_num_pars-1)*SPEC_ITEM_SZ];
704
705   ((1 <= tlv_num_pars) &&
706   valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \false) &&
707   ((u1)tlv_cur[OFF_TYPE + (tlv_num_pars-1)*SPEC_ITEM_SZ] == CONSTR) &&
708   valid_tlv(start + prev_pars + 2, nest_prev_pars, start[prev_pars + 1],
709     g_tlv_spec[tlv_link][OFF_COUNT], tlv_link, \true)
710   ) ==>
711   valid_tlv(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true);
712   */
713
714   // Initial inductive step for Level 1
715   /*@ for sec_prim_tlv, acc_prim_tlv: assert prim_gh_set(pad_msg + pad_bound + 1,
716     0, 0, 0, 0, \true); */
717   // used for script for sec_ghost_set, acc_ghost_set
718   // (instantiation of inductive step)
719   /*@ for sec_prim_tlv, acc_prim_tlv: assert
720   \forall u1* start, integer prev_pars, integer size_pars, integer tlv_num_pars,
721     integer tlv_id, integer nest_prev_pars;
722   \let tlv_cur = g_tlv_spec[tlv_id];
723   \let tlv_link = tlv_cur[OFF_LINK + (tlv_num_pars-1)*SPEC_ITEM_SZ];
724
725   ((1 <= tlv_num_pars) &&
726   prim_gh_set(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \false) &&
727   ((u1)tlv_cur[OFF_TYPE + (tlv_num_pars-1)*SPEC_ITEM_SZ] == CONSTR) &&
728   prim_gh_set(start + prev_pars + 2, nest_prev_pars, start[prev_pars + 1],
729   g_tlv_spec[tlv_link][OFF_COUNT], tlv_link, \true)
730   ) ==>
731   prim_gh_set(start, prev_pars, size_pars, tlv_num_pars, tlv_id, \true);
732   */
733
734   return ret;
735 }
```

A.2 Bug Detection Capability of the Proof

As explained in Sect. 5, we verify that if the parser accepts a padded message, it checks its every byte and ensures its uniqueness for every input message and hash function, according to [47]. This already gives a strong guarantee of absence

of a Bleichenbacher-like attack. However, to further strengthen trust in the correctness of our formal specification, we applied the proof on 4 parser variants (detailed in Sects. A.3–A.6), which break $\mathcal{R}_{\text{sec}}$ and contain an exploitable bug. The resulting proof failures confirm that our specification cannot be proven for such buggy parsers. Since the specification can theoretically remain unproven for other reasons (such as a bug in the specification or a difficulty to perform automatic proof), we add a comment justifying why the failed clause is exactly the expected one. We used our parser and inserted bugs, which open the same kinds of *exploitable* vulnerabilities as the ones reported in the past on some established open source libraries. For the reader's convenience, for each bug we show here an extract of the modified code with original code in comments.

Additionally, we tried to run the proof on 2 parser variants (detailed in Sects. A.7–A.8), which were modified to break $\mathcal{R}_{\text{comp}}$. The resulting proof failures confirm that our specification cannot be proven for such buggy parsers.

In the code extracts below, we clearly label if the parser variant violates $\mathcal{R}_{\text{sec}}$ or $\mathcal{R}_{\text{comp}}$ or both. The full code of the original parser can be found in Sect. A.1. The full code of all parser versions (the correct and the buggy ones) is part of the artifact.

A.3 $\mathcal{R}_{\text{sec}}$ violation—Bug1: Ignored trailing bytes after the ASN.1 structure.

The relevant code extract is shown in Fig. 8.

```
684   if (pad_msg[off++] != TAG_SEQ)
685     return ERR_TAG;
686
687   dig_siz = pad_msg[off++];
688
689   //ORIGINAL:
690   //if (dig_siz + off != pad_msg_sz)
691   //BUGGY:
692   if (dig_siz + off > pad_msg_sz)
693     return ERR_TLV_LEN_CONSIST;
694
695   // Parse DigestInfo Value (TLV level 2 and 3)
696   int ret = pars_PKCS1_lev2(&pad_msg[off], dig_siz, in_msg, in_msg_sz,
697     expect_hash_ind);
```

Fig. 8: $\mathcal{R}_{\text{sec}}$ violation—Bug1: the parser ignores trailing bytes after the ASN.1 structure.

We modify the check on the TLV at Level 1, which ensures the alignment of the ASN.1 structure with the end of the padded message. As a result, the parser ignores any byte stored after the ASN.1 end. It is the original vulnerability reported by Bleichenbacher [28].

Our specification is *not provable* for 3 ensures clauses of function `pars_PKCS1_lev1` (where the Level 1 TLV is handled). It is `asn` of `sec_format` behavior, and `ghost_set` of `sec_prim_tlv` and `acc_prim_tlv` behaviors. It is exactly the set of clauses which use inductive predicates `tlv_valid` and

prim_gh_set to express a correct message format. Notice that the modified check still ensures valid memory for the Level 2 and 3 TLV parsing (realized by pars_PKCS1_lev2) and does not affect its structural properties. Therefore, all RTE checks and contract of pars_PKCS1_lev2 remain proven.

A.4 $\mathcal{R}_{\text{sec}}$ violation—Bug2: Missing check of NULL TLV length (L_5).

The relevant code extract is shown in Fig.9.

```
473   res = read_one_tlv(oid_start, &off, seq_l2_len, TAG_NULL, &null_size);
474   if(res!=0)
475     /*@ for acc_format: assert \forall integer prev_pars;
476       !valid_tlv(dig_inf,prev_pars,dig_inf_sz,g_tlv_2[0], 1,\true);*/
477     return res;
478
479   if(null_size + off !=  seq_l2_len)
480     {/*@ for acc_format: assert (\forall integer prev_pars;
481       !valid_tlv(dig_inf,prev_pars,dig_inf_sz,g_tlv_2[0], 1,\true));*/
482     return ERR_TLV_LEN_CONSIST;}
...
491   //ORIGINAL:
492   //if(null_size != 0)
493     //return ERR_NULL_SZ;
494   //BUGGY: (original code is removed)
```

Fig. 9: $\mathcal{R}_{\text{sec}}$ violation—Bug2: Missing check of NULL TLV length (L_5).

The check for NULL TLV length (L_5) is skipped (see line 492). Although $\mathcal{P}_{\text{tree}}$ is checked on line 479, the content of NULL TLV value $V5$ is ignored. It is the vulnerability reported in [42, 49].

Our specification is not provable inside the pars_PKCS1_lev2 function, more precisely, the prim_tlv ensures clause of sec_prim_tlv behavior. It is indeed the clause expressing the primitive TLVs content, including expected zero length of L_5. Also we did check a correct message format, which explains why dedicated clauses remain proven.

A.5 $\mathcal{R}_{\text{sec}}$ violation—Bug3: Ignored trailing bytes inside OID TLV (V_4).

The relevant code extract is shown in Fig.10.

The parsed OID $V4$ and the expected OID (stored inside an internal array OID_list) are compared using the length of the latter (see lines 501–502). Next to that, the check of the parsed OID length $L4$ is skipped (on line 498). As a result, if $V4$ prefix matches the expected value, its remaining bytes are ignored. It is the vulnerability reported in [15].

Our specification is not provable for the same clause as for Bug2. Next to that, the bug is also detected by one supplementary annotation, i.e. a precondition of cmp_arr. Indeed, it is not guaranteed that the expected OID length used for comparison is not higher than $L4$. The rest of the justification is the same as for Bug2.

```
495   // ======= PRIMITIVE TLVs CHECKS =======
496
497   //ORIGINAL:
498   //if (oid_len == OID_list[expect_hash_ind][0] &&
499   //!cmp_arr(&oid_start[oid_val_off], &OID_list[expect_hash_ind][1], oid_len))
500   //BUGGY:
501   if (!cmp_arr(&oid_start[oid_val_off], &OID_list[expect_hash_ind][1],
502     OID_list[expect_hash_ind][0]))
503   {
504     if (hash_len != hash_len_for_oid[expect_hash_ind])
505     {
506       return ERR_HASH_SZ;
507     }
```

Fig. 10: $\mathcal{R}_{\text{sec}}$ violation—Bug3: Ignored trailing bytes inside OID TLV (V_4).

A.6 $\mathcal{R}_{\text{sec}}$ violation—Bug4: Ignored prefix padding bytes.

The relevant code extract is shown in Fig.11.

```
673   //ORIGINAL:
674   //while ((off < pad_msg_sz) && (pad_msg[off] == EXP_FF))
675   //BUGGY:
676   while ((off < pad_msg_sz) && (pad_msg[off] != EXP_DELIM))
677     off++;
678   if ((off - ff_start < MIN_FF_COUNT) || (off >= pad_msg_sz - 2))
679   { return ERR_FF_COUNT; }
680
681   if (pad_msg[off++] != EXP_DELIM)
682     return ERR_DELIM;
```

Fig. 11: $\mathcal{R}_{\text{sec}}$ violation—Bug4: Ignored prefix padding bytes.

The prefix padding is parsed by searching for the delimiter (byte 0x00), but without ensuring that only 0xFF bytes are present. As a result, any bytes except 0x00 are ignored inside the padding. It is the vulnerability reported in [57].

Our specification is not provable inside function `pars_PKCS1_lev1` for the contained loop invariant, which specifies the padding content. Next to that, this bug makes the branch on line 682 a dead code, with one corresponding failed smoke test. Some of $\mathcal{R}_{\text{sec}}$ clauses are dependent on the loop invariant and remain conditionally proven.

A.7 $\mathcal{R}_{\text{comp}}$ violation—Bug5: Restricted set of accepted hash functions.

The relevant code extract is shown in Fig.12.

On line 504, there is an additional condition, requiring the hash size to be at least 32 (excluding SHA-1).

Our specification is not provable inside function `pars_PKCS1_lev2`, more precisely, the `prim_tlv` ensures of `acc_prim_tlv` behavior. Indeed, if `expect_hash_ind` corresponds to SHA-1, the parser refuses the message even if it does not falsify a correct message content. Next to that, this bug makes unreachable the branch which re-computes SHA-1 over input message M, with two corresponding failed smoke tests.

```
500     //ORIGINAL:
501     //if (hash_len != hash_len_for_oid[expect_hash_ind])
502     //BUGGY:
503         if ((hash_len != hash_len_for_oid[expect_hash_ind]) ||
504           (hash_len < 32))
505     {
506       return ERR_HASH_SZ;
507     }
```

Fig. 12: $\mathcal{R}_{\text{comp}}$ violation—Bug5: Restricted set of accepted hash functions.

A.8 $\mathcal{R}_{\text{sec}}$ and $\mathcal{R}_{\text{comp}}$ violation—Bug6: Compare against a wrong hash value.

The relevant code extract is shown in Fig.13.

```
534     // compare re-computed and parsed hashes
535     //ORIGINAL:
536         //if (cmp_arr(&dig_inf[hash_val_off], hash_res, hash_out_size))
537     //BUGGY:
538         if (cmp_arr(&dig_inf[hash_val_off-1], hash_res, hash_out_size))
539     //@ for acc_prim_tlv: assert prim_tlv_neg(expect_hash_ind);
540     /*@ for acc_prim_tlv: assert (\exists integer prev_pars;
541     prim_gh_set(dig_inf, prev_pars, dig_inf_sz, g_tlv_2[0],
542       1, \true)); */
543       return ERR_HASH_VAL;
544     else
545       // nominal case
546       return PASS_OK;
```

Fig. 13: $\mathcal{R}_{\text{sec}}$ and $\mathcal{R}_{\text{comp}}$ violation—Bug6: Compare against a wrong hash value.

Due to a shifted offset(see line 538), a wrong part of the padded message is compared against the re-computed hash value. (Comparison actually starts on $L3$ and not on $V3$). Because of this bug we accept a wrong value and refuse the correct one, and so both $\mathcal{R}_{\text{comp}}$ and $\mathcal{R}_{\text{sec}}$ are not fulfilled. Note that this bug is not exploitable, mainly because it leaves only 1B (at the end of $V3$) ignored.

Our specification is not provable inside function `pars_PKCS1_lev2`. As expected, we cannot prove the clause expressing the correct hash value comparison. For $\mathcal{R}_{\text{sec}}$, it is the `prim_tlv` clause of the `sec_prim_tlv` behavior. For $\mathcal{R}_{\text{comp}}$, our specification is not proven for assert supporting verification of the clause `prim_tlv`, which is part of the `acc_prim_tlv` behavior.