AI-based KCMVP Pre-certification System: A Hybrid Model of Rule-based Detection and LLM Semantic Analysis

Su-Been Cho[1], Do-Yun Park[1], Da-Eun Lim[1], Jae-Hwan Kim[1], Su-Min Jeong[1], Yu-Lim Hyoung[1], and Hwa-Jeong Seo[1⋆]

Department of Convergence Security,
Hansung University, Seoul (02876), South Korea
{chosubin1208,rhcp030418,limadaeun,jaedol2023,jeong9sumin,yulim4hyoung}@gmail.com

Abstract. The Korean Cryptographic Module Validation Program (KCMVP) is a national certification system that verifies the security and conformity of cryptographic modules deployed in government and public institutions. The current process typically takes about one and a half years, during which frequent supplement requests and the resulting retesting cycles substantially raise costs and delay schedules. To address this, we propose an AI-based pre-certification framework that combines rule-based deterministic detection (L1), RAG-based guideline-evidence retrieval (L2), and LLM-based final decision (L3) into a funnel-shaped pipeline that progressively reduces false positives. L1 applies more than 170 YAML inspection rules across four pattern types (missing, regex, semantic, ast) to perform deterministic detection. L2 retrieves and attaches KCMVP guideline evidence to each violation through multi-stage RAG search, and L3 employs Gemini 2.5 Flash-Lite to make context-aware decisions on false-positive candidates. In an initial evaluation on 128 Ground Truth cases derived from the KISA LEA code, the system detected all 128 cases, achieving 100% recall, while L3 correctly removed 9 of 46 FP candidates (19.6%) without inducing any false negatives (FN), confirming the stepwise refinement effect of the funnel structure. A blind verification on a certified commercial cryptographic module (∼14.5 KLOC) yielded a low detection frequency of 0.58 cases per 1,000 lines of code, supporting the system's practicality in real environments.

Keywords: KCMVP · AI · LLM · Pre-certification · RAG · Rule-based Detection · AST · Multi-layer Analysis · Semantic-based Verification · Pre-conformance Inspection

1 Introduction

The Korean Cryptographic Module Validation Program (KCMVP [20]), administered by the National Intelligence Service (NIS) and the Korea Internet &

⋆ Corresponding author: hwajeong84@gmail.com.

Security Agency (KISA), is a mandatory statutory certification system for cryptographic modules deployed in the information systems of government and public institutions. Under the Electronic Government Act, all cryptographic modules operated on the national network of administrative agencies and the like must pass KCMVP validation, and this functions as a core pillar that supports the Republic of Korea's national cybersecurity infrastructure.

The KCMVP validation procedure broadly consists of four stages: validation application, document review, technical review (main examination), and final deliberation/certification [20].

In particular, at the technical review stage, the validator analyzes the source code and rigorously inspects the correctness of the cryptographic algorithm implementation, compliance with secure coding practices, and the consistency between the source code and the submitted documents.

The main bottleneck in the current validation process occurs at the Supplement Requests stage. When a defect is found during the validation process, the applicant organization must correct the issue, re-perform the relevant tests, and resubmit. Each time this supplement cycle is repeated, additional delays of several weeks to several months arise, and when remediation work is performed through external consulting, a substantial cost burden is incurred. Despite such clear limitations, no automated pre-certification tool currently exists that would allow developers to autonomously inspect their source code and documents against KCMVP requirements before official submission.

To fill this gap, in this study we propose a hybrid pre-certification tool – combining rule-based static inspection with Retrieval-Augmented Generation (RAG) and Large Language Models – that automatically scans the source code and submitted documents for potential KCMVP violations before the official review begins, thereby reducing the frequency and severity of supplement requests. Note, however, that the system proposed in this paper is a pre-certification auxiliary tool and a prototype, and we explicitly emphasize that it does not replace the official KCMVP validation process of national agencies.

The prototype is available on GitHub:
`https://github.com/SuBeen-Cho/KCMVP-Compliance-Tool.git`

1.1 Main Contributions

Proposal of a hybrid three-stage validation pipeline. We propose a funnel-shaped three-stage architecture that combines rule-based static detection (L1), guideline evidence retrieval and augmentation (L2), and generative-model-based final decision (L3), thereby improving the balance among detection, explanation, and precision. On 128 Ground Truth cases the pipeline achieved 100% recall, and L3 re-evaluation removed 9 of the 46 FP candidates generated by L1 (19.6%) without inducing any false negatives, yielding a final precision of 77.6% and an F1-score of 87.4%.

Systematized KCMVP inspection rule set and domain-specific false-positive mitigation. We subdivide inspection rules derived from actual KCMVP documents into more than 170 rules across five categories (common security,

algorithms, modes of operation, documentation, and traceability) and encode them with four pattern types (missing, regex, semantic, ast) in YAML, enabling systematic management and expansion of a large-scale rule set. A blind verification on a certified commercial module (14.5 KLOC) recorded a low FP rate of 0.58 cases/KLOC, confirming practical applicability.

Simultaneous validation of source code and submitted documents. By taking as input and preprocessing not only the source code but also the submitted documents (design, configuration management, testing, etc.) that are mandatorily required during the cryptographic module validation process, this tool enables code rules and document rules to be inspected in parallel within the same pipeline. Through this, the system goes beyond static analysis of code alone and also encompasses rule-based inspection of the document deliverables required by the validation authority.

2 Background

2.1 Overview of the KCMVP System

The Korean Cryptographic Module Validation Program (KCMVP) is a national certification system operated under Article 9, Paragraphs 2 and 3 of the Cyber Security Work Regulations [24] and Article 69 of the Enforcement Decree of the Electronic Government Act [25]. It is the domestic counterpart to the United States' CMVP (FIPS 140-2/3) [17,18], with a similar validation framework and security-requirements structure but differing in the list of approved algorithms and the testing procedures. The purpose of this program is to objectively verify whether the cryptographic modules used in the information systems of government and public institutions satisfy national security requirements. The targets of KCMVP validation are cryptographic modules intended to protect non-classified business data, and they may be implemented as software, hardware, firmware, or a combination thereof.

KCMVP is based on KS X ISO/IEC 19790 (Security Requirements for Cryptographic Modules) [9] and KS X ISO/IEC 24759 (Test Requirements for Cryptographic Modules) [10], and classifies security levels into four grades, from Level 1 to Level 4. This study targets Level 1 software modules, which are the most common subjects of domestic cryptographic module certification.

The core functions that a cryptographic module must provide are the four security services of confidentiality, integrity, authentication, and non-repudiation, and the KCMVP-approved algorithms include LEA [6], the Korean national standard, as well as international standard algorithms such as AES [19] and SEED [8]. Among these, LEA is the only lightweight block cipher developed domestically, and it is the most frequently used algorithm in KCMVP software implementations. Therefore, this study selects LEA as the main analysis target.

2.2 KCMVP Adoption Criteria and Validation Procedure

Products for which KCMVP certification is mandatorily required fall broadly into three types: (i) information protection systems (firewalls, VPNs, intrusion

detection systems, etc.), (ii) quantum cryptographic communication equipment, and (iii) products in which cryptography is the main function. In this respect, unlike hardware and firmware, software modules undergo frequent code modifications, and depending on the scope of change when applying functional improvements or security patches, either full re-validation or partial re-examination is required. This produces recurring temporal and financial burdens even after certification has been obtained.

The KCMVP validation procedure proceeds in the following four stages:

1. **Validation Application:** The cryptographic module developer organization applies to the testing agency (KISA) for validation, submitting the source code, detailed design specification, test specification, configuration management document, and operator/user manual.
2. **Document Review:** The testing agency verifies the formal completeness of the submitted documents and whether they satisfy basic requirements.
3. **Technical Review** (Main Examination): The testing agency rigorously tests the correctness of the algorithm implementation in the source code, compliance with secure coding practices, and the traceability between code and documents.
4. **Final Deliberation and Certification:** The validation authority (NIS) finally confirms whether the test results meet the standards and then issues the certificate.

The entire process typically takes about one and a half years. In particular, supplement requests arising during the technical review process cause additional delays of several weeks to several months at each cycle. Such repeated delays lead to increased validation costs and delayed product release schedules, and they create a practical demand for an automated pre-certification tool. Accordingly, this study proposes an automated pipeline to address this problem.

2.3 LLM- and RAG-based Technologies

A Large Language Model (LLM) is a neural network model pre-trained on massive amounts of code and natural language data, and it is used in a variety of software analysis tasks such as code understanding, pattern recognition, and natural language generation [28].

Google's Gemini 2.5 Flash-Lite [4], used at the L3 stage of this paper, is a multimodal large language model (LLM) optimized for low cost and high throughput; because it is optimized for low cost and high throughput and supports long context inputs, it is suitable for the L3 decision stage of this study, which jointly handles large code fragments and evidence documents.

Retrieval-Augmented Generation (RAG) is a methodology that combines external knowledge-based evidence with the responses of LLMs [12]. Since LLMs may not have included domain-specific regulations such as KCMVP guidelines in their pre-training data, a RAG architecture that dynamically injects external knowledge is needed. In this system, the Retrieval part of RAG is performed at

Table 1. Comparative analysis with related prior work. ✓ = supported, × = not supported.

System	Target	Crypto. Insp.	LLM decision	Doc. Insp.	Trace-ability	Evidence matching
CryptoGuard [26]	Java API	✓	×	×	×	×
CogniCrypt [11]	Java JCA	✓	×	×	×	×
CAVP [16]	Test vectors	✓	×	×	×	×
Proposed system	KCMVP	✓	✓	✓	✓	✓

the L2 stage to search the KCMVP guideline articles, and the retrieved evidence is inserted into the Gemini prompt context at the L3 stage.

2.4 Related Work

Table 1 compares the proposed system with related prior work along five functional dimensions. The comparison targets are representative prior studies on static-analysis-based detection of cryptographic misuse (CryptoGuard, CogniCrypt) and the algorithm testing automation tool of the U.S. CMVP ecosystem (CAVP).

CryptoGuard [26] leverages an inter-procedural slicing technique to detect 16 categories of cryptographic API misuse in Java. CogniCrypt [11] provides IDE-integrated inspection for the use of the Java Cryptography Architecture (JCA). Both tools have demonstrated the effectiveness of automated cryptographic-code inspection, but they do not address KCMVP-specific requirements or LEA.

The Cryptographic Algorithm Validation Program (CAVP) [16], operated by the U.S. NIST, is an algorithm testing automation tool focused on verifying the correctness of test vectors. It verifies the mathematical correctness of algorithm implementations based on publicly available test vectors, but does not provide functionality for analyzing implementation context in the source code or for verifying document traceability. Korea's existing KCMVP pre-certification system is likewise limited to test-vector accuracy testing [23]. Thus, while existing studies have demonstrated the feasibility of automating cryptographic-code inspection, no tool exists that simultaneously supports the incorporation of KCMVP-specific requirements, LLM-based semantic decision-making, and automated traceability between source code and documents. This study proposes a three-stage hybrid pipeline that integrates these three functions.

3 System Design

3.1 Design Goals

To maximize detection accuracy while minimizing False Positives, this pre-certification system subdivides the overall validation process into three mutually complementary stages: rule-based deterministic detection (L1), RAG-based guideline evidence retrieval (L2), and LLM-based final decision (L3).

Goal 1. Ensuring High Recall. This study aimed to achieve high recall through pattern-based deterministic detection. In particular, by applying the `missing` (absence of required patterns) and `regex` (string pattern match) types as explicitly defined rules, we pursue a broad-coverage search aimed at reducing omissions for violations that can be expressed as rules.

Goal 2. Minimizing False Positives (FP). Because simple rule-based detection may structurally include false positives (similar patterns, constants, and test code), this study aims to minimize false positives by examining both code and document context together with evidence through domain-specific filters and the LLM's final decision.

Goal 3. Evidence-based Simultaneous Validation of Code and Documents. This framework processes the source code (ZIP) and submitted documents (PDF) in an integrated manner within a single pipeline, and through the three-tier RAG retrieval at the L2 stage, it automatically attaches actual KCMVP guideline articles to each detected violation item, thereby securing the interpretability of validation results.

3.2 Overall System Architecture

The overall pipeline is designed as a funnel structure as shown in Fig. 1. First, the source code (ZIP) and the submitted documents are received together and preprocessed. On the code side, the system builds a structural summary containing function and call information along with a symbol graph that links calls across files; on the document side, body text and tables are extracted from the PDF and then structured into a section list using the table of contents and the like. This broadly secures the coordinates (file, line, section) at which rules can be applied.

At the subsequent L1 stage, inspection rules defined in YAML are applied across four pattern types. `missing` handles the absence of required elements, `regex` handles string-pattern matching, `semantic` handles keyword and contextual cues, and `ast` handles syntactic/structural cues. When this stage passes through heuristic filters, the broadly captured error-suspect list is consolidated into a candidate set.

At the next stage, KCMVP guideline evidence is retrieved and matched, centered on the rule identifier and the suspected code/paragraph (the suspect part). Evidence documents loaded into ChromaDB or similar stores can be utilized. The evidence documents are stored in markdown by dividing the KCMVP specification documents into topic and article units.

Next, the generative model receives the suspect part, decision instructions, and evidence together, and ultimately distinguishes false positives from actual violations. The flow of refining the broadly captured candidates from the previous stage through evidence- and context-based decisions corresponds to the narrowing section of the funnel. Eventually, the outcomes are consolidated into deliverables such as patch notes and reports, and rejected items are also kept as candidate history if needed so that they can be tracked.

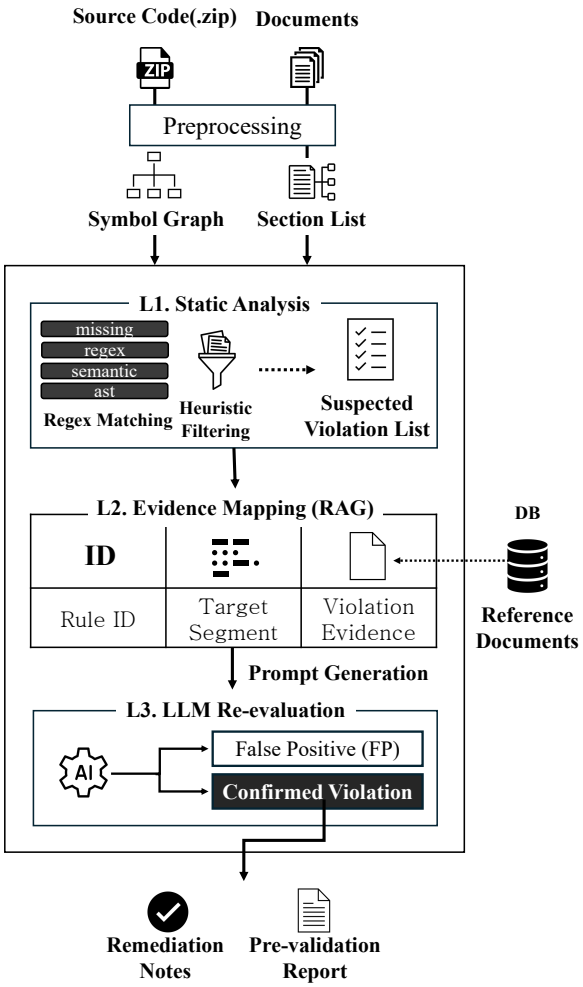

Fig. 1. Overall pipeline configuration diagram.

In addition, although omitted from the figure for clarity, in the implementation the evidence retrieval is invoked twice: once when constructing the prompt for LLM decision-making, and again, by the same module, when attaching evidence fields to the final violation list. The former is to inject specification evidence into the decision context, and the latter is to enable per-violation evidence to be verified in the report and UI.

Table 2 summarizes the main functions and roles at each stage.

Table 2. Per-stage summary of the proposed KCMVP pre-certification pipeline.

Stage	Analysis method	Main objective
Pre-processing	Source normalization; four-stage AST fallback; symbol graph construction; document OCR sectioning	Generating file, line, section, and call-relationship coordinates for rule application
L1	Per-pattern-type rule application; domain-anchor-based candidate refinement	Deterministic primary candidate detection and reduction of obvious false positives
L2	Three-tier RAG retrieval	Attaching KCMVP evidence and decision context for each candidate
L3	Semantic re-evaluation	Distinguishing actual violations from false positives and conservatively removing candidates
TRC	Cross-comparison of header APIs, source definitions, and documents	Traceability inspection across specifications, code, and test specifications

3.3 Per-stage Design of the Validation Pipeline

Preprocessing Stage. The rule engine and the LLM need location information that can be referenced repeatedly—such as file paths, source line numbers, document sections, and tables—so that they can write violation descriptions consistently for the same input and so that the surrounding code and paragraphs around the problematic location can be sliced and passed as context. Furthermore, security and implementation inspections are difficult to grasp with simple string searches alone, and the judgment becomes more accurate only when structural and call information—such as where a function is called from and what it calls—is available. Accordingly, a preprocessing step was introduced that converts source code into line-by-line text together with function-and-call summaries, and converts documents into separated body/table representations with a section-level structure, in order to improve accuracy. Specifically, structural information is extracted from the uploaded ZIP archive and PDF documents to generate analysis artifacts in JSON form.

AST and Symbol Graph. The Abstract Syntax Tree (AST) is an intermediate representation that expresses the syntactic structure of source code in tree form, and it is the standard output of the parsing stage in compiler design [1]. The symbol graph is a structured representation that connects symbolic information—such as function definitions, function calls, and global constant arrays extracted from multiple source files—through nodes and edges.

Because the AST focuses on how operations and statements are arranged and in what structure within a single file or function, it is suitable for inspecting implementation invariants inside a function; the symbol graph, by contrast, shows where each function is defined and which functions in other files it calls, and is therefore used to judge cross-file relationships such as cross-file zeroization and wrapper calls.

The C preprocessing stage directly affects the analysis results. In C/C++, the standard preprocessor (`#define`, `#include`, `#ifdef`) can directly modify the source structure at the token level prior to parsing, and this may cause the original-source location information and the actual syntactic structure after preprocessing to diverge. Because the C parser and rule engine rely not on a mere sequence of tokens but on types, constant values, macro expansion results, identifier interpretation becomes difficult and a malformed node structure may be generated if scattered headers are not appropriately restored.

Accordingly, this system combines `gcc -E` preprocessing, internal header tracking, and libclang-based AST and symbol graph construction, and adopts the following multi-stage fallback structure to improve the parsing success rate.

1. **Stage 1. libclang-First Parsing:** Full parsing of the source file is attempted using LLVM libclang. Since macro expansion and header inclusion are handled internally, the syntax tree can be constructed accurately even given the complex header dependencies often found in cryptographic module code. If parsing succeeds, the function definitions and call relationships are constructed to generate an AST summary together with the symbol graph.

   In the libclang parsing path, the system collects not only simple call information but also type information and cross-file linkage information. In the symbol graph generated through libclang, the key is the linkage based on Unified Symbol Resolution (USR). When functions with the same name exist across multiple files, or when the header declaration is separated from the implementation definition, simple name matching alone can leave the call target ambiguous. Because libclang allows the USR to be obtained from the referenced cursor of a function definition and a call expression, this system first links calls to definition locations via the USR, and falls back to name-based matching only when the USR is empty. For this reason, the call relationships in the symbol graph include not only the calling file, calling line, and called function name, but, when possible, also the definition file and definition line of the called function.

   In addition, the libclang path stores typedef aliases and static constant-array initialization values as separate evidence. typedefs are used to interpret `uint32_t`, function pointer types, and public API signatures, while static constant arrays are used to distinguish public constants—such as LEA delta, S-box, lookup table, and KAT/vector—from candidate secret keys. For arrays, the system locates `CONSTANTARRAY`-typed entries among `VAR_DECL`s, extracts tokens from the initialization list, and stores the variable name, element type, size, and initial-value samples in `array_inits`. This value later

serves as the key evidence for determining whether a hard-coded key candidate is an actual secret key or a public algorithmic constant.

2. **Stage 2. gcc -E + pycparser Preprocessing:** When libclang is unavailable or when a pycparser-based fallback is required, the system first uses the compiler's `gcc -E` preprocessing to produce C text with macros expanded, and then parses it with pycparser.
   In this case, parsing along the pycparser path is focused not on reproducing the entire compilation environment but on recovering, as stably as possible, the function definitions and call locations required for KCMVP precertification. To prevent errors in LLM judgment, block comments and line comments are removed while the number of newlines is preserved, so that the original line numbers are not skewed. As a result, the violation locations output by L1 and the code slices received by L3 can directly point to the original file locations submitted by the user.
   Next, types and standard function declarations that frequently appear in cryptographic implementations—such as `uint32_t`, `size_t`, `memset`, and `memcpy`—are inserted as a minimal preamble. For local headers, `#include` is followed recursively, numeric macros are converted into `enum(NAME = VALUE);`, simple alias macros into `typedef`, and function-like macros into fake declarations of the form `int NAME(int arg, ...);`. This processing removes most preprocessor directives while preserving the name information required for AST extraction—such as array sizes, type names, and macro function calls.
3. **Stage 3. Full Preprocessing:** This is the fallback stage when the compiler preprocessor cannot be used, and its purpose is to reflect, as much as possible, the type and name information contained in project-local headers even without a system preprocessor. After following only the quoted `#include` directives within the project to read headers, comments are removed, and only function-like macros whose name is immediately followed by a parenthesis—such as `#define` entries—are picked out and converted into fake function declarations of the form `int name`. These fixed declarations and macro declarations are placed at the front, and the body with directive lines removed is appended for parsing.
4. **Stage 4. Simple Preprocessing:** If the working root or file paths are missing, or if parsing with header inclusion fails, the system retries using only the fixed preamble and the in-source macro declarations. Only the preamble and the in-source `#define` macros are processed. If the procedure fails even at Stage 3, parsing is performed on the body alone after removing comments and replacing only preprocessor directive lines with blank lines.

If parsing fails at any stage, the system automatically switches to the next stage, and line-number differences caused by preamble insertion are converted back to the actual line numbers of the original source code via offset correction. Listing 1.1 shows an example of the JSON structure of the AST preprocessing result.

```
{
  "path": "src/lea_key.c",
  "ast": {
    "functions": [
      {
        "name": "lea_encrypt",
        "line": 8, "end_line": 20,
        "calls": [
          {"name":"lea_set_key","line":10},
          {"name":"memset_s","line":19}
        ]
      }
    ],
    "file_calls": [
      {"name":"lea_set_key","line":10},
      {"name":"memset_s","line":19}
    ]
  }
}
```

Listing 1.1. JSON schema of the AST preprocessing result.

Document Section Extraction. For documents, two libraries were utilized in order to correctly extract the text of both running prose and tables. Because tables require restoration of their grid structure, pdfplumber [27] is used to detect table candidates within each page, and table data is built as a two-dimensional array based on cell boundaries. At the same time, the bounding box of each table is recorded, and when PyMuPDF [2] is used to extract body text, text blocks that substantially overlap with such a box—measured by area—are excluded from the body candidates. The purpose of this design is to prevent the same content from being duplicated across running text and the table structure, and to separate the roles of each so that tables are kept as structured representations while other descriptions are kept as running body text.

After extracting both text and tables, section separation is performed so that violation regions can be expressed at the paragraph level. Section separation proceeds along two paths—table-of-contents-based and body-header-pattern-based—with the table-of-contents-based path taken first.

1. **Stage 1. Table-of-Contents-First Path:** After searching for the table of contents or "Contents" in the full text concatenated across pages, regular expressions corresponding to entries like "1. Title... 4", "Chapter N", "Section", or "Article page" are used to extract, for each entry, the identifier, title, and in-document page number. Then, by finding the physical page on which a word from the title of the first entry reappears in the body, the offset between the printed page numbers in the table of contents and the physical PDF pages is computed. For each table-of-contents segment, the body of the offset-corrected physical-page range is concatenated, and only tables

falling within the same page range are attributed to that section. From the body text, page headers and footers and page markers consisting entirely of Arabic numerals on a single line are removed so that unnecessary text is not included.

2. **Stage 2. Header-Pattern Path:** When no table of contents exists or when the procedure above fails to produce valid sections, the integrated body string is scanned line by line, and hierarchical-heading candidates are identified by regular expressions. After identification, the beginning of each section's body is located within the entire string to approximate the starting and ending physical pages.

Each per-section object holds the document type determined by the uploaded document's path, a logical file key obtained by appending the section identifier after the relative path with respect to the working root, the section title, the hierarchical depth inferred from the table of contents or header pattern, the start page and end page that the body occupies, the body text of that section, and the list of tables falling within that page range.

If neither header nor table of contents can be obtained from the two stages above, the entire file is grouped and marked as a single section. Ultimately, the unit of document preprocessing is divided into sections so that L3 can receive only the necessary sections together with adjacent sections to re-evaluate the document context.

Scanned PDFs without a text layer are automatically detected, and the Gemini 2.0 Flash Vision OCR model [5] is used to recognize each page image and extract text. This also enables automated rule validation on scanned documents.

L1: Rule-based Static Analysis. L1 is the layer that receives the outputs handed over by preprocessing and mechanically applies the inspection rules defined in the repository to produce a set of violation candidates. Fig. 2 illustrates the overall flow of L1.

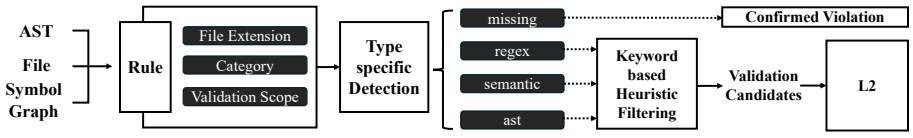

Fig. 2. L1 pipeline flow.

The inputs supplied through preprocessing include, for each source file collected relative to the working root, the original text, the body with comments removed, the per-file AST summary (function definitions, function call lists, etc.), and the project-wide symbol graph (call relationships across files, the set

of files defining zeroization-related calls, etc.). Based on this information, the rule engine evaluates the conditions declared in YAML.

Table 3. L1 inspection-rule categories.

Category	ID	Main inspection items
Common security	COM-xxx	Module-wide common security
LEA	LEA-xxx	LEA-specific constraints
Modes of operation	CBC/CTR/GCM/...	Per-mode-specific constraints
Documents	DOC-xxx	Required-section document format
Traceability	TRC-xxx	Design–code–test consistency

Table 3 shows the detailed classification of rule categories. The categories were designed on the basis of separating responsibilities in the order common code security → per-algorithm → per-mode → documents → traceability, and the rule schema was written following the 2025 Cryptographic Module Submission Authoring Guide [7].

Detection Pattern Types. Table 4 outlines the detection principle of L1 rules by pattern type and how the resulting violations are linked to the LLM-based re-evaluation stage (L3). Because the four types differ in the nature of the evidence each rule requires—absence, string, or syntactic structure—the same pipeline assigns different levels of reliability and downstream weight to each.

1. `missing` – An absence-type rule yields a violation when the regular expression described in `pattern` is not observed at all in the body after comment removal. The unit of application varies between `file` and `project` depending on `scope`, with the latter compressing the fact that the same signal never appeared anywhere in the repository into a single violation.
2. `regex` – The compiled pattern is applied across the entire original text; the line range is computed from the start and end offsets of the matched section, and each match is recorded as an independent violation. For specific identifiers, heuristic filtering is applied by means such as excluding surrounding context, variable names, and comment lines. Because objective string-level matching is the core principle, reliability is kept relatively high, but, depending on severity, items are classified for L3 re-evaluation.
3. `semantic` – A semantic-type rule treats a pattern as a keyword/trigger condition, but also includes cases that append additional procedures per identifier. If the keyword is not observed, a file- or project-level candidate violation reflecting the possibility of a missing topic is generated, and metadata is recorded indicating that the item should later be reviewed at the LLM-based semantic re-evaluation stage. If scope is not explicitly restricted to `file`, aggregation switches to the project level to suppress repeated occurrences of the same signal. Because this type requires contextual re-interpretation, it is re-evaluated at L3.

Table 4. Detection pattern types and their linkage to the L3 stage.

Type	Detection principle	Details	L3
`missing`	Absence check	Required pattern omitted	Optional
`regex`	Regular expression	Pattern matching	Optional
`semantic`	keyword + AI	Context-required pattern	Priority
`ast`	AST analysis	Structural inspection	Priority

4. `ast` – A syntactic-type rule is for cases where the full context is needed, and a dedicated checker that directly judges the structural properties of the Abstract Syntax Tree (AST) produced through preprocessing is run preferentially. Here, a dedicated checker is a specialized function that traverses the pre-parsed AST per rule and structurally judges whether the list is violated; these are predefined for the 20 rules of the LEA algorithm and refine the candidates before sending them to L3. If the checker detects a rule violation, it is flagged; if no violation is found, the rule is considered satisfied and the inspection terminates. Through this, the code lines suspected of violation are extracted as a candidate group and re-evaluated at the L3 stage.

Domain-Anchor-based Candidate Refinement. In this study, a domain anchor is a structural and contextual cue used—before applying a given rule—to judge whether the analyzed file or code fragment falls within the semantic scope of that rule. The L1 rule engine generates candidates broadly to achieve high recall, but because cryptographic module submissions contain public constants, test vectors, wrappers, benchmarks, and mode-of-operation-specific files, together, simple rule application alone increases false positives. Accordingly, this system uses anchors before and after rule application to refine the candidates.

Domain anchors are broadly divided into identification anchors and exclusion anchors. An identification anchor is evidence that strengthens the case that a particular piece of code is an actual rule-application target. For example, when LEA round loops, accesses to round-key arrays, ROL/ROR-based ARX operations, and references to the delta constant appear together, the file is likely to be a LEA implementation file. Conversely, an exclusion anchor is evidence that excludes targets which superficially look like violations but are not violations in the actual validation context. The LEA delta constant, KAT input values, S-boxes, wrappers' simple delegation functions, and benchmark-only fixed plaintexts fall into this category.

Candidate refinement in this system uses both kinds of anchors together to retain structural violations that should be detected while reducing low-value false-positive evidence. Through this, the candidate quality of L1 is improved, and only candidates that genuinely require contextual judgment are forwarded to L3.

```yaml
- id: COM-001
  name: "Residual data clearing (zeroization)"
  category: common
  scope: project
  pattern_type: "missing"
  pattern: "<safe zeroization API names>"
  severity: high
  kcmvp_ref: "KS X ISO/IEC 19790:2015 ..."
```

Listing 1.2. YAML-structure summary of the COM-001 rule.

YAML Rule-Set Structure. Listing 1.2 shows an example of a YAML-based rule set generated based on the Table 4 patterns. Rules are identified and classified by `id`, `name`, `category`, and `scope`; the pattern is set in `pattern_type`, and the regular expression is entered in `pattern`. `severity` and `kcmvp_ref` are auxiliary information for severity grading and evidence citation.

L2: RAG-based Evidence Matching. L2 is the stage that supplies context through evidence retrieval and input construction. Fig. 3 illustrates the overall flow of the L2 stage. The violation candidates identified at L1 are passed in as individual instances, and for each item, the minimal metadata required for decision-making—rule identifier, pattern type, file and line location, violation description, and so on—is organized. At the same time, the surrounding source (or document) segment around that location is extracted as a code/document slice in order to reduce the input tokens to the model while still preserving context.

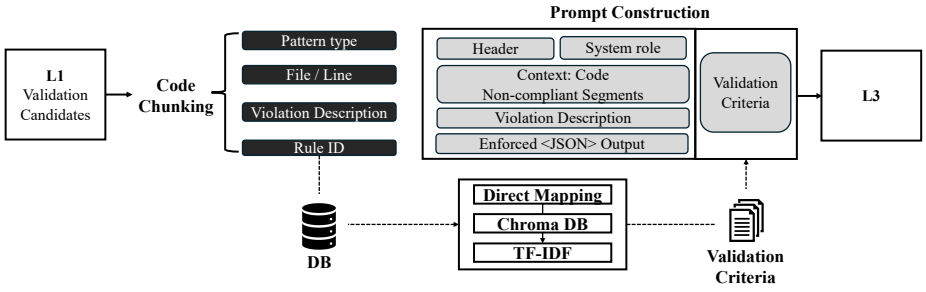

Fig. 3. L2 pipeline flow.

Next, by matching with rule `id` as the key, evidence documents are attached from KCMVP-related guidelines and technical materials stored in the DB. Finally, the collected slices, violation descriptions, and retrieved evidence are bun-

dled into a single prompt together with the header, system role, and output-format instructions, and submitted to the language model; the response is then forwarded to L3.

Decision-Use Context Excerption. If the entire original text is fed verbatim to the language model, cost and latency grow with the input length, and as more unrelated sentences appear, attention is diluted and decision quality may degrade. Accordingly, only the minimal contiguous segment necessary for judgment is excerpted. The excerpt width is varied to match the pattern type and the document structure, preserving local context—such as call relationships, surrounding conditions, and the flow across adjacent clauses—without breaking continuity, while staying within the token budget.

1. **Code Slicing.** Based on the file path and line position, a contiguous block of source lines centered on that point is sliced to compose a snippet for decision-making. The window size before and after is varied by pattern type: `regex` violations use narrow context, while `semantic`/syntactic candidates use relatively wider context so that surrounding declarations and call relationships can also be examined.
2. **Document Excerption.** For document re-evaluation, the system relies on the clause-level structure obtained from preprocessing, but instead of showing only a specific clause, it bundles several clauses belonging to the same document type in order to create a single block of context. The intent is to also examine whether equivalent content is described in other clauses even if the clause flagged by the violation contains no keywords. However, because the body of each clause can be very long, only the front portion of each clause is sliced so that a single clause does not occupy the entire input, and an overall upper bound is also placed on the concatenated string when multiple clauses are joined, thereby controlling the amount and cost the model processes at once.

Three-Tier Retrieval Structure. Evidence retrieval is divided into three stages, with the most reliable path attempted first. The case in which a rule is directly linked to the original text of an official guideline is prioritized highest in order to clarify provenance; only when that result is absent or insufficient does semantic search reinforce or replace it with related passages. If gaps still remain, the procedure falls back to a path that sweeps a broader document set by keyword and statistical similarity, ultimately focused on reducing evidence gaps.

1. **Direct Mapping (Tier 1):** Corresponding to the rule-set identifier (`id`), the designated markdown guideline is loaded according to that relationship. The guideline was constructed as candidate evidence chunks by segmenting items based on the cryptographic-module guideline documents [7,22,21]. When a clause title directly reveals meaning useful for the decision—such as requirements, violation patterns, or commentary—priority is assigned to place it at the front, and the remaining clauses are appended at lower priority; finally, the number of chunks to be returned is truncated to a cap.

Because this presupposes that the reviewer has explicitly linked the rule with the official document, the specificity and reproducibility of the sources are enhanced.

2. **ChromaDB Vector Search (Tier 2):** This is a path that splits the guideline markdown into clause/paragraph units in the vector store (ChromaDB), encodes each fragment as a real-valued vector using an embedding model, projects the query sentence into the same space, and selects semantically similar fragments. When a specific guideline clause has already been secured by Direct Mapping, the existing list is not replaced; instead, a vector search is performed with the same query, and a small number of fragments from different sources are appended to broaden the evidence. When Direct Mapping is empty, metadata such as article identifiers is used as a filter to find chunks semantically close to the query, which serve as the primary returned results. Through this, meaningful evidence is reinforced or substituted without breaking the explicit rule–document linkage.
3. **TF-IDF Keyword Fallback (Tier 3):** The TF-IDF path ranks each chunk by a similarity score that reflects the term frequency and the rarity across the document set as a whole, and returns only a small top-k. The query is either provided at call time or uses the search phrase stored in the rule mapping. It is invoked as a last resort when vector search is disabled or fails to produce valid results; while it does not guarantee semantic agreement, it serves as a conservative safety net that quickly retrieves lexically close clauses from a broad document set to mitigate evidence gaps.

The evidence chunks selected and ordered as above are injected into the prompt as a reference-guideline block, and they are also added to the evidence field of the final report JSON in summary form to provide evidence.

Prompt Construction. After the above excerption and evidence retrieval, the prompt is constructed. Going beyond simple role instructions and code-insertion structures, the prompt design of this system adopts a Hierarchical Hybrid Prompting architecture that combines eight techniques across three tiers in order to simultaneously achieve decision accuracy and false-positive suppression.

Tier 1 serves as a frame that fixes the judgment perspective. Persona Prompting assigns the model the role of a "KCMVP cryptographic-module security review expert" so that decisions are made not on general security knowledge but according to KCMVP rules. RAG Guideline Injection inserts the KCMVP guideline excerpts collected during the L2 evidence retrieval into the front part of the prompt, making the rule provenance and decision criteria explicit. GCFS (Global Code Flow Context Injection) compresses the function definitions and call relationships of the symbol graph into a per-file key-lifecycle flow, conveying to the model the project-wide key creation – use – destruction flow that is not visible from a single snippet alone.

Tier 2 is applied selectively according to the characteristics of each rule. Few-shot examples are provided for 18 rules that are difficult to grasp contextually, presenting boundary-case decision criteria as concrete code patterns through violation, normal, and FP code examples. Structured evidence explicitly inserts

Table 5. Configuration of the Hierarchical Hybrid Prompting architecture.

Technique	Role
Judgment-frame setting	
Persona Prompting	Assigns the role of a KCMVP reviewer
RAG Guideline	Injects specification evidence into the prompt
GCFS	Summarizes the project-wide key lifecycle
Per-rule judgment execution	
Few-shot examples	Provides violation, normal, and FP decision examples
Structured evidence	Makes type, array, and call-path evidence explicit
Chain-of-Thought (CoT)	Guides stepwise reasoning
Confidence request	Supports post-processing using a 0–100 confidence score
Result-safety verification	
Asymmetric FP threshold	Imposes a higher confirmation cost for false-positive removal
Double verification	Blocks FN risk through independent secondary verification
Rejudge	Re-reviews low-confidence violations

type aliases, array initialization values, function parameters, and call paths for rules that depend on the AST checker and the symbol graph. CoT is applied to 23 rules in which multi-step contextual understanding is critical—such as LEA round-key index boundaries and the CBC/CTR/GCM initialization order—and makes the stepwise reasoning order explicit to guide the model's chain of reasoning. The confidence request requires, for every candidate, a 0–100 integer in the `confidence` field, and this value is utilized in the Tier-3 post-processing.

Tier 3 is the post-processing layer that verifies the safety of the decision results. The asymmetric FP threshold sets confidence < 25 for `ast`/`semantic` candidates and confidence < 40 for `regex` candidates as the criterion for FP removal, and forces the violation to be retained regardless of the model's decision when confidence ≥ 80, thereby protecting recall. Double verification performs secondary verification for decisions that lean toward FP removal, using a "senior auditor" persona, thereby blocking the FN risk that an actual violation is mistakenly removed as a false positive.

L3: LLM-based Semantic Re-evaluation. L3 is the stage in which the language model reads the context for violation candidates received through the prompt and finally decides whether each is a false positive or a violation. Receiving the results of L1 and L2 as the prompt input, it performs the final decision using Gemini 2.5 Flash Lite. Fig. 4 illustrates the overall flow of the L3 stage.

LLM Decision. The model's decision criteria are built into the prompt as follows. The rule statement corresponding to `rule_id` is composed of a per-rule decision-use string dictionary and the `ai_context` (specification details) carried over from

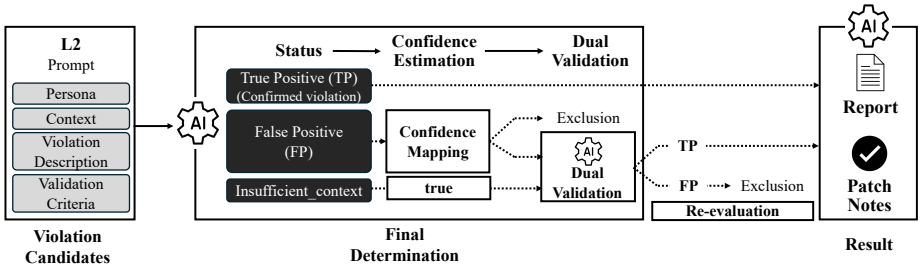

Fig. 4. L3 pipeline flow.

YAML and similar sources, and when necessary, KCMVP guideline excerpts retrieved by RAG during the L2 (evidence retrieval) stage are placed in the same prompt block. The prompt branches into per-type instructions according to `pattern_type`, designed so that the false-positive/violation conditions and the lower bound of confidence differ between absence types (`semantic`, `ast`) and presence types (`regex`, etc.).

The LLM reads the code slice given with this prompt and determines whether it is an actual violation or a false positive; in the `is_real_issue` field of the structured output, if `true` the violation is retained, and if `false` it is finally excluded as a false positive. The decision rationale is written in `description` so that users can verify the evidence when reviewing patches and recommendations, thereby increasing the explainability of the LLM's decisions.

In addition, if the confidence of the first response falls roughly in the range 65–74, a re-evaluation prompt is sent once more to reorganize the reasons and the score. When sufficient context is not provided from L2, `insufficient_context` may be set to `true`, allowing the process to avoid declaring a violation outright on the basis of a short excerpt and instead leave it conservatively or pass it on for further review.

```
{
  "is_real_issue": "<true | false>",
  "confidence": "<integer 0..100>",
  "description": "<brief rationale, 2-3 sentences>",
  "suggestion": "<one-line remediation>",
  "insufficient_context": "<true | false>"
}
```

Listing 1.3. Example output structure of the LLM.

Final Output. Listing 1.3 shows the skeleton of the structured final response that the implementation requires from the model. If only free-form description were allowed, it would be difficult to reliably automate report aggregation and the linkage of patch/recommendation wording, and because the decision is hard, the LLM is required to produce the final output in JSON form.

As the final output, `is_real_issue` is the flag that decides whether to keep the given candidate in the final violation set or to remove it from the list, and `confidence` preserves the numerical strength of the decision even within the same boolean classification, serving as re-review metadata. `suggestion` fixes the direction of correction in a single line so that it can be carried as-is into a patch-generation prompt or a UI recommendation. As described above, `description` is the field that records the decision rationale in a human-readable form, and `insufficient_context` serves as a signal for downstream processing and display, indicating that the model will not commit to a decision from this call alone because the slice was too short.

Patch Notes and Report Generation. After the L3 final decision, the violation list is sorted by severity and confidence, and LLM-based code and document patches are automatically generated. Code patches derive correction examples based on the L2 RAG evidence and the violating code, while document patches provide authoring examples grounded in the KCMVP guidelines for design omissions. Finally, a validation report containing the violation statistics and the LLM's overall assessment (pass likelihood and revision priorities) is produced in markdown and PDF form.

Traceability Verification. In KCMVP certification, not only the source code but also the design specification, configuration management document, test specification, and module must remain mutually consistent. Whether the APIs implemented in the code are specified in the design specification, whether the error codes mentioned in the design specification are defined in the actual source, and whether the public functions are sufficiently exposed in the test specification are items that must be verified at the technical review stage. The traceability verification of this system takes the code summary produced at the preprocessing stage and the document preprocessing results, and performs a cross-comparison across the three document–code axes.

Cross-Comparison Mechanism. On the code side, public function declarations of the form `[return type][function name]([parameters]);` are extracted from header files using regular expressions. `static` and underscore-prefixed functions are regarded as internal implementations and excluded; only public APIs are taken as traceability targets. From source files, function definitions corresponding to header declarations are extracted to construct declaration–definition pairs, and error-code constants declared by `#define` (`ERR_*`, `KCMVP_ERR_*`, etc.) are collected separately. In addition, the definitions and call graph of the symbol graph are used as reinforcing means so that the linkage can still be identified even when an API is delegated to another file through a wrapper.

On the document side, function names, error codes, and test items are searched within the design specification and test specification structured at the clause level by the PDF preprocessing. For the design specification, the body and tables of the API-specification clauses are parsed, and the function-name column

is matched against the code name list; for the test specification, test-item names and references to test-target APIs are extracted. Based on this, a three-way comparison is performed.

1. **Design Specification vs. Header:** The list of public APIs extracted from headers and the list of API references extracted from the design specification are cross-compared; header functions not listed in the design specification are recorded as "undocumented" candidates, and design-specification APIs not implemented in the header are recorded as "unimplemented in code" candidates.
2. **Design Specification vs. Source:** Error codes declared via `#define` and the error codes mentioned in the design specification are compared bidirectionally, and error codes existing only in the design specification and those existing only in the code are each classified as violation candidates.
3. **Test Specification vs. Header:** Whether core public functions—such as encryption/decryption, key setting, and initialization—are mentioned as test items in the test specification is checked, and missing items are recorded as violation candidates.

If inconsistencies are found, the corresponding API names, error codes, and test items are recorded as violation candidates and included in the final report.

However, the current traceability-verification implementation has limitations because it relies on regular expressions and name-based matching. Function definitions wrapped by macros, function pointers, and multi-line declarations may be missed during regex extraction, and when function names are listed in tabular form in the design specification or when the code and document use different naming conventions, matching may fail. For this reason, traceability-verification results are reported as review recommendations rather than confirmed violations, so that a human can ultimately make the final judgment.

4 Prototype Implementation

4.1 Implementation Environment

This interface is designed with the purpose of enabling users to clearly specify their analysis settings and submission materials and then review the entire pipeline's outputs without missing anything. That is, the interface treats code and submitted documents as equal validation targets; separates the roles of navigation, body, and violation information so that users can quickly verify and judge the mechanical detection results in context; and has the final report reveal the summary and details sequentially, with an emphasis on reducing the cognitive load that may arise during pre-conformance inspection and on improving readability for the user.

4.2 Prototype Screen Layout

Initial Screen. Fig. 5 shows the initial screen of this pre-certification tool. The user can specify analysis conditions—such as the security level, product category, cryptographic algorithm, and mode of operation—matching the conditions of the actual test application, then submit the source code via a GitHub URL or a ZIP archive, and additionally upload PDFs by document type, including the design specification, configuration management, and test specification.

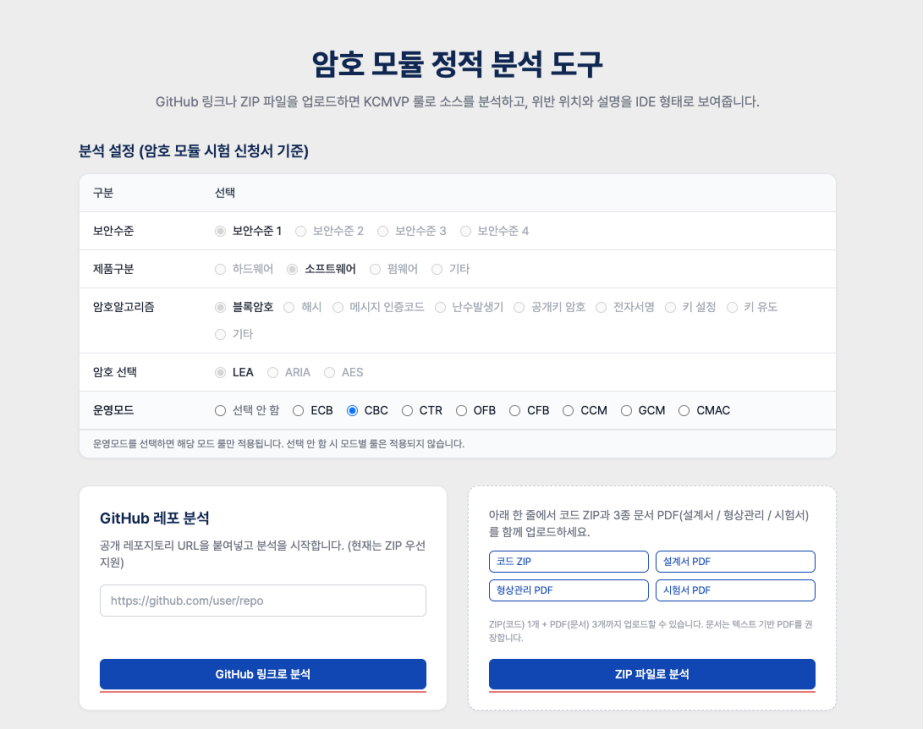

Fig. 5. Initial-screen layout.

Main Screen. Fig. 6 and Fig. 7 show the main verification screen that compares code and violations after the analysis is complete. The review screen of this system adopts a three-pane layout. The layout is arranged so that the user naturally proceeds in the following order: selecting the file to analyze from the file tree on the left, verifying the violation locations through line numbers and highlights in the central code/document viewer, and viewing the rule, severity, confirmed/candidate status, and patch notes at a glance in the violation list on the right. At this point, clicking on a violation in the list immediately jumps to

its location, enhancing user convenience. A violation-count badge next to each file name allows the user to prioritize which file to view first.

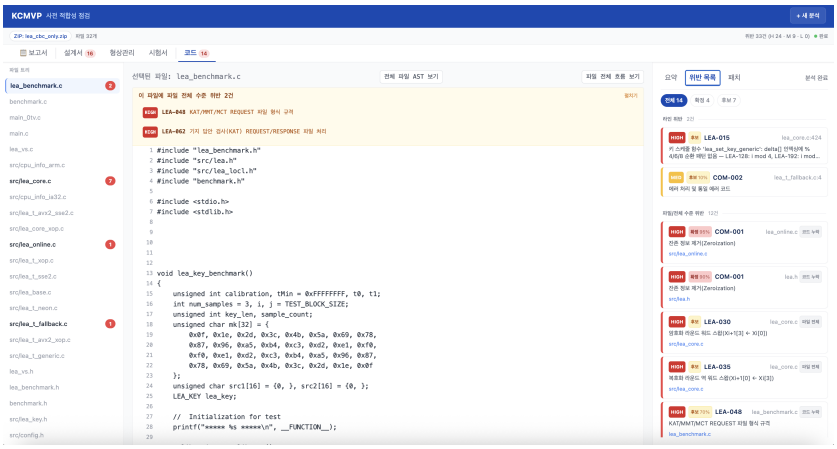

Fig. 6. Main screen and panel structure.

Final Report Screen. The report (see Fig. 7) screen summarizes confirmed violations, violation candidates, and totals as cards at the top, and divides the inspection axes—such as common security, algorithms, modes, documents, and traceability—into per-category tables, introducing an information hierarchy that lets the user first identify where failures or review needs arose before drilling down into details. Alongside output buttons such as PDF and print, the right panel additionally shows the per-source (code/document) counts as auxiliary information, and finally the AI summarizes the content so that the results can be checked at a glance.

Ultimately, an IDE-style three-pane layout was adopted to provide an integrated working environment in which the analysis target and the analysis results can be examined simultaneously.[1]

[1] The actual implementation video can be viewed at `https://www.youtube.com/watch?v=6zdJuxVIIgmE`.

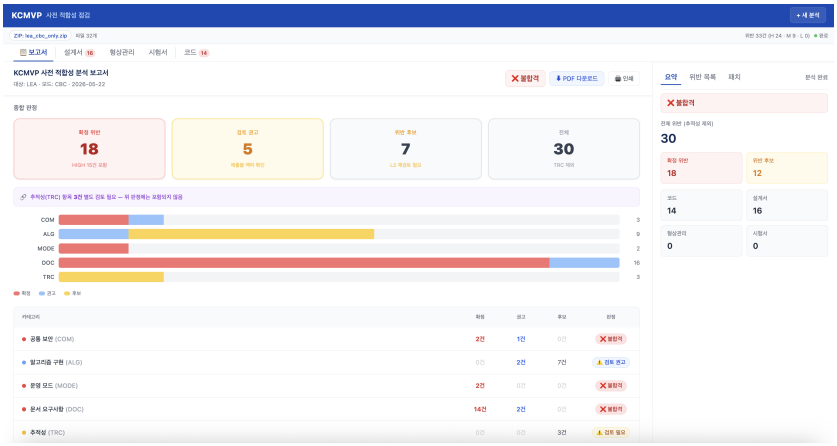

Fig. 7. Main screen and Patch Notes.

Table 6 shows the main technology stack used to implement the system.

5 Evaluation

The evaluation of this study was carried out on the web prototype implemented in Sect. 4.

In this study, the evaluation is conducted on a verified violation dataset from which FPs have been removed through analysis of actual KISA LEA code. The quantitative detection-performance evaluation in Sect. 5.1 was derived not from modules that have actually obtained KCMVP certification, but from synthetic data with violations intentionally inserted.

5.1 Experimental Evaluation

Test Dataset. Based on a C-language implementation of the LEA algorithm [6] and the security policy document of a validated cryptographic module [15],

Table 6. System implementation technology stack.

Component	Technology used
Backend	FastAPI (Python 3.11+)
Frontend	React 18 + Zustand
C/C++ parser	libclang [13] + pycparser [3]
PDF extraction	PyMuPDF [2] + pdfplumber [27] + MarkItDown [14]
LLM	Gemini 2.5 Flash Lite [4]; Gemini 2.0 Flash Vision OCR [5]
Vector store	ChromaDB
Rule format	YAML

Table 7. Code violation detection performance (Sets 1–7, based on 128 GT cases).

Metric	Result
True Positive (TP)	128 cases
False Negative (FN)	0 cases
Recall	100%
L1-stage False Positive candidates	46 cases
L3 filtering: FP removal	9/46 (19.6%)
L3 filtering: misjudgments (FN-inducing)	0 cases
Final False Positive (FP)	37 cases
Precision	77.6%
F1-score	87.4%

we constructed an archive in which rule-set violations were intentionally injected. The evaluation dataset consists of seven sets of code-ZIP and design-specification-PDF pairs, covering major validation areas such as the CBC and CTR modes of operation, key management, zeroization, random-number generation, and the LEA key schedule, round function, and Monte Carlo Test (MCT) loop. The Ground Truth (GT) is based on 128 verified cases obtained by removing FPs through analysis of actual KISA LEA code.

Code Violation Detection Performance. The objective of this evaluation is to quantitatively verify the overall performance and effectiveness of the KCMVP pre-certification pipeline in terms of detection accuracy and the contribution of LLM re-evaluation. First, recall was selected to objectively measure the degree to which the pre-certification system meets its core requirement of minimizing missed violations. In addition, through the L3 filtering-effect indicator, we quantified both the system-level contribution through which the LLM re-evaluation layer substantively reduces the user's review burden, and the frequency at which the side effect of mistakenly removing actual violations occurs during this process. The quantitative measurement results based on the evaluation metrics defined above are shown in Table 7.

The quantitative evaluation results in Table 7 consistently confirm the per-stage detection characteristics and performance trends of this system. First, the preprocessing and L1 stages are by design optimized for identifying the absence of specific patterns; they identify critical "missing"-type KCMVP violations—such as the non-use of safe zeroization functions or CSPRNGs—with a high detection rate, achieving a Recall of 100% (128/128 cases) on Sets 1–7. While L1 and the AST-dedicated checkers showed strong detection capability even on complex code, detection relying on simple keyword matching inevitably gave rise to false-positive (FP) candidates. However, the subsequent L3 stage demonstrated its AI effectiveness by semantically analyzing candidates such as improper matches against other algorithms' rules or misidentifications of public constants as cryptographic keys, accurately filtering out 9 of the 46 FP candidates generated at the L1 stage (19.6%) without inducing any false negatives. In addition, the

Table 8. Blind verification results on a real cryptographic module.

Metric	Result
Codebase / LOC	34 files / 14.5K
Total detections	7 cases
FP / per-candidate detection frequency (cases/KLOC)	0.58 cases/KLOC
Residual data clearing candidates	5 cases
API ordering violations	1 case
LEA memory clearing related	1 case

document-verification engine also showed high effectiveness by detecting, across the design specifications of the seven sets, frequently omitted critical items such as secret-key lifecycle management. On the other hand, because `missing`-type rules judge the absence of a specific pattern as a violation, even in a normal project in which the relevant functionality is delegated to other files, the system may incorrectly judge functionality to be absent if the number of files is small.

Blind Verification on a Real Cryptographic Module. In addition to the synthetic dataset, we conducted blind verification on the source code of a commercial cryptographic module developed by a domestic security company for actual KCMVP certification submission (34 C files, approximately 14.5 KLOC). Because this codebase includes multiple algorithms—such as LEA, ARIA, EC-KCDSA, CTR-DRBG, HMAC-DRBG, and PBKDF—and has already passed cryptographic-module validation, the detected counts were not assumed to be actual violations, but were conservatively regarded as false positives or additional-review candidates for the measurement. The detection results are shown in Table 8.

On a complete commercial codebase of 14.5 KLOC, the detection frequency was low at 0.58 cases/KLOC, and no additional L3 removal errors were observed. This demonstrates that the system does not merely flag potential violations broadly, but rather produces violation candidates centered on items for which rule evidence and code context are confirmed. Accordingly, we verified that a high detection rate is maintained without FPs increasing excessively in real-world environments.

The items that remained as additional-review candidates are as follows. For residual data clearing omissions (COM-001), even when the source code contains a zeroization statement, the compiler may judge it as a dead store and eliminate it, making it difficult to confirm a violation by source-level analysis alone. Because of this structural characteristic, the current system is limited to detection at the level of source-code pattern matching, and the future introduction of intermediate-representation (IR)-based analysis or symbolic-execution techniques that account for compiler optimization will also be required.

Items corresponding to the absence of test infrastructure do not surface in functional behavior tests and are currently detected only through rule-based checks. This system implements a basic level of traceability verification, but, because of its reliance on regular-expression-based function extraction, omis-

sions appear to have occurred for non-standard naming conventions and macro-wrapped functions. It is expected that detection accuracy for this type of issue can be improved by combining it with file-tree-structure analysis and reinforcing it with a complete AST-based implementation.

5.2 Limitations Analysis

1. **TRC traceability verification's reliance on regular expressions.** Because matching depends mainly on regular expressions and naming rules for whether header function declarations or in-document references exist, there is a risk of omissions and misidentifications for non-standard naming conventions, macro-wrapped APIs, and the like.
2. **The gap between the evaluation dataset and real-world environments.** Since this evaluation, centered on publicly available and synthetic materials, does not fully represent the scale or coding practices of commercial codebases that actually apply for KCMVP certification, a follow-up evaluation that utilizes actual confidential data through cooperation with certification authorities is required.
3. **Evidence extraction limitations due to document expression diversity.** Security policy documents and design specifications may describe identical requirements using different terminology and structures depending on the authoring organization, format, and version. In particular, for documents centered on tables, appendices, scanned images, and abbreviations, context may become fragmented or inter-item relationships may be lost during the PDF text extraction process. In such cases, the system may fail to retrieve relevant evidence sufficiently, or may link sentences that are semantically adjacent but are not direct evidence. Accordingly, future work requires reinforcement of document structure recognition, table-level parsing, and per-requirement evidence normalization techniques.

6 Conclusion

In this study, we proposed and implemented a hybrid framework for KCMVP pre-compliance inspection that ties rule-based static analysis (L1), RAG-based guideline-evidence attachment (L2), and LLM-based semantic re-evaluation (L3) into a single funnel-shaped pipeline. The system applies rules—covering common security, LEA implementation constraints, multiple modes of operation, submission-document formats, and code–document traceability—that are formalized in YAML, and through the KCMVP guideline mapping linked to the rule identifiers, it allows evidence to be attached to the interpretation of the results.

Through the prototype evaluation, we confirmed the effectiveness of the complementary structure in which the deterministic L1 secures a broad detection coverage and L3 reduces false positives through contextual reasoning. The multi-layered evidence retrieval at the L2 stage attaches standards and guideline citations to the violation items, aiding interpretability and the prioritization of

remediation. In addition, the system incorporates practical mechanisms tailored to the non-standard nature of real code, such as preprocessing/parsing fallbacks, domain-oriented false-positive mitigation, and multi-stage detection of common-security rules.

On the other hand, this system still remains a pre-certification auxiliary tool. Some of the AST rules rely on a regex fallback, the traceability verification has limitations centered on declaration and text matching, and the experiments were conducted primarily on publicly available and synthetic materials, so they cannot be regarded as representative of the entire set of actually submitted modules. Future work is as follows.

- **Advancement of AST Analysis:** Migrate the rules that currently rely on fallbacks to structural analysis by utilizing pycparser, tree-sitter, and similar tools.
- **Strengthening of Traceability Verification:** Extend the extraction and matching logic to encompass macros, multi-line declarations, and API mappings listed in design-specification tables.
- **Validation on Real Data:** Conduct reproducibility evaluations using actual submission deliverables in part, through cooperation with certification authorities and industry.
- **Expansion of Algorithm and Mode Coverage:** The current LEA-centered support is to be extended in two directions. (1) Within block ciphers: expand the rule set to other KCMVP-approved block ciphers such as AES, SEED, and ARIA. (2) Across cryptographic primitive types: investigate extension to hash functions (SHA-256, SHA-3, LSH, etc.) and public-key algorithms (EC-KCDSA, RSA, etc.). Hash functions require inspection items distinct from block ciphers, such as input-length restrictions, padding handling, and fixed initialization-vector verification. Public-key algorithms demand stateful, composite checks including key-length adequacy, PRNG coupling, signature-nature-verification ordering, and certificate-chain handling. In particular, because public-key operations (key generation, signing, and verification) are distributed across separate functions and files, the current single-file-centric AST analysis must be strengthened into cross-file call-flow analysis.
- **L3 Operational Optimization:** Study the cost–quality balance, including transitions to local and open-source LLMs, call limits, batching strategies, and regression testing.

Building on these directions, we expect that this work can lead to subsequent standardization and tool development aimed at reducing the review costs entailed by certification preparation.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Pearson Education, 2nd edn. (2006)

2. Artifex Software: PyMuPDF: Python bindings for MuPDF. [Online]. Available: https://pymupdf.readthedocs.io/ (2024), accessed: 2026-03-29
3. Bendersky, E.: pycparser: Complete C99 parser in pure Python. GitHub repository (2008), [Online]. Available: https://github.com/eliben/pycparser. Accessed: 2026-03-29
4. Google: We're expanding our Gemini 2.5 family of models. Google Blog (2025)
5. Google DeepMind: Gemini 2.0 flash. Google DeepMind Blog (2025), [Online]. Available: https://deepmind.google/technologies/gemini/flash/. Accessed: 2026-03-31
6. Hong, D., Lee, J.K., Kim, D.C., Kwon, D., Ryu, K.H., Lee, D.G.: LEA: A 128-bit block cipher for fast encryption on common processors. In: WISA 2013. Lecture Notes in Computer Science, vol. 8267. Springer (2014)
7. Korea Internet & Security Agency (KISA): Cryptographic module submission authoring guide. Tech. rep., KISA (Sep 2025)
8. Korea Telecommunications Technology Association (TTA): 128-bit symmetric block cipher (SEED). Tech. Rep. TTAS.KO-12.0004/R1, TTA (2005), december 2005. In Korean
9. Korean Agency for Technology and Standards: KS X ISO/IEC 19790:2015 – information technology – security techniques – security requirements for cryptographic modules (2015), korean adoption of ISO/IEC 19790:2012
10. Korean Agency for Technology and Standards: KS X ISO/IEC 24759:2015 – information technology – security techniques – test requirements for cryptographic modules (2015), korean adoption of ISO/IEC 24759:2014
11. Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., Kamath, R.: CogniCrypt: Supporting developers in using cryptography. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE (2017)
12. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Advances in Neural Information Processing Systems 33 (NeurIPS) (2020)
13. LLVM Project: Clang: A C language family frontend for LLVM. [Online]. Available: https://clang.llvm.org/ (2010), accessed: 2026-03-31
14. Microsoft: MarkItDown: Python tool for converting files and office documents to Markdown. GitHub repository (2024), [Online]. Available: https://github.com/microsoft/markitdown. Accessed: 2026-03-29
15. National Cyber Security Center (NCSC): List of validated cryptographic modules. [Online database], KCMVP. [Online]. Available: https://www.ncsc.go.kr/, accessed: 2026-05-21
16. National Institute of Standards and Technology (NIST): Cryptographic algorithm validation program (CAVP). [Online]. Available: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program, accessed: 2026-03-31
17. National Institute of Standards and Technology (NIST): Security requirements for cryptographic modules. Tech. Rep. FIPS 140-2, NIST (2001), may 2001, updated December 3, 2002
18. National Institute of Standards and Technology (NIST): Security requirements for cryptographic modules. Tech. Rep. FIPS 140-3, NIST (2019), march 2019
19. National Institute of Standards and Technology (NIST): Advanced encryption standard (AES). Tech. Rep. FIPS 197-upd1, NIST (2023), updated May 9, 2023

20. National Intelligence Service (NIS): Guidelines for cryptographic module testing and validation. Tech. rep., National Intelligence Service (2025), nIS Guideline (enacted 2021-11-01, revised 2025-06-01). In Korean
21. National Intelligence Service (NIS) and National Security Research Institute (NSR): Guide for vendor implementations part 2: Implementation guide for validation-target cryptographic algorithms. Tech. rep., NIS/NSR (Mar 2024)
22. National Security Research Institute (NSR): 128-bit block cipher LEA specification. Tech. rep., NSR
23. National Security Research Institute (NSR): Cryptographic Module Pre-validation Service User Guide. KCMVP (2023), manual v1.0. [Online]. Available: `https://www.kcmvp.or.kr`. In Korean
24. Presidential Decree of the Republic of Korea: Cyber security work regulations (2024), presidential Decree No. 34287 (partial amendment 2024-03-05, effective 2025-01-01). In Korean
25. Presidential Decree of the Republic of Korea: Article 69 of the enforcement decree of the electronic government act (2025), presidential Decree No. 35948 (amendment by other law 2025-12-30, effective 2026-01-02). In Korean
26. Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.: CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM (2019)
27. Singer-Vine, J.: pdfplumber. GitHub repository (2016), [Online]. Available: `https://github.com/jsvine/pdfplumber`. Accessed: 2026-03-29
28. Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.Y., Wen, J.R.: A survey of large language models. arXiv preprint arXiv:2303.18223 (2023)