From Manual to LLM-Assisted Formal Verification of a PKCS#1 Signature Parser

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. Software engineering approaches assisted by Artificial Intelligence (AI) are receiving increasing attention. This work evaluates the ability of large language models (LLMs) to assist engineers in the formal verification process. We conduct an experiment using ChatGPT on a PKCS#1 (Public-Key Cryptography Standard) v1.5 signature parser that was recently (manually) specified in ACSL and verified with the Frama-C platform. Our approach prompts the LLM to generate ACSL annotations directly from the C code, cross-checks the generated specifications against known security requirements, and iteratively refines them to achieve a complete proof. In this verification loop, annotations are progressively improved using feedback from Frama-C. For failing proof obligations, we instruct the LLM to supply a sequence of ACSL assertions that guide the prover. Using structured prompts, we repeatedly obtain a complete proof for the target parser. This demonstrates that substantial parts of the formal verification workflow can be mechanized. We present our LLM-assisted verification approach, report its benefits and limitations, and compare it with the prior manual proof in terms of specification auditability and human effort.

Keywords: AI-assisted verification · deductive verification · security properties · PKCS#1 signature parser · Frama-C · ChatGPT

1 Introduction

The growing number of sensitive operations in the digital domain makes it increasingly difficult to guarantee security through manual expert review alone. Formal verification of security-critical software provides a high-assurance approach capable of ruling out entire classes of attacks. However, it requires substantial investment due to the significant manual effort demanded from formal methods experts, which limits its broader adoption in industry. Software verification approaches assisted by Artificial Intelligence (AI) are therefore promising and are attracting increasing attention.

Goals and Approach. The purpose of this work is to evaluate the ability of large language models (LLMs) to assist verification engineers and security experts in the formal verification of critical code. We conduct an experiment on the C implementation of a parser for PKCS#1 v1.5 signatures [15]. Recently, we formally verified this signature parser [8] with the Frama-C verification platform [13] in order to prove that it is secure against variants of Bleichenbacher's signature forgery attack [5], a long-standing vulnerability that has affected several open-source libraries. The verified parser consists of 155 lines of C code. It includes seven C functions (including three stubs) and was annotated with 364 lines of ACSL (ANSI/ISO C Specification Langage). In the present work, our goal is to reproduce the proof using an LLM and to compare manual and LLM-assisted verification approaches.

We selected a GPT-based large language model accessed via the ChatGPT interface (Plus subscription), with enhanced reasoning capabilities enabled. This choice provides rapid access to a state-of-the-art reasoning model, suitable for complex tasks such as formal verification, at the cost of reduced control over model behavior and limited reproducibility compared to API-based access. Evaluation using API-based deployments and alternative LLMs is left for future work. Due to the rapid evolution of the underlying models, we conducted experiments across multiple versions, ranging from GPT-5.1 to GPT-5.4, after verifying that their training data did not include our recent manual proof.

We evaluate the ability of the LLM to generate a formal specification from C code, and assess whether the resulting specification captures the required security properties, in particular resistance to various forms of signature forgery attacks. This experiment was conducted on both the correct version and several buggy variants of the parser, which were deliberately modified to introduce vulnerabilities. This setup allows us to observe whether the LLM reproduces bugs present in the code within the generated formal specification.

We also assess other important qualities of LLM-generated ACSL annotations: *auditability*—the degree to which the annotation facilitates reliably correct human review—and *provability*—the complexity involved in successfully completing the formal verification of the ACSL annotation. We describe how to avoid common pitfalls by providing high-level guidelines to the LLM. In particular, we instruct the LLM to avoid ACSL constructs that typically require significant manual effort to prove. Since a human review of (parts of) the formal specification is required to ensure the informal requirements are adequately transcribed, we place special emphasis on a clear, human-readable structure of ACSL annotations to facilitate auditability.

We also investigate prompting techniques during *proof debugging*, a labor-intensive step that must be completed after introducing (candidate) formal specification. In several independent experiments, we completed proofs by using LLM-generated assertions to bridge proof steps that were too difficult for the prover to complete automatically. During this step, auditability is less of a concern: a failure to complete a proof here does not lead to an inadequate specification.

Our prompt design is mechanical and generic, with the aim of enabling the verification of other code bases in the future. While this represents an initial step toward automating the verification process—especially for proof debugging—we still require human intervention beyond what is captured in the prompts.

Using the proposed LLM-assisted approach, we managed to obtain a suitable ACSL specification and a complete proof of the target parser. The resulting annotated code with all necessary tools to reproduce the proof, as well as the trace of interactions with the LLM are available in the companion artifact [10].

Finally, we assess the human effort involved in developing effective LLM prompts and interacting with the LLM during verification, and compare these aspects with manually conducted formal verification as presented in [8]. Without explicit instructions, the LLM generated a shorter and faster-to-prove formal specification for the concrete use case, but the lack of centralization and reusability—prioritized during our manual verification—can make it less efficient and harder to review for different, potentially larger message formats.

Contributions. The contributions of this work are the following:

- an experiment to evaluate the capacity of an LLM to assist security experts and verification engineers during the formal verification of critical code, specifically assessing the LLM's ability to analyze the code, generate suitable ACSL annotations, and incorporate feedback provided by the user or by the Frama-C framework;
- the development of LLM prompts and a corresponding workflow that guide the entire process of code analysis, formal specification generation, and proof debugging;
- a comparison between an LLM-generated and a manually crafted formal specification, developed for the same PKCS#1 parser implementation in C.

Outline. Section 2 recalls relevant aspects of our previous work on the formal verification of a PKCS#1 parser. Section 3 formulates the research questions and describes the applied methodology, in particular the prompting steps and the division of effort between the LLM and the human expert. Section 4 presents the evaluation results, provides details on the prompts, and discusses the threats to validity. Finally, Section 5 compares our work with related efforts and concludes.

2 Manual Formal Verification of the PKCS#1 Parser

In previous work [8], we presented the formal verification of a PKCS#1 v1.5 parser written in C. We manually developed its formal specification, along with all supplementary ACSL annotations, and formally verified it using Frama-C [13]. The full annotated code is available in the associated artifact [9]. Our main goal was to formally verify two main properties: the parser's security and compatibility. In this section, we recall the main characteristics of this previous proof.

Proof Design. Our focus was to prove the parser's resistance to entire families of Bleichenbacher signature forgery [5] and memory corruption attacks. While Bleichenbacher attacks enable the forgery of RSA digital signatures (e.g., on X.509 certificates), memory corruption attacks are an intrinsic risk for parsers implemented in C, especially when they process data potentially crafted by an attacker. The verification [8] demonstrates the ability of formal methods to provide strong *security guarantees* directly over the C implementation of security-critical software. Additionally, a *compatibility* property was proven, ensuring that the parser is not overly strict and does not reject messages with a correct format.[1]

For the proof, we used the Frama-C platform [13] with WP and RTE plugins. WP performs deductive verification based on weakest precondition calculus, while RTE generates annotations whose validity implies the absence of runtime errors.

Two aspects are essential to ensure the soundness of the verification. First, the so-called *smoke tests* [3], incorporated within the WP plugin, provide a technique to detect potential logical inconsistencies in the ACSL formal specification. While these tests do not guarantee the absence of all inconsistencies, they are highly effective at catching many of them.

Second, the RTE plugin automatically generates assert clauses that must be proved to ensure the absence of *runtime errors* (RTEs) in the C code (defined by the C standard as *undefined behaviors*). Indeed, proving the absence of such behaviors is crucial not only for ruling out memory corruption attacks (such as buffer overflows), but also for preserving the soundness of the verification. In addition, we enabled RTE options to generate assertions for detecting various *arithmetic overflows* (including those defined by the C standard as *implementation-defined* or even *well-defined* behavior). This prevents any risk of misuse of such operations. All assertions generated by RTE are subsequently discharged by the WP plugin.

Finally, to complete the proof for some complex proof goals, we used the interactive proof editor of WP and manually created proof scripts—that is, recorded sequences of applications of predefined proof tactics. Once a suitable proof script is created and applied by WP, the SMT solvers resolve the resulting proof goal(s) automatically.

PKCS#1 v1.5 Signature. The structure of PKCS#1 v1.5 [15] signature is TLV-based, meaning that all data are organized as *Tag-Length-Value* triples, where the Tag defines the kind of the TLV, while the Length field provides the size of the Value. Some TLVs are *constructed*, meaning that their Value fields recursively contain other nested TLVs. The terms *parent* and *child* are often used to describe the relation between enveloping and nested TLVs. Due to its recursive nature, effective modeling of this relation is one of the key challenges. The top level TLV structure of a PKCS#1 v1.5 signature is prefixed by a padding sequence of 0xFF bytes, whose length is determined by the RSA key size. RSA computation

[1] The definition of a correct format we used is slightly stricter than specified in [15], which allows an optional NULL field. For more detail, see [8].

is then performed over the entire encoded message, including both the padding and the TLVs, resulting in the corresponding RSA signature.

Bleichenbacher Attacks. The Bleichenbacher attack variants exploit various leniencies in the parsing of the signature format [8]. Namely, since in well-formed signature many bytes have an expected value (e.g. the padding bytes are all 0xFF, the length of a parent TLV can be derived from the length of its children, ...), some parsers forget to check those bytes, potentially leaving room for an attacker to craft an ill-formed signature, resulting into the parsing of a message that appears to have a correct hash. Conversely, if a parser checks the padding completely, the cryptographic strength of RSA is maintained and the attack is not applicable.

ACSL Formal Specification. Our formal specification exhaustively describes the message format enforced by the parser's checks in the nominal acceptance case. More precisely, is based on the fact that every byte of a compliant message is deterministically determined by the signed message, the hash function, and the RSA key size. Hence, we show that the parser accepts a message only after having checked every single byte of it against its (unique) expected value, thereby ruling out any signature forgery based on variants of Bleichenbacher attack.

To assess the soundness of our formal specification, in addition to reviewing it, we attempted to prove it on modified versions of the parser that we intentionally made vulnerable to Bleichenbacher-type attacks [9]. Each buggy variant was weakened by omitting certain strict format checks, thereby allowing arbitrary content in specific message fields. For all faulty versions, the proof failed. Since limitations of the verification tools could also lead to proof failures, we manually analyzed the situation and confirmed that the unproven ACSL annotations were *indeed* those impacted by the intended vulnerabilities.

Throughout our work, we devoted significant effort to ensuring that the formal specification is both auditable and reusable for future extensions. By using inductive predicate definitions and ghost code instrumentation, we centralized the specification in the ACSL contract of the parser's entry-point function and a set of core predicates. This approach reduces the manual auditing effort.

Moreover, within the formal specification, we separate protocol-specific aspects (such as concrete Tags present in PKCS#1 v1.5 signature) from general structural properties of the TLV-based message format (such as length consistency within nested TLV structures). This separation facilitates the reuse of the structural specification when verifying parsers of different TLV-based formats.

3 Research Questions and Evaluation Methodology

3.1 Research Questions

Our investigation is guided by the following research questions:

RQ1: What is the capacity of LLMs to summarize and explain the code?

RQ2: What is the capacity of LLMs to generate adequate formal specifications in terms of security, soundness, provability, and auditability?

RQ3: What is the capacity of LLMs to support program verification through a repair loop driven by feedback of the verification tool? More specifically, how efficient are two complementary techniques: (i) repairing ACSL annotations (such as function contracts or predicates), and (ii) generating proof-guiding ACSL assertions that help bridge proof steps that are difficult for the prover?

RQ4: How do LLMs behave when applied to a buggy version of the code? Notably, what are their capabilities in terms of bug detection, and how resilient are they to propagating bugs from the C code to the ACSL specification?

RQ5: What are the benefits and drawbacks of LLM-generated formal specifications compared to the manually developed specification previously published in [8], particularly in terms of auditability and required human effort?

3.2 Methodology: Tools

We chose to work with models GPT-5.1–GPT-5.4 in the ChatGPT user interface available through the Plus subscription tier. This choice allowed us to quickly explore and report on the capabilities of a state-of-the-art large language model. However, the lack of automation in the user interface prevented us from providing quantitative measures of prompting efficiency, which we leave for future work. For verification, we used Frama-C v29.0 (Copper) together with its WP and RTE plugins to assess intermediate versions of the annotated code and to provide feedback on properties that had not yet been proven.

3.3 Methodology: LLM-based Workflow

To build a workflow for this case study, we adopt an approach similar to that presented previously in [19], leveraging the LLM in two main modalities.

Consider the overall (simplified) workflow depicted in Fig. 1. In the first phase (Steps 1–8 in Sect. 3.3.1), given an input C code, we prompt the LLM to generate ACSL specification and supplementary ACSL annotations. In the second phase (Steps 9–11 in Sect. 3.3.2), these generated annotations are iteratively and adaptively refined based on feedback from the Frama-C verification tool, until the tool successfully confirms a complete proof. In Sect. 3.3.3 we describe the remaining responsibilities of the verification engineer within this workflow. The numbering of steps corresponds to the LLM prompts provided in the companion artifact [10].

3.3.1 Code Analysis and ACSL Generation

Code Analysis. At Step 1, we ask the LLM to analyze the target C code, in particular to identify its entry points and build the call graph, as well as to extract the informal properties ought to be enforced by parser checks for both nominal and error returns. As additional assistance to the verification engineer,

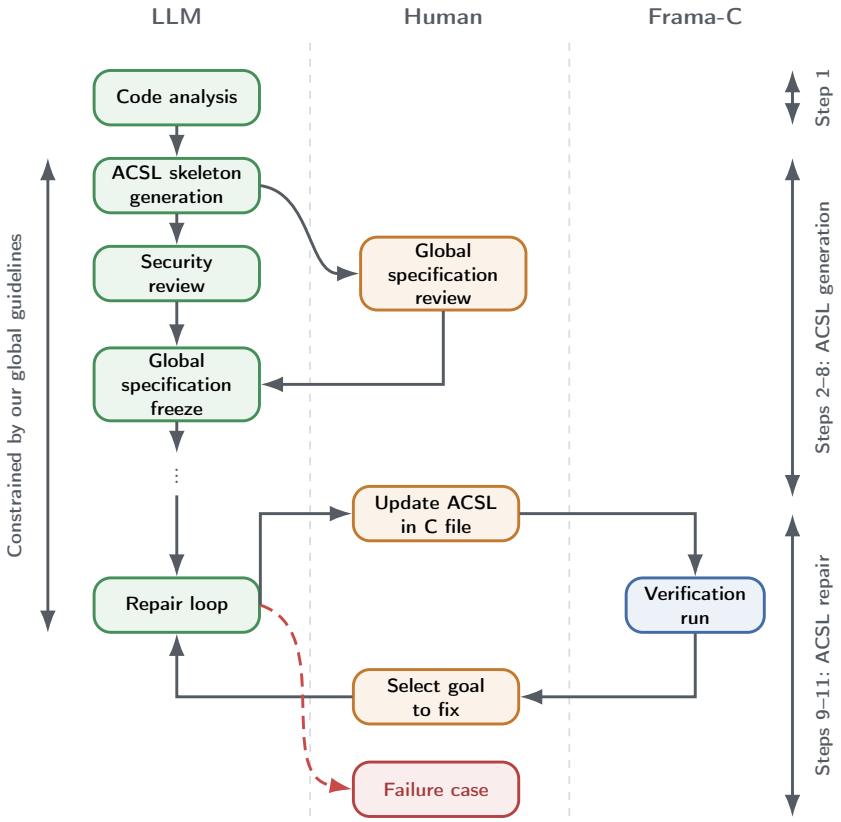

Fig. 1: LLM-assisted verification workflow.

the LLM is also asked to identify missing definitions of C language constructs (such as missing function or type definitions). This information is especially useful when extracting the verification perimeter from a real-world codebase. Retaining this analysis within the session context (automatically ensured by the user interface) is expected to enhance the quality of subsequent ACSL generation.

ACSL Skeleton. At Step 2, we prompt the LLM to formalize in ACSL the informal properties identified in Step 1. As in [2], we divide generated clauses into two categories.

First, we distinguish the *global specification*, which includes for our use case:

- the contract(s) of the entry point(s),
- the contract(s) of the *stubs*, i.e., functions that are only declared, while their definitions are provided outside the verification perimeter,
- any other specification artifacts on which they depend (such as logical functions and predicates).

Note that, in general, identifying precisely which artifacts and clauses belong to the global specification is a manual task. The remaining annotations are referred to as *supplementary annotations*, which include assertions, loop contracts, and contracts of defined callees. A careful human review of the generated global specification is required in order to check that the LLM did not introduce an inconsistency or too weak requirements. On the contrary, faulty supplementary annotations can only result in a failure to complete a proof and will not lead to deem a buggy code correct. Hence, for supplementary annotations, a review is only needed to ensure that the LLM respected the prompt's requirement (no `admit` clause, no change in the C code itself). While this was done manually in our experiment, this task mostly consists in purely syntactic checks and could thus be automated.

The need for manual review motivates the requirement for auditability of the global specification. Similarly to our previous work (see Sect. 2), we centralize the global specification within the entry point function contract, a few corresponding predicates, and function contracts for stub functions. Since the previous manual verification required the use of manually created proof scripts to help automated solvers validate some proof obligations, we did not require the LLM to reproduce the approach from Sect. 2 directly. Instead, to improve auditability and provability, we introduced a set of global guidelines that the LLM must follow whenever generating or repairing ACSL annotations. Section 4 details these guidelines.

Check of ACSL for the Security Requirements. At Step 3, we prompt the LLM to evaluate the previously generated formal specification for security. Specifically, it assesses whether the specification prevents any variants of Bleichenbacher signature forgery. This evaluation is performed by the LLM based on its training data and an internet survey. To enhance reliability, we also supply additional hints outlining criteria for the applicability of such attacks. Section 4 details these hints.

This LLM-based security check is redundant with the manual review conducted by the human verification engineer at Step 2. However, we test the LLM's capability to detect security vulnerabilities by applying Steps 2 and 3 to our buggy parsers, in which vulnerabilities are intentionally introduced, as described in Sect. 2. This checks that bugs are not transferred from the original C code into the ACSL formal specification without the LLM issuing a warning. Using LLMs to check the compatibility is left for future work.

Of course, alternative workflows involving the LLM to obtain a secure formal specification are conceivable, e.g. based on a natural language specification of PKCS#1 v1.5 and attack literature. We leave their exploration for future work. Manual review of the global specification remains necessary in any workflow.

Freeze of ACSL Global Specification. After Step 3, the global specification has been successfully audited by both the LLM and a verification engineer. Thus, Step 4 consists in instructing the LLM to freeze it. This constraint deliberately

prevents the LLM from circumventing verification issues by weakening the global specification in the next steps.

ACSL Generation Completion. At Step 5, the correctness of the ACSL syntax is checked by Frama-C, without running WP. From Step 3 until proof completion, we prompt the LLM to format its ACSL outputs as deltas relative to the previous state. The incorporation of these outputs into the full ACSL-annotated C code is currently performed manually. Developing scripts to automate this process is left for future work.

Steps 6–8 further prepare the ACSL annotations, still without verification tool feedback, thereby helping to reduce the number of iterations required during the subsequent repair phase.

As mentioned in Sect. 2, the RTE plugin automatically introduces ACSL assertions that should be proved to ensure the absence of runtime errors. At Step 6, we prompt the LLM to add supporting ACSL annotations to facilitate the proof of these assertions. These include, for example, function contracts that propagate additional information or, in the case of preconditions of the entry-point function, introduce new modeling hypotheses.

At Step 7, we prompt the LLM to double-check a list of common issues frequently observed in generated ACSL annotations. For example, we ask the LLM to complete pointer separation assumptions (i.e., the absence of aliasing).

Step 8 prepares the LLM to handle smoke test failures. Since smoke testing may be less represented in training data, we use the few-shot prompting technique and provide the LLM with more information and examples, all drawn from the dedicated paper [3]. Similarly, at Step 6, we supply an extract from the RTE manual [7] about the handling of targeted arithmetic overflows (cf. Sect. 2).

3.3.2 ACSL Repair

After the ACSL annotation generation phase, we enter the repair loop. This phase incorporates verification tool feedback to iteratively improve the annotations, aiming at completing formal verification.

Within the iterative Step 9, the verification engineer runs Frama-C using the WP and RTE plugins, analyzes the results, and selects one or more properties that remain unproven. Currently, this selection is the primary manual effort in the process. However, since no fundamental obstacles are anticipated, automating this task with the LLM is planned for future work. If the proof is still incomplete, the information identifying the unproven property is inserted into a pre-prepared prompt skeleton and sent back to the LLM.

The prompt instructs the LLM to choose one of three possible actions:

(i) propose a fix to ACSL annotations,
(ii) initiate a debugging session using intermediate ACSL assertions,
(iii) raise an intervention alert.

Proposing updated ACSL annotations is the preferred action. If no suitable modification is identified to unlock the unproven property, the LLM proceeds to

debugging by introducing a sequence of ACSL assertions. Each inserted assertion must be fulfilled at a specific program point and serves as an intermediate lemma for subsequent assertions. This sequence of ACSL assertions effectively decomposes a complex property into smaller achievable steps. Finally, if the LLM detects an error case as defined by the prompt, an alert is raised requesting human intervention; these cases will be discussed in the following Sect. 3.3.3.

Once debugging with a sequence of assertions successfully proves the property, a final cleanup may be performed by removing irrelevant assertions at Step 10. This is a speculative step, prompted as a one-shot attempt. If the successful proof is lost, the changes are reverted at Step 11.

3.3.3 Human Effort and Interventions

Our objective is to provide *strong guarantees based on a mathematically rigorous verification tool*, rather than on the judgment of the LLM. In the nominal workflow—when no error cases are detected—the following tasks are assigned to the verification engineer:

- to review the final formal specification for soundness and security aspects;
- to integrate updates generated by the LLM into the ACSL-annotated C code and to run the Frama-C tool;
- to decide how to filter verification feedback before sending it back to the LLM, in particular determining which not-yet-proven properties to report.

In addition, we have defined several error cases that are automatically detected by the LLM, in which it also requests human intervention:

- The generated global specification does not enforce the strong security property required by the provided hint. As part of the final formal specification validation at Step 3, the engineer determines whether it is meaningful to proceed with the next verification steps.
- Modification of the global specification (including model hypotheses) after it has been reviewed by the engineer. The LLM is explicitly prompted at Step 4 to avoid such modifications, otherwise they must be reviewed again.
- The LLM concludes (at Step 9) that the code is not provable under the agreed (and frozen) ACSL formal specification (and might contain a bug).
- The LLM concludes (at Step 9) that there is dead code present, either intrinsically in the C code or caused by the agreed ACSL formal specifications (including model hypotheses)[2]. Some smoke test failures may be expected (due to intentionally introduced dead code, for example, sanity checks or protections against fault injection attacks).
- The LLM concludes (at Step 9) that certain proof steps are too complex for the verification tool and stops proof debugging using assertions. Complex properties or suboptimal choices during ACSL annotation generation can

[2]Imposing more constrained preconditions can indeed reduce the input domain and consequently make some program points unreachable.

So the strongest informal statement is: On nominal return, the parser has established that the provided block is a well-formed PKCS#1 v1.5 encoded digest container for the caller-expected hash algorithm, and that its embedded digest matches the recomputed digest of the supplied input message.

Fig. 2: Informal nominal parsing property summarized by the LLM.

indeed face the limits of the provers. The verification engineer may respond by optimizing the generated ACSL annotations, employing other Frama-C features (such as proof scripts), or using interactive provers like Rocq.

4 Evaluation Results

We answer our research questions (cf. Sect. 3.1) based on observations during the LLM-assisted formal verification of the PKCS#1 v1.5 parser. We illustrate our findings with concrete examples for readers familiar with ACSL and the PKCS#1 v1.5 parser; other readers can skip them. Given the stochastic nature of ChatGPT, we repeated the verification process four times. In each case, we were able to complete the proof, thereby ensuring security against both Bleichenbacher signature forgery and memory corruption attacks.

Our prompts were gradually refined to make the verification process more mechanical. The artifact [10] provides their final versions. All extracts of communication with the LLM presented in this paper are from the final reproduction. Prompts appear on a white background, and LLM responses on a light blue background. Minor reformatting and corrections of typographical errors have been applied in the paper (but not in [10], to prioritize reproducibility).

4.1 RQ1. Code Analysis

During our experiments, the LLM generally demonstrated *a strong capability to analyze C code details.* In particular, it was able to correctly and comprehensively reconstruct the message format enforced by the parser, accurately recognizing all format checks, including those applied for the TLV structure nesting. Figure 2 shows the parser's global properties upon nominal return, expressed by the LLM.

In another interesting example, the LLM was also able to detect intentional dead code in the implementation—identified after a smoke test alarm—which originated from the ACSL contract of a stub function, also written by the LLM.

4.2 RQ2. ACSL Generation

We evaluate the quality of LLM-generated ACSL according to three important aspects: auditability, provability and soundness. Soundness concerns corresponding to the most challenging security-related scenario are also discussed in Sect.4.4. We observed that *the quality of all aspects improved significantly when we provided the LLM with global guidelines* for all subsequent ACSL generations or repairs. Figure 3 shows an excerpt of these guidelines.

...
3. Use following predicate pattern for each TLV, which has some children (in DER terminology, Constructed TLV). You CAN choose different layout if you need to relate content of two or several different parent TLVs. Otherwise follow this pattern whenever possible.

```
predicate CTLV_TLVname(integer size, uchar * p, ...) =
// structure and tags
(let t1_off == 0;
p[t1_off] == TAG1 &&
(\let t2_off = t1_off + p[t1_off+1] + LEN_TAGLEN;
p[t2_off] == TAG2; &&
(\let t3_off = t2_off + p[t2_off+1] + LEN_TAGLEN;
p[t3_off] == TAG3; &&
(\let t4_off = t3_off + p[t3_off+1] + LEN_TAGLEN;
// TLV_name size consistency
size == t4_off)))) &&
// recursive application of this predicate pattern if the child
is also a parent/Constructed TLV
// predicate or direct expression of semantic properties
if the child is NOT a parent/Constructed TLV
...
```

4. For ACSL contract of a stub function representing data transformation (e.g. crypto), use a logic function with byte return value. You may use axioms to specify it, but do prefer to avoid it.
5. Except point 4, don't use logic functions and axioms.
6. Don't use lemmas, unless there is no alternative how to proceed.
7. All hypotheses should be included into the model only EITHER as (a) an entry point precondition OR (b) a stub function postcondition.
For both cases, be careful not to include logical contradictions and inconsistencies! For both cases, all introduced hypotheses must pass the check of being realistic, i.e. check if it can be PROBABLY fulfilled by the outside world (entry point's caller and stub functions' callees). If you modify some of the existing hypotheses, raise a warning to me using the message "MODIFY MODEL HYPOTHESES - MANUAL REVIEW MANDATORY".
...
9. Don't use ACSL admit clause.

Fig. 3: Global guidelines constraining the generation of ACSL by the LLM.

Auditability. As explained in Sect. 3, a human review remains mandatory after ACSL formal specification has been generated. Guideline 3 in Fig. 3 prescribes *a unified form for predicates,* especially regarding TLV nesting, which reduces the likelihood of mistakes during the review process. The LLM *correctly followed the pattern* for generating predicates corresponding to particular nested TLVs.

To optimize auditability and soundness, Guidelines 4, 5, 7, and 9 constrain the use of ACSL constructs that introduce new model hypotheses. In particular, we strongly limit the use of axioms (i.e., the introduction of global hypotheses) and prohibit the use of the ACSL admit clause (i.e., the local hypothesis introduction). The prescribed locations for introducing hypotheses are the parser's entry point preconditions and the postconditions of stub functions, as dictated

I kept the model hypotheses only in the two allowed places from guideline 7:

- entry-point preconditions:
  - supported hash selector
  - readable input buffers
- hash-stub postconditions:
  - exact output length
  - exact bytewise content through one logic byte-valued function

I did not add axioms or lemmas.
...
One deliberate modeling choice is the entry-point precondition `SupportedHashAlg(expect_hash_ind)`. The C code indexes arrays with `expect_hash_ind` and has no defensive bounds check, so this assumption is necessary both for soundness and for faithful deductive verification. This is a realistic caller obligation and therefore compatible with guideline 7.

Fig. 4: Hypotheses introduction.

by Guideline 7. Figure 4 shows the sound hypotheses choices made by the LLM, illustrating its correct code analysis and respect to the guidelines.

Provability. To optimize provability, we instructed the LLM to avoid ACSL constructs that are often difficult for SMT solvers, such as global lemmas and excessive use of logic functions. Guideline 4 prescribes the use of logic functions for one specific purpose, which is discussed in Sect.4.5..

The LLM was also *effective to generate additional ACSL annotations to prove the absence of RTEs.* Since the RTE plugin automatically generates the necessary ACSL assertions, the LLM typically introduced supplementary ACSL clauses acting as function preconditions, as illustrated in Fig. 5. Similarly, we obtained complete and correct analysis when the LLM was asked to provide hypotheses for C pointer aliasing.

To further improve provability, as part of Step 7 described in Sect. 3, we included a hint that effectively helped the LLM avoid one of its recurring mistakes: specifying memory validity of **`const`** C arrays using the **`\valid`** ACSL clause. Since this clause assumes both reading and writing access, it introduces *a logical contradiction* as **`const`** arrays should not be writable. Such mistakes were effectively detected by WP smoke tests.

Soundness. In the last reproduction, the formal specification was generated with a mistake: it incorrectly specified the boundary between the TLV structure and its prefix (containing `0xFF` bytes). Our implementation does not contain such a bug. We observed that this incorrect specification was not generated systematically—it occurred only once and did not reappear in subsequent reproductions, even when re-using the same session context via the ChatGPT session

In `read_one_tlv`, the expression
C: `⋆num_read + LEN_TAGLEN`
is evaluated before the comparison. RTE will ask you to prove that this addition cannot overflow signed int. The minimal way to guarantee that is:
acsl: `requires ⋆num_read ≤INT_MAX - LEN_TAGLEN;`
This is realistic and belongs exactly where your guideline 7 says hypotheses should go: in a function precondition.

Fig. 5: Assumptions to prove absence of RTEs.

`OFF_B2 + 1 + MIN_FF_COUNT + 1 + LEN_TAGLEN ≤dig_tlv_off ≤size - LEN_TAGLEN ∧`
Please correct this line.

Thought for a second
Use this corrected ACSL line:
`OFF_B2 + 1 + MIN_FF_COUNT + 1 ≤dig_tlv_off ≤size - LEN_TAGLEN ∧`
Why:
- `dig_tlv_off` is the offset of the outer `TAG_SEQ` .

...
- So the minimal possible offset of `TAG_SEQ` is:
  - `OFF_B2 + 1` = first byte after `EXP_B2`
  - `+ MIN_FF_COUNT` = minimum FF padding
  - `+ 1` = delimiter
  - `LEN_TAGLEN` must not be added on the left-hand side, because it belongs to the bytes starting at `dig_tlv_off`, not before it.

Fig. 6: Indicated mistake and generated fix in the formal specification.

branch feature[3]. Notably, *the LLM corrected itself when this issue was pointed out,* as shown in Fig. 6. This finding confirms *the importance of human review of formal specifications* in our use case (at Step 2, cf. Sect. 3.3.1).

4.3 RQ3. ACSL Repair Phase

During all reproductions, the LLM exhibited *strong skills in both reasoning and fixing annotations based on Frama-C feedback.* By incrementally extending the global guidelines to avoid common mistakes in ACSL annotations in new reproductions, we managed to reduce the number of mistakes that needed to be addressed in the repair loop. We believe that the guidelines are general enough to have a similar effect when reused for other code bases.

Figure 7 shows an example of a repair loop step: an ACSL contract correction for the supplementary C function `read_one_tlv` that implements TLV

[3]See https://help.openai.com/en/articles/6825453-chatgpt-release-notes

The failing VC says that, at the two call sites inside `pars_PKCS1_lev2`, WP cannot prove the current precondition:
`requires` `0 ≤⋆num_read ≤num_to_read;`
But this precondition is actually stronger than what the C code needs. Observation on the C code `read_one_tlv` starts with:
`if` `(⋆num_read + LEN_TAGLEN >num_to_read){`**`return`** `ERR_LEN;}`
So if `⋆num_read >num_to_read`, the function is still perfectly safe:
• it does not dereference `msg[⋆num_read]`,
• it immediately returns `ERR_LEN`.
Therefore, requiring `⋆num_read ≤num_to_read` at function entry is unnecessarily strong and blocks modular proof at callers that intentionally rely on the function to detect "already out of bounds" situations.
...
Minimal fix:
We should weaken the precondition of `read_one_tlv` to what is actually needed for safety and RTE:
• keep `0 ≤⋆num_read`
• keep `⋆num_read ≤INT_MAX - LEN_TAGLEN` to avoid overflow in `⋆num_read + LEN_TAGLEN`
• require msg validity only in the branch where the header is actually read

Fig. 7: Fix of the ACSL contract for a supplementary C function.

fetching. Interestingly, the reasoning is correct, except for one subtle point: although the proposed fix does weaken the contract as required—minimizing the properties to be proven by the function's callers—it is not minimal in its size, as the last memory validity weakening is not strictly necessary.

Similarly, during ACSL assertions generation for a proof debugging session at Step 9, we observed *a systematic approach of the LLM with a correct reasoning,* decomposing complex logical formula into simpler ones, complex arithmetic formula into particular arithmetic operations, and distributing assertions into nominal and erroneous branches. We illustrate that in Fig. 8, where the generated assertion sequence helps the prover to enumerate the array `OID_list` of the supported hash OIDs, nicely bridging the gap between the (failed) code check, repeated by the first assertion and the (negated) predicate `OIDValueMatches` (which uses the enumeration results, such as `SHA256_OID`).

Note that for all our proof reproductions, we manually intervened with small hints on top of the pre-prepared prompts to further help the LLM. For example, in an earlier reproduction, before constraining the usage of axioms as explained in the previous section, we assisted the LLM with an axiom instantiation and modified its choice of ACSL logic functions. We also make it swap some lines in LLM-generated output to facilitate correct C file update and removed some suboptimal choices in generated assertions to avoid Frama-C/WP limitations.

```
1 /*@ assert oid_len != OID_list[expect_hash_ind][0] ||
2     (\exists integer i; 0 <= i && i < oid_len &&
3      oid_start[oid_val_off + i] != OID_list[expect_hash_ind][1 + i]); */
4 /*@ assert expect_hash_ind == SHA256_INDEX ==>
5     (oid_len != SHA256_OID[0] || (\exists integer i; 0 <= i && i < oid_len
          && oid_start[oid_val_off + i] != SHA256_OID[1 + i])); */
6  ... the analogical asserts are introduced here for other hashes ...
7 /*@ assert !OIDValueMatches(expect_hash_ind, oid_start + oid_val_off,
      oid_len); */
```

Fig. 8: The enumeration using ACSL assertions (extract).

Hint 1: The Bleichenbacher attack is applicable if the parser accepts a sufficient amount of sufficiently unconstrained bytes in any part of the parsed message.
Hint 2: The ACSL annotations should enforce that the parser accepts only up to 2 different PKCS#1v1.5 payload (parsed message) representations for the particular message-to-hash, hash function and the payload (i.e. RSA) size.

Fig. 9: The security hints provided to the LLM at Step 3. (Two representations are accepted since the specification [15] allows an optional NULL field.)

4.4 RQ4. Bug Detection

As described in Sect. 3, after the ACSL annotation generation step using the provided C code, we tasked the LLM with cross-checking the security of the annotations against Bleichenbacher attack.

We evaluated the LLM's behavior while applying Steps 1–3 on the first four vulnerable parsers from [9]—buggy parser variants deliberately modified to contain vulnerabilities enabling Bleichenbacher attack. For each variant, we conducted two repetitions. Figure 9 displays the additional hints provided, which summarize criteria that LLM should use to strengthen the security analysis[4].

We observed both possible behaviors during the ACSL generation: more frequently the bug was transcribed into the ACSL annotations, but in some cases, the LLM did not follow the vulnerable code. Instead, it generated and justified ACSL annotations based on its intent, e.g., deduced from the LLM's knowledge of the PKCS#1 standard.

We found that *the LLM was not entirely reliable in subsequent security analysis.* While most attempts were correct, in one instance the LLM provided a wrong conclusion: it reproduced the vulnerability (skipping a check of NULL TLV length to be zero), and subsequently did not identify the vulnerable ACSL specification. For this check, the LLM silently assumed PKCS#1-compliant message format (specifying zero length) instead of reflecting the actual property expressed by ACSL. Notably, because the error originated from this assumption, the provided hints alone were ineffective in guiding the LLM to correct

[4]Not fulfilling the second hint's property does not necessarily imply that the attack is applicable, but it detects all vulnerable parsers while raising few false alarms.

the mistake. It demonstrates that the human review—included in Step 2 of our methodology—remains necessary to decrease the probability of an unsound global specification.

4.5 RQ5. LLM-assisted Proof vs. Manual Proof

While both the manual and LLM-assisted proofs achieve the same security goals, differences in proof design impacted various proof qualities. We provide a comprehensive comparison of both approaches.

Auditability. As described in Sect. 2, the use of generic inductive predicate definitions parameterized with ghost code instrumentation enabled *a high level of centralization for the global specification and promoted its reusability* across various TLV-based message formats. However, because the corresponding proof required manual proof scripts, we did not attempt to reproduce this approach with the LLM.

Instead, we instructed the LLM to follow a unified form of expressions for describing TLV structures. Nevertheless, although it follows the same pattern, each instance of TLV nesting is handled by a separate predicate, which provides *reduced auditability compared to the more centralized manual approach* and increases the risk of errors by the LLM.

On the other hand, *without any specific prompting, the LLM also improved the auditability of some parts of the specification* in ways not used by the manual solution. Specifically, it introduced ACSL logic functions to model stub C functions that perform cryptographic hashing operations. This approach enables the specification of the hashing operation to be propagated into the ACSL contract of the entry point function. In contrast, the manual approach relied on locally inserted ghost code and preconditions of stub functions. Recognizing this advantage, we adopted the LLM-generated solution in subsequent reproductions and prescribed it as Guideline 4.

The LLM generated fewer lines of ACSL annotations—226 compared to 364 required for the manual approach—thereby reducing the spec-to-code ratio from 2.35 to 1.35[5]. The count of global specification—i.e., ACSL annotations requiring a manual review as explained in Sect. 3—was also lower for the LLM approach (99 lines) than for the manual one (227 lines). The larger amount of ACSL annotations in the manual approach is due to instrumentation introduced for the sake of centralization and reusability. When this design is reused for a different TLV-based message format with a greater number of parsed TLVs, we expect it to demonstrate its advantages and to reverse this ratio.

Provability. The absence of inductive predicates in the LLM-assisted approach allowed us to complete the proof without relying on a manually created proof script. Difficult proof steps were managed by introducing debugging ACSL assertions.

[5]In both cases, we used the `cloc` utility, which counts ACSL annotations as other C comments. To isolate ACSL, all standard C comments were removed. Empty lines in C code are ignored.

Soundness. Both approaches generated similar proof hypotheses, introduced as entry point preconditions. In the manual approach, we assumed several additional pointer separation conditions to reflect the parser's intended behavior, although these were not strictly necessary for a successful and sound proof. In both cases, the hypotheses introduced are realistically satisfiable by a potential external caller. The RTE plugin generated 46 assertions necessary for proving absence of RTEs. As expected, the same number was generated for the input C code also during the manual proof[6].

Proof Results. During this work, both the manual and LLM-generated proofs were executed using Frama-C v29.0 (Copper) with the external SMT solver Alt-Ergo 2.5.4, running on a VirtualBox VM with Ubuntu 24.04. The host machine used Windows 11 and provided an Intel(R) Core(TM) i7 CPU @ 2.70GHz with 4 processors and 8 GB dedicated to the VM.

The WP plugin generated and successfully proved 393 goals, which is approximately 9% fewer than for the manual proof. This included 4 reachability goals and 61 smoke tests. The internal WP simplifier, Qed, proved 266 goals, leaving the remaining 123 goals to be discharged by the Alt-Ergo prover.

We observed *a significantly faster proof for the LLM-generated approach*, totaling 1min16s (with up to 8.6s for a single proof goal), compared to 2min57s (up to 17.6s for one proof goal) observed for the manual proof. As for the larger amount of ACSL annotations, we believe that this is the price to pay for higher centralization and reusability of the specification.

Human Effort in the Proof Preparation. We estimate the total human effort for this work to be approximately 2.5 person-months, compared to 6 person-months required for the manual proof. In both verification processes, we prioritized reusability over rapid completion. The manual proof estimate includes time spent studying related attacks and developing a new verification problem methodology—knowledge that subsequently accelerated the LLM-assisted approach. Conversely, a notable portion of the time devoted to the LLM-assisted verification was spent becoming familiar with LLM concepts, behaviors, and effective prompting strategies. Overall, we estimate that the LLM-assisted approach significantly accelerates the formal verification process for this case study.

4.6 Threats to Validity

Internal Validity. If the LLM was trained directly on the manually verified parser in our previous work [8], our experiment would produce non-representative results. To reduce this threat, we verified that the training cut-off dates[7] for all models we used occurred before the publication of our previous work. We also used the ChatGPT settings to block continuous learning from our initial debugging attempts[8]. Yet, we cannot entirely rule out that the learning could

[6] In the manual approach 6 additional assertions were introduced inside the ghost code it uses.

[7] https://developers.openai.com/api/docs/models

[8] https://help.openai.com/en/articles/7730893-data-controls-faq

be triggered by other users referencing our paper. Notably, the ChatGPT model we used has also access to the internet. However, the validity of our experiment is supported by the fact that the generated ACSL annotations featured a completely different design than in our previous work, even across repeated reproductions.

Furthermore, due to their stochastic nature, LLM outputs vary across individual reproductions and may not reliably repeat a successful verification process. Because of the user interface used, we were limited in our ability to provide strong quantitative measures to increase the assurance. However, to improve representativeness, we repeated the full formal verification process four times. Across these reproductions, we observed substantial overlap in the generated specification patterns. Although manual intervention was still required for various details, the LLMs were able to follow our correction requests appropriately.

We entered this study with prior knowledge of the PKCS#1 parser, its buggy versions and its formal verification. This significantly reduced the effort required to evaluate the LLM outputs and identify blocking points. On the other hand, we deliberately did not push the LLM to reproduce our previous solution; instead, we constrained it gradually based on its recurring mistakes and general good-practice guidelines. Moreover, when testing the LLM's ability to detect bugs deliberately incorporated into different parser variants, we followed the same procedure as for the correct variants. As reported above, the LLMs were not fully reliable in detecting the bugs, and our methodology relies on concurrent manual review. In addition, we manually identified an ACSL bug that the LLM introduced when annotating the correct C code. This demonstrates that the methodology can reveal bugs that are not directly anticipated.

External Validity. We used ChatGPT and the Frama-C verification framework. The results may differ when other tools are used. Throughout our work, we employed several GPT versions 5.1–5.4, which produced similar results. Evaluating other major LLMs is left for future work.

While we believe that our methodology provides a solid basis for similar use cases, additional manual guidance is likely to be required when adapting it to formally verify other code bases, especially if they are not well-structured and well-commented, or if they implement parsers for different TLV-based formats.

5 Related Work and Conclusion

Related Work. The potential of LLMs to assist formal verification inspired researchers to investigate their application within various verification workflows. Some studies have focused on completing proofs where requirements are already formalized as global specifications, introducing supplementary specifications such as loop invariants, debugging assertions, or contracts of auxiliary functions. These approaches have been applied across different programming and specification languages and their verifiers, for example, for C, ACSL, Frama-C/WP and CPAchecker [18] [11] [12], for Rust and Verus [19], and for Dafny [16]. In these works, feedback from the verification tool is used as an effective method to refine the generated formal specifications. In another step, global specifications

can be automatically generated, leveraging a variety of resources. For example, in [1] natural language specifications, Java code and feedback from the verification tool KeY were used. A recent work [6] investigated specifications generation using the outputs of analysis tools (PathCrawler, Frama-C/Eva) alongside the C code. In addition, it evaluates the ability of a LLM to generate the global specification based on the program's *intent*, inferred solely from the code—despite the presence of intentionally introduced coding bugs. By contrast, in our approach, we provide the intent as the explicit security requirement within the prompts.

Some related works have utilized LLM assistance for the formal verification of security problems. In [14] formal specifications for smart contracts were generated, which automate the execution of—often high-value—digital agreements among users on blockchain-based platforms such as Ethereum. In [17], the authors defined a domain-specific language (DSL) to specify message formats, generated and formally verified their parsers in F*, and subsequently extracted them into C code. While this work also addressed the PKCS#1 use case, their approach does not target a security proof directly on the existing C code. Due to the labor-intensive manual crafting of DSL formal specifications, LLMs have been deployed to automate their generation [4] and compared with the human-crafted specifications. Interestingly, mistakes were detected in both manual and LLM-generated formal specifications, highlighting the value of a combined approach. To the best of our knowledge, an LLM-assisted formal verification of a PKCS#1 parser in C has never been evaluated before.

Conclusion and Future Work. Our experiment demonstrated that, while human oversight remains necessary, state-of-the-art LLMs can be highly effective in supporting the formal verification of security-critical C programs using the ACSL specification language. We believe that LLM-assisted formal verification has the potential to significantly enhance both offensive and defensive security analysis. In this work, we presented LLM-assisted formal verification of a concrete security problem—PKCS#1 v1.5 signature forgery using Bleichenbacher attack—with promising results. We proposed a generic methodology, facilitating its reusability in both academic and industrial contexts for other studies.

For future work, the effectiveness of various combinations of global specification generation and human cross-checking can be investigated, incorporating natural language descriptions or previously audited formal specifications as resources. We expect that further automation will be possible, improving communication between verification tools and the LLM. Finally, we plan to extend our verification approach to parsers of other message formats, such as X.509, whose complexity was one of the motivations for this study.

Acknowledgment. Part of this work was 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 project VASSAL (Grant 101160022).

References

1. 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
2. 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
3. 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
4. Fakhoury, S., Kuppe, M., Lahiri, S.K., Ramananandro, T., Swamy, N.: 3DGen: AI-assisted generation of provably correct binary format parsers. In: Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025). pp. 2535 – 2547. IEEE (2025). https://doi.org/10.1109/ICSE55347.2025.00173
5. Finney, H.: Bleichenbacher's RSA signature forgery based on implementation error (2006), https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/
6. Granberry, G., Ahrendt, W., Johansson, M.: Seeking specifications: The case for neuro-symbolic specification synthesis. Formal Aspects of Computing **38**(2), 1–36 (2026). https://doi.org/10.1145/3785411
7. Herrmann, P., Signoles, J.: RTE Runtime Error Annotation Generation (2024), https://frama-c.com/download/frama-c-rte-manual.pdf
8. Hána, M., Kosmatov, N., Prevosto, V., Signoles, J.: Formal verification of PKCS#1 signature parser using Frama-C. In: Proc. of the 20th International Conference on integrated Formal Methods (iFM 2025). LNCS, vol. 16194, pp. 336–358. Springer (2025). https://doi.org/10.1007/978-3-032-10794-7_17
9. 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
10. Hána, M., Kosmatov, N., Prevosto, V., Signoles, J.: From manual to LLM-assisted formal verification of a PKCS#1 signature parser. Companion artifact. (May 2026). https://doi.org/10.5281/zenodo.20056822
11. 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
12. 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

13. 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
14. Liu, Y., Xue, Y., Wu, D., Sun, Y., Li, Y., Shi, M., Liu, Y.: PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation. In: 32nd Annual Network and Distributed System Security Symposium (NDSS 2025) (2025). https://doi.org/10.14722/ndss.2025.241357
15. 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
16. Mugnier, E., Gonzalez, E.A., Polikarpova, N., Jhala, R., Yuanyuan, Z.: Laurel: Unblocking automated verification with Large Language Models. In: Proceedings of the ACM on Programming Languages, Volume 9, Issue OOPSLA1, pp. 1519–1545. ACM (2025). https://doi.org/https://doi.org/10.1145/3720499
17. 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
18. 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
19. Yang, C., Li, X., Misu, M.R.H., Yao, J., Cui, W., Gong, Y., Hawblitzel, C., Lahiri, S., Lorch, J.R., Lu, S., Yang, F., Zhou, Z., Lu, S.: AutoVerus: Automated proof generation for Rust code. In: Proceedings of the ACM on Programming Languages, Volume 9, Issue OOPSLA2, pp. 3454–3482. ACM (2025). https://doi.org/10.1145/3763174