SPEC2CODE: Mapping Protocol Specification to Function-Level Code Implementation

Yuekun Wang[1], Lili Quan[1*], Xiaofei Xie[1], Junjie Wang[2], and Jianjun Chen[3]
[1]Singapore Management University, Singapore
[2]Tianjin University, China
[3]Tsinghua University, China

***Abstract*—Protocol specifications, defined in Request for Comments (RFCs), play a critical role in ensuring the correctness of protocol software systems. To check consistency, specification–implementation pairs are essential for testing and verification. However, existing efforts in specification-to-code mapping remain largely manual and are typically limited to the file level, lacking the fine-grained granularity needed for function-level analysis, which is crucial for effective consistency checking. To address this gap, we present SPEC2CODE, the first LLM-driven framework that automates fine-grained mapping from protocol specifications to function implementations.**

Given a RFC document and a protocol codebase, SPEC2CODE first performs preprocessing to extract structured specification requirements (SRs) and function-level code representations, along with contextual and dependency information. To ensure scalability, SPEC2CODE employs a two-stage process comprising relevance filtering and clustering-based SR organization to reduce the candidate pairs. For accuracy, SPEC2CODE performs fine-grained constraint-level matching on each candidate SR–function pair using LLMs, leveraging enriched context to determine whether a function fully, partially, or does not relate to an SR.

We evaluate SPEC2CODE on real-world implementations of HTTP, TLS and BFD protocols, including *Apache Httpd*, *Nginx*, *OpenSSL*, *BoringSSL*, *FRRouting*, and *BIRD*. Experimental results show that SPEC2CODE outperforms four state-of-the-art baselines, achieving up to 49%, 66%, and 66% improvement in precision, recall, and F1, respectively. Additionally, SPEC2CODE successfully recovers the mappings for 16 known inconsistency bugs and discovers 11 previously unreported inconsistencies using an integrated lightweight consistency verifier, 5 of which have been confirmed by project developers.

***Index Terms*—Software Verification, Protocol Compliance, Large Language Models.**

I. INTRODUCTION

Protocol specifications, particularly Request for Comments (RFCs), serve as formal blueprints that define the expected behaviors and constraints of networked systems, guiding their correct implementation. A program is considered correct with respect to its specification if it behaves as intended [1]. However, inconsistencies often arise when developers misinterpret or overlook normative constraints, leading to bugs that undermine correctness, security, and interoperability. For example, in Transport Layer Security (TLS) [2], a misplaced record-length check allowed oversized records to bypass alerts and caused denial-of-service risks [3], while in Hypertext Transfer Protocol (HTTP) [4], inconsistent parsing of headers has led to interoperability issues across implementations [5]. To address this, verification and validation techniques (e.g., testing) have been proposed to assess whether an implementation faithfully satisfies the given specifications.

Program verification and validation fundamentally rely on the availability of corresponding specifications for the programs being analyzed. In practice, due to the large size of protocol systems and scalability limitations of the verification algorithms, it is often infeasible to verify an entire implementation against its all RFC requirements. Instead, verification is commonly performed at a finer granularity, focusing on code snippets such as components or individual functions and their associated specifications. Unfortunately, explicit mapping between RFC-defined specifications and these code snippets may not be well established or systematically maintained in real-world projects [6]–[8]. As a result, protocol software verification and testing [9] often need manual effort, requiring developers or testers to analyze complex RFC documents and derive specific specifications for individual pieces of code, a process that is both time-consuming and labor-intensive.

Therefore, there is a strong need for an automated method capable of mapping fine-grained specifications derived from RFCs to their corresponding code snippets. Nonetheless, this task remains inherently difficult and error-prone [10]–[12], primarily due to the logical abstraction gap between natural-language specifications and low-level code implementations. To address this problem, existing research has mainly explored traceability link recovery (TLR) techniques.

Many TLR studies have explored automated techniques based on information retrieval (IR) [13]–[20], machine learning (ML) [21]–[25], and intermediate artifacts [26]–[31]. These approaches recover trace links by measuring similarity between specifications and code, ranging from lexical overlap to vector-space embeddings. Recent studies [20], [32], [33] have also explored the use of large language models (LLMs) and retrieval-augmented generation (RAG) to improve performance. However, existing TLR approaches predominantly rely on similarity-based matching (e.g., using vector space models, semantic indexing, or word embeddings) between specifications and code, which demands a high degree of semantic understanding to establish accurate mappings. This challenge becomes even more pronounced at finer levels of granularity, such as function-level mapping. Most prior work has been evaluated at the file or class level on Java datasets, which

* Corresponding author. Email: quanlili2022@163.com.

is relatively less difficult since files and classes often encapsulate coherent semantics and provide auxiliary signals (e.g., class names, attributes). In contrast, functions, especially in languages like C/C++, frequently involve low-level constructs such as macros, pointers, and cross-file dependencies, which require substantially deeper program understanding. Functions often share strong interdependencies and overlapping semantics, posing additional challenges for existing methods. Current TRL techniques are insufficient for capturing such semantic relevance. For example, the state-of-the-art approach LiSSA and FTLR achieve only 0.53 and 0.55 F1 score, even at the class level [19], [32], underscoring the difficulty of this problem.

Recent advancements in LLMs offer promising opportunities for fine-grained specification-to-code mapping, given their strong capabilities in natural language understanding [34], semantic reasoning [35], code comprehension [36] and synthesis [37]. However, applying LLMs to this task introduces several technical challenges related to both *accuracy* and *efficiency*. First, decomposing large-scale software and complex requirements into fine-grained units results in a vast number of potential code-specification pairs that need to be checked. Naively applying LLMs to evaluate every possible pair leads to prohibitively high computational costs. Second, without sufficient contextual information, LLMs may produce false positives when evaluating the relevance between a code snippet and a specification fragment. Third, the many-to-many nature of the mapping, where a single specification can correspond to multiple code units and vice versa, further increases the complexity of the task.

To bridge this gap and address the aforementioned challenges, we propose SPEC2CODE, a novel LLM-driven framework designed to identify the precise mapping of protocol specifications to their corresponding code implementations. Given a protocol system and its associated RFC document, SPEC2CODE first performs preprocessing to extract relevant specification constraints and function-level details, including contextual information and dependency relationships. To address scalability and efficiency concerns, SPEC2CODE applies a combination of relevance filtering and specification clustering techniques to significantly reduce the space of candidates, eliminating irrelevant functions and narrowing the relevant specifications for each function. Finally, to improve mapping accuracy, SPEC2CODE performs a fine-grained analysis on each pair of candidate SR-functions. By incorporating contextual and dependency information, it determines whether the function fully, partially, or does not implement the given specification, based on whether all, some, or none of the defined constraints are implemented.

We demonstrate the effectiveness of SPEC2CODE on three network protocols, specifically focusing on HTTP, TLS and Bidirectional Forwarding Detection (BFD) [38]. Due to the absence of existing benchmarks for function-level specification-to-code mapping, we manually analyzed the HTTP RFC and its corresponding implementation, *HTTPD*, to construct a ground-truth dataset comprising 221 specification-to-function pairs. This dataset serves as a new benchmark for the comprehensive evaluation of SPEC2CODE in mapping RFC-defined specifications to their function implementations.

For each protocol, we selected two distinct and widely adopted implementations: *HTTPD* and *Nginx* for HTTP, *OpenSSL* and *BoringSSL* for TLS, and *FRRouting* (FRR) and *BIRD Internet Routing Daemon* (BIRD) for BFD. Across these systems, SPEC2CODE successfully establishes thousands of fine-grained specification-to-code mappings (see Table II). To evaluate the effectiveness, we compare SPEC2CODE against two state-of-the-art TLR methods, LiSSA [32] and FTLR [19]. For broader comparison, we also include two representative code search techniques, DeepCS [39] and deGraphCS [40], as they represent an alternative for linking textual descriptions with code.

The experimental results show that SPEC2CODE significantly outperforms all baselines, achieving on average a 49%, 66%, and 66% increase in precision, recall, and F1-score, respectively. In addition, we evaluated SPEC2CODE on a set of 16 known bugs related to specification inconsistencies. We found that SPEC2CODE successfully identified the correct specification–function pairs for all 16 cases. To further assess its utility, we conduct a direct inconsistency checking procedure by prompting an LLM to analyze all candidate pairs identified by SPEC2CODE. After manual validation, we discovered 11 new inconsistencies, where 5 have been confirmed by project developers. These findings highlight the accuracy of SPEC2CODE and its practical utility in detecting bugs arising from specification–implementation inconsistencies.

In summary, this paper makes the following contributions:

- To the best of our knowledge, SPEC2CODE is the first approach for mapping protocol specifications from RFCs to function-level implementations.[1]
- We release a structured dataset of function-level specification-to-code mappings for the HTTP protocol, enabling rigorous evaluation and supporting future research.
- We apply SPEC2CODE to three widely used network protocols, HTTP, TLS and BFD, and conduct comprehensive evaluations across six real-world implementations, demonstrating the effectiveness and efficiency of our approach.
- We integrate a lightweight LLM-based consistency verifier, which identifies 11 unknown inconsistency bugs.
- Our code and data are available online [41].

II. PROBLEM DEFINITION

Definition 1 (Relevance and Implementation): Let s be a specification requirement (SR) and f a function. We say that f *implements* s if its logic satisfies the constraints (i.e., the triggering conditions) and/or performs the actions described in s. This implementation can be:

- *Full* or *Partial*: whether f covers all or only some aspects of s;

[1]While this paper mainly focuses on protocol mapping due to the availability of RFCs, our method can be easily generalized to other types of software specification–to–code mapping.

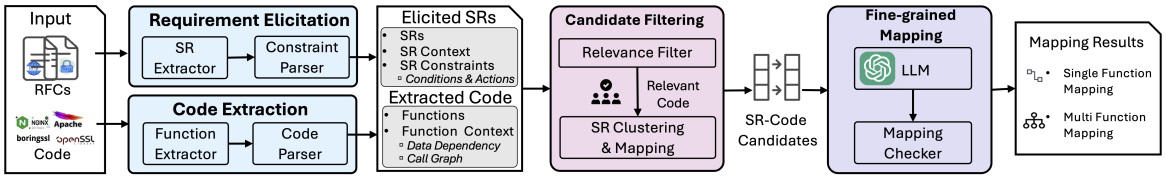


Fig. 1: Overview of Spec2Code

- *Correct* or *Incorrect*: whether the satisfied logic fully aligns with the semantics of s.

We say that an SR s is *relevant* to a function f if f implements any part of s (regardless of whether it is full/partial or correct/incorrect). Otherwise, f is *irrelevant* to s.

Definition 2 (Consistency and Inconsistency): Given a relevant pair (s, f): it is *consistent* if f fully and correctly implements all aspects of s; it is *inconsistent* if f only partially implements s, omits required logic, or performs unintended behavior contradicting s. Consistency is assessed only for relevant pairs; irrelevant functions need no check.

Definition 3 (Specification-to-Code Mapping): Given a software codebase $\mathcal{C} = \{c_1, \ldots, c_m\}$ and a specification requirement set $\mathcal{S} = \{s_1, \ldots, s_n\}$ (e.g., RFC documents), the goal is to automatically construct a mapping $\mathcal{M} \subseteq \mathcal{S} \times \mathcal{P}(\mathcal{C})$, where each pair $(s_i, C_i) \in \mathcal{M}$ indicates that the set of code elements $C_i \subseteq \mathcal{C}$ collectively implement the specification requirement s_i. Each code element $c_j \in \mathcal{C}$ typically corresponds to a code snippet (e.g., a function or code file), and $\mathcal{P}(\mathcal{C})$ denotes the power set of $\mathcal{C}$.

Note that, due to the varying granularity of both specification requirements (e.g., individual or multiple constraints) and code implementations (e.g., single or multiple functions), the mapping between them can naturally form a many-to-many relationship. Without loss of generality, we formulate the problem as a one-to-many mapping, where, given a single SR s_i, the goal is to identify the corresponding set of function-level code snippets. Fundamentally, a many-to-many relationship can be represented as a composition of multiple one-to-many mappings.

III. Related Work

A. Traceability Link Recovery (TLR)

TLR aims to identify and maintain links between software artifacts, such as requirements and source code. Early TLR approaches mainly rely on IR techniques, including vector space models [13], [14], latent semantic indexing [15], and latent dirichlet allocation [16], to measure textual similarity. To mitigate vocabulary mismatch, where different artifacts use distinct terms to refer to the same concept, some approaches [17], [18] enrich textual representations with biterms extracted from both requirements and code, while others [19], [20] compute similarity at fine-grained units (e.g., sentences, methods) and aggregate the scores to derive coarse-grained links.

Later works introduce machine learning models trained on manually annotated links, such as RNNs with word embeddings [21], feed-forward networks with cluster-pair rank models [22], ranking of word embeddings [23], active learning [24], and self-attention [25]. Instead of directly bridging the semantic gap, other approaches [26]–[31] leverage intermediate artifacts to establish transitive links. For example, the Connecting Links Method (CLM) [26], [31] relates two artifacts via a shared third artifact using IR similarity.

More recently, several studies [20], [32], [33] explore LLMs and RAG to improve the accuracy and generalizability of TLR. Despite these advances, existing methods remain restricted to file-level traceability and depend on preprocessed artifacts. In contrast, Spec2Code enables fine-grained, function-level mapping through end-to-end analysis of raw specifications and complete codebases.

B. Code Search

Code search aims to retrieve relevant functions from a natural language query. Existing work falls into two main types: IR-based and deep learning (DL)-based methods.

IR-based methods rely on lexical matching between query terms and code tokens [42]. They often use query expansion (e.g., WordNet synonyms [43]) or API enrichment (e.g., CodeHow [44]) to improve retrieval. While effective in simple cases, these methods cannot capture deeper semantic relations and depend heavily on the presence of relevant keywords.

DL-based methods embed code and queries into a shared semantic space. DeepCS [45], CodeSearchNet [46], and UNIF [47] use sequence models to represent code and queries as flat token sequences, but often overlook structure. To address this, MMAN [48], GraphSearchNet [39], and deGraphCS [40] adopt graph-based models that incorporate abstract syntax trees (ASTs) and data-flow graphs.

These approaches typically match functionality descriptions to code. In contrast, Spec2Code tackles a different task: mapping constraint-driven SRs, such as syntax rules or protocol behaviors from RFCs, to their implementations. Moreover, code search assumes one-to-one matching, whereas Spec2Code supports many-to-many mappings.

IV. Approach

A. Overview

Figure 1 presents an overview of Spec2Code, which consists of three main components: *Requirement Elicitation&Code Extraction*, *Candidate Filtering*, and *Fine-Grained Mapping*. The component *Requirement Elicitation&Code Extraction* extracts SRs and their contextual dependencies from the specification document. On the code side, it parses the source code to obtain function definitions along with their contextual information.

Algorithm 1: SPEC2CODE

```
Input   : Specification Requirement Document $\mathbf{S}$
          Software Codebase $\mathcal{C}$
Output  : Mapping $\mathcal{M}$
1  $\mathcal{M} \leftarrow \emptyset$
2  $PartialMap \leftarrow \{\}$ // Temporarily hold partial mappings
3  $SRs \leftarrow \text{Extract}_{SR}(\mathbf{S})$ // Extract SRs along with their contextual information and constraints
4  $F \leftarrow \text{Extract}_{C}(\mathcal{C})$ // Extract Functions, call graph and data dependencies
5  $F' \leftarrow \text{Filter}(F, \mathbf{S})$ // Filter irrelevant functions
6  $SRCluster \leftarrow \text{SemanticCluster}(SRs)$ // Semantic clustering of SRs
7  for $f \in F'$ do
8      $SR' \leftarrow \text{ClusterMap}(f, SRCluster)$ // Map a function to a cluster of SRs
9      for $s \in SR'$ do
10         $(Re, Cons) \leftarrow \text{SRMap}(f, s)$ // Map the function to a SR and returns the implemented constraints
11         if $Re$ is $Full$ then
12             $\mathcal{M} \leftarrow \mathcal{M} \cup \{(s, f)\}$
13         else if $Re = Partial$ then
14             $PartialMap[s] \leftarrow PartialMap[s] \cup \{(f, Cons)\}$
15 for $s \in keys(PartialMap)$ do
16     $FuncSet \leftarrow \{f \mid (f, Cons) \in PartialMap[s]\}$
17     $CombinedCons \leftarrow \bigcup\{Cons \mid (f, Cons) \in PartialMap[s]\}$
18     if $Verify(s, FuncSet, CombinedCons)$ then
19         $\mathcal{M} \leftarrow \mathcal{M} \cup \{(s, FuncSet)\}$
20 return $\mathcal{M}$
```

Subsequently, SPEC2CODE applies *Candidate Filtering* to narrow the large candidate space introduced by numerous SRs and functions. It first uses a *Relevance Filter* to eliminate functions that are clearly unrelated to all specifications. Then, in the *SR Clustering and Mapping* phase, it further narrows the candidate SRs for each function by mapping the function to a cluster of semantically related SRs.

Finally, in the *Fine-Grained Mapping* stage, we leverage an LLM to analyze each SR–code candidate pair and identify which constraints in the SR are implemented by the code. The model also classifies the mapping degree as *full*, *partial*, or *none*, reflecting whether all, some, or none of the constraints are covered. For *full* mappings, the SR–code pair is recorded as a single-function mapping. For *partial* mappings, we further aggregate relevant functions and verify whether they collectively implement all constraints specified in the SR.

Algorithm 1 presents the algorithm underlying our method. Lines 1–2 initialize the mapping set and a dictionary for recording partial mappings. At Line 3, all SRs are extracted from the given specification document. At Line 4, all functions and their associated contextual information are extracted from the codebase. Line 5 applies the relevance filter to remove functions that are clearly unrelated, based on the semantic summary of the specification. At Line 6, the SRs are clustered into semantically similar groups, which are then used at Line 8 to align each function with a relevant SR cluster. At Line 10, the algorithm checks the relevance between a function and each candidate SR in the assigned cluster, identifying the subset of constraints from the SR that are implemented by the function. For *full* mappings, the corresponding SR–function pair is added to the mapping set at Line 12. For partial mappings, the algorithm aggregates associated functions (Lines 16–17) and verifies whether the function group collectively implements all constraints in the SR (Line 18). If the verification succeeds, the group is added to the mapping set as a multi-function mapping (Line 19).

B. Requirement Elicitation&Code Extraction

This module automatically extracts specification requirements (SRs) from the RFC documents and function-level representations from the codebase, together with their contextual information. It forms the first stage of our approach and provides the structured inputs for subsequent candidate filtering and fine-grained mapping. The process consists of three sub-components: an SR Extractor, a Constraint Parser, and a Code Parser.

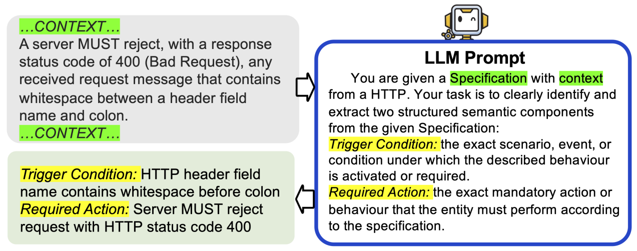
1) SR Extractor: Protocol specifications often use standardized terms such as `MUST`, `SHOULD`, and `MAY`, as defined in RFC 2119 [49]. In our approach, requirement elicitation is automated by detecting these modal keywords and segmenting the specification text into individual sentences, each treated as an SR. To support accurate mapping, we further enrich each SR with its surrounding context, including the containing paragraph and adjacent paragraphs. In addition, we extract normative internal references by automatically resolving cross-references in the RFC XML source.

2) Constraint Parser: Because normative sentences in RFCs typically express conditional obligations (e.g., "if *condition*, then *required behavior*"), we represent each SR as a set of *constraints* composed of triggering conditions and required actions. This abstraction directly reflects the linguistic patterns of RFCs and aligns with how implementations realize such logic (e.g., conditional checks and corresponding actions in code). These constraints may be implemented by different functions across the codebase. To bridge this granularity gap between abstract specifications and distributed implementations, we introduce a constraint parser that extracts fine-grained constraints from each SR, where each constraint comprising the *trigger condition* and *required action*. This enables more precise matching between specifications and their corresponding implementation functions.

An example of constraint parsing is shown in Figure 2. The LLM is guided by a carefully designed prompt comprising three elements: (1) task-specific instructions for extracting semantic constraints, (2) the target SR and its surrounding context, and (3) definitions of the *trigger condition* and *required action*. Based on this prompt, the LLM produces a structured output containing one or more trigger–action pairs that capture the operational semantics of the SR.

3) Code Parser: This component extracts function-level information and contextual metadata from the code. It begins by parsing the code into an AST using tools such as Clang and identifies all function definitions by traversing the tree.

To capture context information, the component constructs a function call graph. For direct calls, it establishes explicit

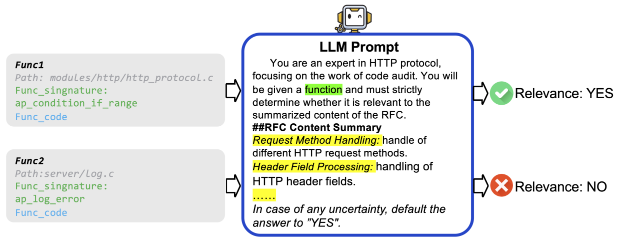


…CONTEXT…
A server MUST reject, with a response status code of 400 (Bad Request), any received request message that contains whitespace between a header field name and colon.
…CONTEXT…

Trigger Condition: HTTP header field name contains whitespace before colon
Required Action: Server MUST reject request with HTTP status code 400

LLM Prompt
You are given a Specification with context from a HTTP. Your task is to clearly identify and extract two structured semantic components from the given Specification:
Trigger Condition: the exact scenario, event, or condition under which the described behaviour is activated or required.
Required Action: the exact mandatory action or behaviour that the entity must perform according to the specification.

Fig. 2: An example of Parsing Constraint Specified in SR

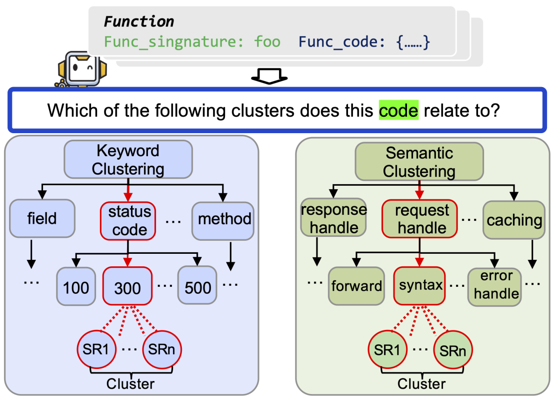


```
Func1
Path: modules/http/http_protocol.c
Func_singnature:
ap_condition_if_range
Func_code
```

```
Func2
Path:server/log.c
Func_singnature:
ap_log_error
Func_code
```

LLM Prompt
You are an expert in HTTP protocol, focusing on the work of code audit. You will be given a function and must strictly determine whether it is relevant to the summarized content of the RFC.
##RFC Content Summary
Request Method Handling: handle of different HTTP request methods.
Header Field Processing: handling of HTTP header fields.
……
In case of any uncertainty, default the answer to "YES".

Fig. 3: An Example of Relevance Filter



```
Function
Func_singnature: foo  Func_code: {……}
```

Which of the following clusters does this code relate to?

Fig. 4: An Example of SR Clustering and Mapping

caller–callee edges. For indirect calls, such as those made via function pointers, it conservatively links the call site to all functions with matching signatures.

Additionally, it analyzes each function body to extract data dependencies, including accesses to global variables and macro definitions. These elements often encode constants, conditions, or constraints, and are critical for constraint matching in the mapping process.

C. Relevance Filter

The *Data Preprocessing* stage typically extracts m SRs and n functions, resulting in a large search space of $m \times n$ potential SR–code pairs. Exhaustively evaluating all pairs with LLMs is computationally infeasible, as it would require $O(m \times n)$ inference calls.

To mitigate this, we introduce a *Relevance Filter* to eliminate functions that are unlikely to correspond to any SR.

As shown in Figure 3, it uses an LLM to assess the relevance of each function against the full set of SRs. The prompt includes four elements: a task description, function information (including path, signature, and code), a specification summary, and a structured output format.

Specification summary. Directly feeding all SRs into the prompt may exceed the input length limitations of LLMs. To address this challenge, we generate a hierarchical summary that captures the key semantics of the specification document. 1) Section-Level Summarization. We use an LLM to summarize each section of the document individually, extracting its main behaviors and underlying technical rationale. 2) Semantic Clustering. We then employ the LLM again to perform semantic clustering over the section summaries obtained in the first step. This process groups semantically related sections into high-level functional categories, yielding a concise and interpretable representation of the specification.

The LLM returns a binary relevance judgment (YES or NO) for each function, indicating whether it is likely to be relevant to any SR based on the specification summary.

D. SR Clustering and Mapping

Although the *Relevance Filter* effectively eliminates functions that are clearly irrelevant to all SRs, the remaining search space remains large because most functions are only associated with a small subset of SRs. To further reduce unnecessary comparisons, we introduce a clustering-based approach that maps each function to a semantically related SR cluster, thereby restricting its candidate SRs for subsequent analysis (i.e., the loop at Line 9-14 of Algorithm 1).

The core idea is to group related SRs into clusters by extracting and organizing their key features. This process involves two main steps: (1) extracting semantic features from the specification, and (2) mapping each SR to the appropriate feature-based cluster. For the subsequent mapping of code to specifications, we first align functions with the features associated with relevant SR clusters.

To support this, we propose constructing a Semantic Classification Tree (SCT) with a hierarchical, multi-level structure. Similar to the semantic summarization used in the *Relevance Filter* step, we adopt high-level functional categories as the first level of the SCT. At the second level, we define refined subcategories generated by an LLM, which analyzes individual sections of the specification to capture finer semantic distinctions. Each SR is then assigned to an appropriate subcategory (i.e., a leaf node in the tree) based on semantic relevance, resulting in clusters of contextually similar SRs. In this work, we use a two-level SCT structure for simplicity, though it can be further refined depending on the desired granularity. Due to the space limit, the detailed prompt for the semantic summarization is put on our website [41].

Keyword-based SCT. In addition, we observe that domain-specific knowledge can often be leveraged to construct an accurate SCT directly. For example, in many network protocol specifications, such as those in RFCs, well-defined keywords (e.g., Status Codes, HTTP Methods) are explicitly listed and directly reflect the core functions of the SRs. In such cases, an SCT can be constructed only using these keywords.

Specifically, we construct a keyword-based classification tree with two levels. The first level represents keyword categories derived from the document structure (e.g., Status Codes, Methods), and the second level contains specific identifiers under each category (e.g., `POST`, `GET`). An LLM assigns each SR to a leaf node by detecting whether it explicitly references the associated keyword. For example, Section 18 of RFC 9110 [50] defines structured terminology including status codes (e.g., 304, 400, 403), header fields (e.g., `Content-Length`, `If-Range`), and HTTP methods (e.g., `GET`, `POST`). We use these terms to build a two-level classification tree.

Note that, to mitigate potential inaccuracies in tree construction method, we can maintain two SCTs in parallel: one based on semantic summarization and one based on keyword extraction (when applicable). Both trees can be used together to identify relevant SR clusters for each given function.

Figure 4 illustrates examples of the two SCTs, on the left, a keyword-based tree, and on the right, a semantic-based tree. Each leaf node represents a cluster of SRs sharing closely related functional semantics. To narrow the candidate space for a given function, we apply a top-down LLM-guided matching process: the model first identifies the most relevant high-level category, then selects the most relevant subcategory (leaf node). These clusters serve as focused candidate sets of SRs for fine-grained constraint-level mapping. For instance, the function `foo` in Figure 4 is linked to clusters such as *Status Code 300* (from the keyword-based tree) and *Request Syntax* (from the semantic-based tree), which narrows down the candidate SRs for fine-grained constraint-level matching.

E. Fine-grained Mapping

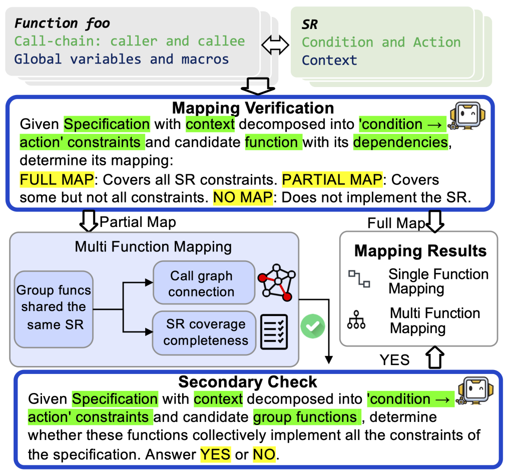
For each candidate SR-function pair, we perform a fine-grained relevance analysis using an LLM to assess whether the function f implements all or part of the SR. However, LLMs often produce false positives when lacking sufficient contextual information. To address this, we enrich each SR-function pair (s,f) with detailed contextual inputs:

1) For each SR s, we provide a succinct context that includes surrounding text and the parsed SR constraints, specifically, the *condition* and *action* components as derived in Section IV-B.
2) For each function f, we include the code snippet itself, a one-level call-chain context (i.e., its immediate callers and callees), and relevant dependencies such as global variables and macro definitions.

The LLM is prompted to identify which constraints in the SR are implemented by the given function. The LLM is also prompted to classify the mapping degree as *full*, *partial*, or *none*, reflecting whether all, some, or none of the constraints are implemented.

We define two types of mappings:

- **Single-function mapping (SFM):** A function f fully covers all constraints of a SR s.
- **Multi-function mapping (MFM):** The constraints of the SR s are distributed across multiple functions $\{f_1, f_2, \dots\}$, each implementing only part of the SR.



Given Specification with context decomposed into 'condition → action' constraints and candidate function with its dependencies, determine its mapping:
FULL MAP: Covers all SR constraints. PARTIAL MAP: Covers some but not all constraints. NO MAP: Does not implement the SR.

Given Specification with context decomposed into 'condition → action' constraints and candidate group functions , determine whether these functions collectively implement all the constraints of the specification. Answer YES or NO.

Fig. 5: The Process of Mapping Verification

As shown in Figure 5, any *full* mapping is recorded as a single-function mapping. For *partial* mapping, we explore whether a group of partially matched functions can collectively implement all constraints of a given SR, forming a multi-function mapping. For each SR, candidate multi-function groups are constructed by aggregating its associated partially matched functions.

Given that MFM is inherently more complex and prone to false positives, we introduce two heuristic-based verification methods to enhance its correctness.

- For each function group, we aggregate the SR constraints individually implemented by its member functions. If their union fully covers the target SR, we then check whether the functions form a connected subgraph in the call graph, indicating collaborative execution at runtime. Only groups satisfying both completeness and connectivity are considered as multi-function mapping candidates.
- For each candidate group that passes the previous step, we present the complete set of functions and the corresponding SR to the LLM for a secondary check. Specifically, the LLM is asked to assess whether the group collectively constitutes a full mapping. A group is accepted as a valid multi-function mapping only if the LLM determines that, together, these functions fully implement all constraints of the SR, with no part left unaddressed.

V. EXPERIMENTAL SETUP

A. Protocol Specification and Implementation

We evaluate SPEC2CODE using three widely adopted network protocols: HTTP/1.1, TLS 1.3 and BFD, as specified in RFCs 9110–9112 [50]–[52], RFC 8446 [53] and RFC 5880 [38], respectively.

For each protocol, we conduct fine-grained function-level mapping between the SRs and two representative real-world implementations. Specifically, we analyze *HTTPD* and *Nginx* for HTTP/1.1, *OpenSSL* and *BoringSSL* for TLS 1.3, and *FRR* and *BIRD* for BFD.

TABLE I: Inconsistencies Bugs Dataset

	#Bug	**CVE ID/Issue ID**
HTTPD	3	CVE-2016-8743 [54]; CVE-2020-13950 [55]; Bug_40232 [56]
Nginx	2	CVE-2013-2028 [57]; CVE-2013-4547 [58]
OpenSSL	7	issue_25402 [59]; issue_25309 [60]; issue_25086 [61]; issue_25041 [62]; issue_25040 [63]; issue_17934 [64]; issue_17948 [65]
FRR	4	issue_13085 [66]; issue_15697 [67]; issue_13432 [68]; issue_18588 [69]

B. Experimental Datasets

1) Benchmark Dataset: To address the absence of function-level benchmarks for specification-to-code mapping, we manually construct a ground-truth dataset aligning HTTP/1.1 requirements with their corresponding implementations in the *HTTPD* project. Due to the high cost of manual annotation, we focus on the `modules/http` directory, which encapsulates core HTTP logic and offers representative coverage of protocol-relevant functionalities.

Our approach automatically elicits 327 SRs and 127 functions for manual verification. Then we conduct a three-round labeling process: in each round, two experienced researchers independently annotate one-third of the pairs, resolving disagreements through discussion. Inter-rater agreement (Cohen's κ) improves from 0.55 to 0.67 and finally 0.83, indicating substantial agreement. The dataset ultimately contains 197 single-function and 24 multiple-function mappings.

2) Inconsistency Bugs Dataset: We construct an inconsistency bug dataset for six protocol implementations: *HTTPD*, *Nginx*, *OpenSSL*, *BoringSSL*, *FRR*, and *BIRD* by collecting *known* bugs from the National Vulnerability Database (NVD), GitHub issues, and official vulnerability disclosures.

From the NVD, we retrieve relevant Common Vulnerabilities and Exposures (CVE) entries by querying each implementation's name as a keyword. We then manually review each entry to determine whether it constitutes an inconsistency bug, specifically one caused by a violation of the corresponding RFC specification. We also manually review GitHub issue reports and official vulnerability disclosures to identify cases where the implementation deviates from normative specification requirements. Each confirmed case is recorded as a pair consisting of the violated SR and the corresponding vulnerable function, i.e., <SR, function> pairs.

Since both RFCs and implementations evolve, we trace each SR across revisions (e.g., RFC2616→RFC9110 for HTTP) to capture its latest formulation. Function alignment is done by exact name matching within each codebase. The resulting dataset includes 16 inconsistency bugs, as shown in Table I.

C. Baselines

As baselines, we select one state-of-the-art tool for TLR and two representative code search methods. 1) **LiSSA** [32], a recent state-of-the-art retrieval-augmented method that uses embedding similarity and LLM-based classification to recover traceability links between artifacts. 2) **FTLR** [19], an unsupervised fine-grained TLR approach that embeds requirements (sentence-level) and code (method-level) for link recovery without project-specific training. 3) **DeepCS** [39], a deep learning-based framework that jointly embeds code snippets and natural language descriptions into a unified vector space. 4) **deGraphCS** [40], a graph-based model that leverages gated graph neural networks with attention to learn accurate code representations for large-scale code retrieval.

D. Metrics

We evaluate the performance of SPEC2CODE and all baseline methods using four standard metrics: precision, recall, F1-score, and SR@k. The first three metrics follow standard formulations and are omitted for brevity. SR@k is a widely adopted metric in code search and retrieval tasks, measuring whether the top-k predicted functions successfully cover the ground-truth implementation of each SR.

To formally define SR@k, let G_s denote the set of ground-truth functions implementing SR s, and let $\text{Top}_k(s)$ denote the top-k functions predicted by the model. We define a per-SR indicator function *Hit*(s) based on SFM and MFM:

$$\text{Hit}(s) = \begin{cases} 1, & \text{if } s \text{ has an SFM and } G_s \cap \text{Top}_k(s) \neq \emptyset \\ 1, & \text{if } s \text{ has an MFM and } G_s \subseteq \text{Top}_k(s) \\ 0, & \text{otherwise} \end{cases}$$

For single-function mappings, Hit(s) returns 1 if the ground-truth function appears in the top-k predictions. For multi-function mappings, it returns 1 only if all ground-truth functions are all included in the top-k set.

The final SR@k score is computed as the average hit rate over all SRs $\mathcal{S}$:

$$\text{SR@}k = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \text{Hit}(s)$$

E. Implementation

Model Selection: The filtering task primarily requires coarse-grained semantic matching to quickly exclude irrelevant functions, making efficiency a priority. In contrast, the mapping verification task demands fine-grained semantic understanding and precise alignment between specification and code. To balance effectiveness and efficiency, we conducted a small-scale comparison of several models on both tasks (filtering: GPT-4o, GPT-4o-mini, DeepSeek-V3; verification: DeepSeek-R1, GPT-o3-mini). Considering the overall accuracy and resource cost, we finally chose *DeepSeek-V3* for efficient filtering, and *GPT-o3-mini* for more challenging mapping verification.

Parameter Setting: We set the temperature to 0 to ensure a deterministic generation. For all baselines, we adopt the best-performing configurations as reported in their original papers.

Running Environment: All experiments were conducted on a server with the following configuration: an AMD9654 processor, 256 GB of RAM, and 4 NVIDIA A6000 GPUs.

VI. EXPERIMENTAL RESULTS

In this section, we systematically evaluate the effectiveness and practical utility of SPEC2CODE by answering the following four research questions (RQs):

- **RQ1:** What are the characteristics of the mappings identified by Spec2Code?
- **RQ2:** How does the performance of Spec2Code compare with that of state-of-the-art baselines?
- **RQ3:** How does each key component contribute to the overall effectiveness of Spec2Code?
- **RQ4:** Can we use the generated mappings to detect new inconsistency?

A. RQ1: Characteristics of Identified Mappings

Table II summarizes the mappings identified by Spec2Code for the HTTP, TLS, and BFD protocols. We observe that the same set of SRs can lead to very different numbers of mappings across implementations. For instance, from 327 HTTP SRs, Spec2Code identifies 641 mappings in *HTTPD*, nearly twice the 332 in *Nginx*. For TLS, the numbers are 397 in *OpenSSL* versus 346 in *BoringSSL*, and for BFD, 188 in *FRR* versus only 92 in *BIRD*.

This discrepancy stems from one SR mapping to multiple independent functions. *HTTPD* and *OpenSSL* show such one-to-many mappings due to wrappers and compatibility layers, whereas *Nginx*, *BoringSSL*, and *BIRD* follow a leaner design, yielding fewer mappings per SR. Notably, this differs from multi-function mappings: one-to-many denotes independent functions implementing the same SR, while multi-function mappings involve functions collaboratively implementing it.

A higher number of mappings does not imply complete coverage. As shown in the coverage columns, *HTTPD* covers 263/327 SRs (80%) compared to 210/327 (64%) in *Nginx*; *OpenSSL* covers 247/410 SRs (60%) versus 212/410 (52%) in *BoringSSL*; and *FRR* covers 145/170 SRs (85%) versus 84/170 (49%) in *BIRD*.

Our analysis indicates three factors. At the *requirement level*, `MUST` requirements typically achieve higher precision than `SHOULD`/`MAY`, owing to clearer semantics, while `SHOULD`/`MAY` are implemented more selectively. Prohibitive `MUST NOT` clauses are satisfied by omission and thus often appear "uncovered". At the *module level*, projects may omit entire sections of the specification (e.g., HTTP `TRACE`, TLS 1.3 0-RTT, or many BFD extensions in BIRD). In such cases, all requirements in that section, including `MUST` ones, appear uncovered by construction. Additional gaps arise from implicit or delegated implementations (e.g., *HTTPD*'s reliance on APR routines), which static analysis cannot capture. Together, these factors explain coverage gaps, such as FRRouting vs. BIRD (85% vs. 49%), showing how both obligation levels and project-specific design choices shape coverage across implementations.

✎ **Answer to RQ1** SR-code mappings exhibit significant variability across implementations, shaped by the interplay between protocol specifications and factors like implementation strategies, modular design, and code visibility.

B. RQ2: Performance

To compare against baselines in terms of the performance, we treat each SR as a query for code search methods and retrieve a ranked list of candidate functions. For LiSSA, we adapt its classifier to our SR–function pairs and use its best-performing configuration, which omits SR preprocessing. We evaluate all methods using two complementary strategies: (1) a quantitative comparison on our benchmark dataset (see Section V-B1); and (2) manual validation of 60 randomly sampled SR–function mappings per protocol implementation.

Beyond these accuracy-oriented evaluations on the latest versions of the implementations, we further assess whether the recovered mappings can also cover previously reported inconsistency bugs using historical versions of the codebase where those bugs were originally observed. This dual perspective distinguishes accuracy on clean releases from bug coverage on vulnerable versions.

1) Benchmark-Based Evaluation: We first evaluate SR@1 for single-function mappings and SR@$|G_s|$ for multi-function mappings, where $|G_s|$ is the number of the ground-truth functions for SR s. We also compute precision, recall, and F1-score based on the top-1 prediction and top-$|G_s|$ predictions for single-function and multi-function mappings, respectively. Specifically, since LiSSA returns a ranked list of candidate functions for each SR, each with a classification label (YES or NO), we group the top-G_s candidates that are classified as YES as the candidate function set for MFM.

The results in Table III show that for *single-function mappings*, Spec2Code achieves an SR@1 of 0.95, at least 53% higher than all baselines (DeepCS 0.54, deGraphCS 0.55, LiSSA 0.62, FTLR 0.30). Spec2Code also achieves the highest precision (0.94), recall (0.98), and F1-score (0.96), consistently surpassing all baselines. For *multi-function mappings*, Spec2Code reaches an SR@$|G_s|$ of 0.96, while DeepCS and LiSSA achieve 0.17, deGraphCS 0.21, and FTLR 0.08. Precision (0.92), recall (0.96), and F1 (0.94) remain the highest with Spec2Code. These results demonstrate the superiority of Spec2Code across both mapping scenarios.

Note that in the multi-function mapping setting, all four metrics converge to the same value for each baseline. This is because each SR corresponds to a unique ground-truth function set, and evaluation is based on exact set matching. Any mismatch between the predicted and ground-truth sets results in both a false positive (an incorrect mapping present) and a false negative (a correct mapping missing). DeepCS and LiSSA yield identical metric values because, despite producing different predicted sets, they produce the same number of true positives, false positives, and false negatives per SR.

2) Manual Inspection: For each protocol, we randomly sample 50 single-function and 10 multi-function SRs from Spec2Code 's results. For single-function SRs, we compute SR@1. For multi-function SRs, we manually validate each mapping: if correct, the mapped functions serve as ground truth and we compute SR@$|G_s|$ for the baselines; if incorrect, we instead compute SR@5, since such mappings rarely

TABLE II: Spec–Code Mapping and Coverage Statistics.

Protocol	Impl	#Function	LOC	#Mapping (SFM/MFM)	#SR Covered	MUST	SHOULD	MAY
HTTP	HTTPD	7,491	942K	641 (573/68)	263/327 (0.90)	142/171 (0.94)	75/95 (0.88)	46/61 (0.83)
	Nginx	2,819	168K	332 (302/30)	210/327 (0.92)	117/171 (0.95)	58/95 (0.89)	35/61 (0.85)
TLS	OpenSSL	3,283	648K	397 (368/29)	247/410 (0.88)	177/306 (0.92)	40/60 (0.80)	30/44 (0.72)
	BoringSSL	2,285	197K	346 (322/24)	212/410 (0.89)	152/306 (0.93)	35/60 (0.82)	25/44 (0.75)
BFD	FRRouting	3,072	686K	188 (170/18)	145/170 (0.89)	87/100 (0.93)	38/45 (0.86)	20/25 (0.80)
	BIRD	218	36K	92 (84/8)	84/170 (0.87)	46/100 (0.91)	24/45 (0.83)	14/25 (0.78)

Note. Coverage values are reported as *Covered/Total (Precision)*. Per implementation, we used stratified random sampling with a per-implementation cap (up to 50). Samples were allocated proportionally to MUST/SHOULD/MAY; if a type had fewer items than its target, all of its mappings were inspected. Overall precision equals the coverage-weighted average of type-wise precision. *LOC denotes Lines of Code.*

TABLE III: Performance Comparison of Spec2Code and Baselines on Benchmark

Tool	Single-Function				Multi-Function			
	SR@1	Precision	Recall	F1	SR@$\|G_s\|$	Precision	Recall	F1
DeepCS	0.54	0.55	0.58	0.56	0.17	0.17	0.17	0.17
deGraphCS	0.55	0.57	0.59	0.58	0.21	0.21	0.21	0.21
LiSSA	0.62	0.63	0.34	0.44	0.17	0.17	0.17	0.17
FTLR	0.30	0.32	0.55	0.40	0.08	0.08	0.08	0.08
Spec2Code	**0.95**	0.94	0.98	0.96	**0.96**	0.92	0.96	0.94

TABLE IV: The Result of Manual Inspection for Single-Function (SFM) and Multi-Function (MFM) Mappings

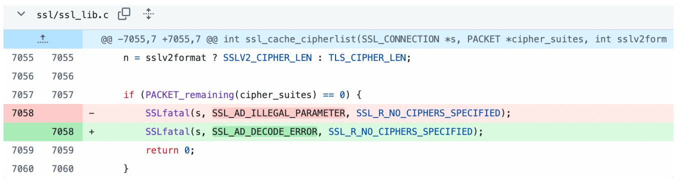
Impl.	**Spec2Code**		**LiSSA**		**FTLR**		**DeepCS**		**deGraphCS**	
	SFM	MFM	SFM	MFM	SFM	MFM	SFM	MFM	SFM	MFM
HTTPD	0.90	0.80	0.54	0.10	0.46	0.10	0.48	0.10	0.52	0.10
Nginx	0.92	0.90	0.50	0	0.48	0	0.44	0	0.56	0
OpenSSL	0.88	0.90	0.44	0.10	0.42	0.10	0.46	0	0.48	0.10
BoringSSL	0.90	0.80	0.50	0	0.38	0.10	0.42	0	0.42	0
FRR	0.86	0.80	0.40	0	0.36	0	0.40	0	0.38	0.10
BIRD	0.84	0.70	0.36	0.10	0.32	0	0.30	0	0.40	0
Average	0.88	0.82	0.46	0.05	0.40	0.05	0.42	0.02	0.46	0.05

involve more than five functions. The final score is the average over both SR types.

Table IV presents the results of our manual inspection. For single-function mappings, Spec2Code consistently outperforms all baselines across six implementations. On average, Spec2Code achieves an SR@1 of 0.88, about twice that of DeepCS (0.42) and FTLR (0.40), and substantially higher than LiSSA and deGraphCS (both around 0.46). In *HTTPD* and *Nginx*, Spec2Code reaches 0.90 and 0.92, while LiSSA achieves 0.54 and 0.50, both far lower. The trend holds in TLS: Spec2Code attains 0.88 on *OpenSSL* and 0.90 on *BoringSSL*, while LiSSA scores 0.44 and 0.50, respectively. For BFD, FRRouting achieves 0.86 and BIRD 0.84 with Spec2Code, whereas all baselines stay near or below 0.40.

For multi-function mappings, Spec2Code achieves an average SR@$|G_s|$ of 0.85, far surpassing DeepCS (0.03), deGraphCS (0.05), LiSSA (0.05) and FTLR (0.05). Across all implementations, Spec2Code maintains robust performance (0.80–0.90), whereas the baselines rarely exceed 0.10.

3) Coverage of Known Inconsistency Bugs: To complement the accuracy results on latest releases, we further evaluate Spec2Code on historical versions with known inconsistency bugs. Using our inconsistency bug dataset (see Section V-B2), which provides ground-truth <SR, function> pairs, we assess whether these buggy mappings are recovered. A bug is considered covered if the corresponding ground-truth pair appears in the mappings produced by Spec2Code for the same project. The number of covered bugs thus indicates the ability of Spec2Code to analyze real-world specification violations.


```
ssl/ssl_lib.c
@@ -7055,7 +7055,7 @@ int ssl_cache_cipherlist(SSL_CONNECTION *s, PACKET *cipher_suites, int sslv2form
      n = sslv2format ? SSLV2_CIPHER_LEN : TLS_CIPHER_LEN;

      if (PACKET_remaining(cipher_suites) == 0) {
-         SSLfatal(s, SSL_AD_ILLEGAL_PARAMETER, SSL_R_NO_CIPHERS_SPECIFIED);
+         SSLfatal(s, SSL_AD_DECODE_ERROR, SSL_R_NO_CIPHERS_SPECIFIED);
          return 0;
      }
```
Fig. 6: The Inconsistency Bug (#25309) in OpenSSL

The results show that all 16 bug pairs are successfully covered across *HTTPD*, *OpenSSL*, and *Nginx*, confirming its utility in recovering mappings relevant to real bugs. An example is shown in Figure 6, which illustrates bug #25309 [60] in OpenSSL. The function `ssl_cache_cipherlist` checks the length of the `cipher_suites` field and incorrectly returns an `illegal_parameter` alert when the length is zero. This violates RFC 8446, which states: *"Peers which receive a message ... containing an out-of-range length MUST terminate the connection with a decode_error alert,"* since a zero-length cipher suite list is out of range.

Answer to RQ2 Spec2Code significantly outperforms baselines in mapping specification requirements to code, and it also recovers the mappings of all 16 known inconsistency bugs, demonstrating both high accuracy and practical value for bug analysis.

C. RQ3: Ablation Study

We examine the role of each component in Spec2Code from two complementary angles: (*i*) a *system-level ablation* that removes modules and measures end-to-end performance and efficiency, and (*ii*) a *component-level validation* that assesses the intrinsic reliability of each LLM-driven step.

System-level Ablation: To quantify contribution, we evaluate three ablated variants and one pure-LLM baseline: (1) Spec2Code_NRF, which disables the relevance filtering module; (2) Spec2Code_NCP, which omits SR clustering; (3) Spec2Code_NFC, which removes function-level context during mapping verification; and (4) PureLLM, which directly

TABLE V: Performance comparison of SPEC2CODE and its variants on the HTTP benchmark dataset.

Tool	Precision	Recall	F1	# LLM Calls
SPEC2CODE	**0.94**	0.98	**0.96**	**4,023**
SPEC2CODE_NRF	0.91	0.98	0.94	6,057
SPEC2CODE_NCP	0.87	**1.00**	0.93	34,695
SPEC2CODE_NFC	0.89	0.95	0.92	4,023
PURELLM	0.88	0.96	0.92	70,959

Variants: NRF = w/o Relevance Filter; NCP = w/o SR Clustering; NFC = w/o Function-level Context; PureLLM = direct end-to-end mapping by LLM without filtering/clustering.

applies the Mapping Verification prompt to perform end-to-end mapping with using gpt-o3-mini.

Table V summarizes the results. Disabling the relevance filter (SPEC2CODE_NRF) increases LLM calls from 4,023 to 6,057 (a 50% increase) while only slightly reducing F1 from 0.96 to 0.94, indicating that some irrelevant pairs are pruned by the filter but the verification step is still capable of rejecting many false candidates. In contrast, removing SR clustering (SPEC2CODE_NCP) causes LLM calls to surge to 34,695 (8.6×) with only a 0.03 F1 drop, showing that clustering is the dominant lever for efficiency at scale. Eliminating function-level context (SPEC2CODE_NFC) reduces precision from 0.94 to 0.89 and F1 from 0.96 to 0.92, evidencing that contextual information is critical for accurate judgement during mapping verification. Finally, the pure LLM baseline (PURELLM) attains lower F1 (0.92) at a substantially higher cost (70,959 calls), underscoring the necessity of SPEC2CODE's filtering and structure-aware design. Without filtering, a large number of irrelevant SR–function pairs are passed to the LLM, introducing noise and leading to more false positives.

Component-level Validation: Beyond system-level ablations, we assess the intrinsic reliability of individual LLM-driven steps in SPEC2CODE. We evaluate: (1) the *Constraint Parser*, which decomposes SRs into structured trigger–action constraints; (2) the *Relevance Filter*, which removes functions unrelated to all SRs; (3) the *Semantic Classification*, which assigns SRs to clusters in the Semantic Classification Tree; and (4) the *Mapping Verification*, which determines whether the candidate function(s) implements a given SR.

For the Constraint Parser and Semantic Classification, we randomly sampled 50 SRs each and checked correctness of the structured decomposition and cluster assignment, respectively. For the Relevance Filter, we measured recall against the benchmark to ensure that no gold mappings were discarded. For Mapping Verification, we evaluated 100 benchmarked SR–function pairs and reported precision, recall, and F1.

Table VI summarizes the outcomes. All components show high reliability ($\geq$0.94), with only one SR misclassified during semantic clustering. Although LLMs may occasionally hallucinate, their impact on the pipeline is negligible. Together with the system-level ablation, these results confirm that SPEC2CODE's efficiency stems from effective pruning and clustering, while its accuracy benefits from reliable component behavior and function-level context during verification.

TABLE VI: Component-level validation of LLM-driven steps in SPEC2CODE.

Component	Metric (#Sample)	Result
Constraint Parser	Accuracy (50)	0.98
Relevance Filter	Recall (100)	1.00
Semantic Classification	Accuracy (50)	0.98
Mapping Verification	P / R / F1 (100)	0.94 / 1.00 / 0.97

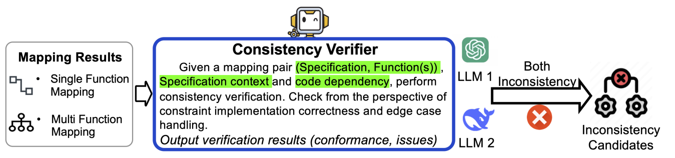

Mapping Results
Single Function Mapping
Multi Function Mapping
Consistency Verifier
Given a mapping pair (Specification, Function(s)), Specification context and code dependency, perform consistency verification. Check from the perspective of constraint implementation correctness and edge case handling.
Output verification results (conformance, issues)
LLM 1
LLM 2
Both Inconsistency
Inconsistency Candidates

Fig. 7: Example of Inconsistency Detection

Answer to RQ3 Relevance filtering and SR clustering are key to SPEC2CODE's efficiency, while function-level context is crucial for its LLM-based mapping accuracy. Component-level validation further shows that all LLM-driven steps are highly reliable, with only rare hallucinations, reinforcing the robustness of the pipeline.

D. RQ4: Inconsistency Detection

Accurate mappings are the foundation for detecting specification–implementation inconsistencies. Once relevant SR–function pairs are established, the next step is to examine whether these pairs faithfully implement the intended semantics. As defined in Definition 2, inconsistency arises when the mapped function only partially implements an SR, omits required logic, or contradicts the specified behavior. Thus, inconsistency detection builds directly upon the recovered mappings: mapping provides *where* to check, while inconsistency detection evaluates *how well* the implementation aligns with the specification.

Concretely, we use two complementary LLMs—GPT-o3-mini and DeepSeek-R1 (Fig. 7). Each independently evaluates every SR–function mapping with the same inputs (SR text with context, relevant code dependencies, and call-graph information) and a shared rubric on constraint adherence and corner-case handling. Decisions are aggregated by majority vote; with two models, this means a mapping is flagged inconsistent only if both judge it non-compliant. Detailed prompts and rubrics are available online [41].

Across the six protocol implementations, our checker flagged 56 potential inconsistencies: 22 in *HTTPD*, 11 in *Nginx*, 8 in *OpenSSL*, 4 in *BoringSSL*, 8 in *FRR*, and 3 in *BIRD*. Manual validation confirmed 15 true inconsistencies (precision ≈27%), among which 11 were previously unknown, and 5 of these 11 new bugs were acknowledged by the project developers. Although the checker produces false positives, these results show that, given only the recovered mappings, LLM-based analysis can uncover specification–implementation violations even in mature implementations. This highlights the practical utility of SPEC2CODE for assisting bug discovery.

The following case studies illustrate how recovered mappings first establish relevance between SRs and functions, and how inconsistency detection then exposes concrete specification violations.

Case Study 1 (Missing or Misplaced Condition Checks). RFC8446 requires a `record_overflow` alert when the size of `TLSInnerPlaintext+1` exceeds 2^{14} bytes. Our mapping linked this SR to the function `tls13_post_process_record` in OpenSSL, which performs this check only after stripping padding, violating the RFC and allowing oversized records to bypass alerts.

Case Study 2 (Violation of Error Handling Behavior). In OpenSSL, the function `tls_parse_compress_certificate` returns success when the input packet contains an extraneous trailing byte. However, the specification mandates that any message that cannot be fully parsed must trigger a fatal `decode_error` alert. This missing alert handling indicates a deviation from the specification's mandated behavior.

Answer to RQ4 Given the detected mappings of SPEC2CODE, a basic LLM was able to discover 11 new inconsistencies, highlighting the significance of accurate SR-code mappings in bug detection.

VII. DISCUSSION

A. Lesson Learned

Improving mapping quality. While SPEC2CODE achieves strong precision on explicit function-level logic, requirements enforced *implicitly* (e.g., `MUST NOT` clauses satisfied by omission) or delegated to *external libraries* remain hard to trace. Such cases may appear "uncovered" even when the implementation is correct. Future work could extend SPEC2CODE with external dependency analysis and negative-case reasoning to better cover such requirements.

LLMs show promise for inconsistency detection but need refinement. Our checker uncovered 15 true inconsistencies, including 11 previously unknown issues, showing the potential of LLM-based reasoning. However, the high false positive rate suggests that dedicated techniques, such as constraint reasoning or lightweight formal verification, are still needed to complement LLMs and improve reliability.

Recommendations for Protocol Developers. Our observations suggest three practices that could enhance clarity and maintainability. First, clarify unsupported or selectively implemented requirements (e.g., optional features, delegated checks, or omitted `SHOULD`/`MAY`) so omissions are not misinterpreted as bugs. Second, summarize compliance boundaries in release notes or documentation; coverage differences across implementations (e.g., FRRouting vs. BIRD) highlight trade-offs that should be communicated clearly. Third, mitigate risks from redundant implementations: when one SR is realized by multiple functions, developers should consolidate logic where possible, or otherwise use documentation or tooling to link parallel implementations and ensure consistent updates.

TABLE VII: Runtime and cost of applying Spec2Code across implementations.

	HTTPD	Nginx	OpenSSL	BoringSSL	FRRouting	BIRD
Runtime (H)	3.1	1.1	1.3	0.9	1.4	0.4
Cost (USD)	20.3	7.5	8.8	6.1	9.6	2.5

B. Cost and Scalability

Table VII summarizes runtime and API cost across all implementations. All runs finish within 3.1 hours and under $20.3, with small systems such as BIRD costing only $2.5. Thus, SPEC2CODE is both feasible and affordable for large protocol codebases. By contrast, a pure LLM approach would take over 1 day (due to the API rate limit) and cost over $1,000, making it impractical at scale.

C. Threats to Validity

Our study faces internal and external threats to validity. Internally, benchmark construction and mapping evaluation may introduce bias. To mitigate this, two researchers independently performed the tasks and resolved discrepancies through discussion. LLM use also poses randomness risks, which we addressed by using 0 temperature for deterministic responses.

Externally, our evaluation focused solely on protocol implementations, which may limit generalizability. Future work will explore applying SPEC2CODE to other domains with different specification styles. Finally, although we manually built a benchmark dataset and performed additional validation, residual labeling inaccuracies remain possible.

VIII. CONCLUSION

In this work, we present SPEC2CODE, the first LLM-based framework for automating mappings from natural language specifications to function-level code implementations. By combining scalable relevance filtering and clustering with context-aware LLM-based verification, SPEC2CODE effectively bridges the gap between protocol standards and real-world code. Our evaluation on HTTP, TLS, and BFD implementations shows that SPEC2CODE significantly outperforms all baselines and enables precise recovery of known inconsistency bugs. These results highlight SPEC2CODE's potential as a practical foundation for automated specification compliance and software correctness checking.

IX. ACKNOWLEDGMENT

This research is supported by the National Research Foundation, Singapore, the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN) and the Lee Kong Chian Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] V. S. Alagar and K. Periyasamy, *Specification of software systems*. Springer Science & Business Media, 2011.

[2] S. Turner, "Transport layer security," *IEEE Internet Computing*, vol. 18, no. 6, pp. 60–63, 2014.

[3] "Cve-2022-38153," 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-38153

[4] R. Fielding and J. Reschke, "Hypertext transfer protocol (http/1.1): Message syntax and routing," Tech. Rep., 2014.

[5] "Cve-2025-6442," 2025. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-6442

[6] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. Zheng, "Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 1–13.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets," in *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, 2005, pp. 265–276.

[8] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: Multiple host ambiguities in http implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1516–1527.

[9] L. Quan, Q. Guo, H. Chen, X. Xie, X. Li, Y. Liu, and J. Hu, "Sadt: syntax-aware differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 524–535.

[10] J. Cleland-Huang, O. C. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions," in *Future of software engineering proceedings*, 2014, pp. 55–69.

[11] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang, "Strategic traceability for safety-critical projects," *IEEE software*, vol. 30, no. 3, pp. 58–66, 2013.

[12] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery," in *2012 20th IEEE international conference on program comprehension (ICPC)*. IEEE, 2012, pp. 183–192.

[13] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE transactions on software engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[14] A. Mahmoud, "An information theoretic approach for extracting and tracing non-functional requirements," in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*. IEEE, 2015, pp. 36–45.

[15] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 125–135.

[16] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE international conference on Software Engineering-Volume 1*, 2010, pp. 95–104.

[17] H. Gao, H. Kuang, K. Sun, X. Ma, A. Egyed, P. Mäder, G. Rong, D. Shao, and H. Zhang, "Using consensual biterms from text structures of requirements and code to improve ir-based traceability recovery," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–1.

[18] H. Gao, H. Kuang, W. K. Assunção, C. Mayr-Dorn, G. Rong, H. Zhang, X. Ma, and A. Egyed, "Triad: Automated traceability recovery based on biterm-enhanced deduction of transitive links among artifacts," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[19] T. Hey, F. Chen, S. Weigelt, and W. F. Tichy, "Improving traceability link recovery using fine-grained requirements-to-code relations," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 12–22.

[20] T. Hey, J. Keim, and S. Corallo, "Requirements classification for traceability link recovery," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 2024, pp. 155–167.

[21] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 3–14.

[22] W. Wang, N. Niu, H. Liu, and Z. Niu, "Enhancing automated requirements traceability by resolving polysemy," in *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 2018, pp. 40–51.

[23] T. Zhao, Q. Cao, and Q. Sun, "An improved approach to traceability recovery based on word embeddings," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 81–89.

[24] C. Mills, J. Escobar-Avila, A. Bhattacharya, G. Kondyukov, S. Chakraborty, and S. Haiduc, "Tracing with less data: active learning for classification-based traceability link recovery," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 103–113.

[25] M. Zhang, C. Tao, H. Guo, and Z. Huang, "Recovering semantic traceability between requirements and source code using feature representation techniques," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 873–882.

[26] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe, "Recovering transitive traceability links among software artifacts," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 576–580.

[27] A. D. Rodriguez, J. Cleland-Huang, and D. Falessi, "Leveraging intermediate artifacts to improve automated trace link retrieval," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 81–92.

[28] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "When and how using structural information to improve ir-based traceability recovery," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 199–208.

[29] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lü, and A. Egyed, "Can method data dependencies support the assessment of traceability between requirements and source code?" *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 838–866, 2015.

[30] K. Moran, D. N. Palacio, C. Bernal-Cárdenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson, "Improving the effectiveness of traceability link recovery using hierarchical bayesian networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 873–885.

[31] R. Tsuchiya, K. Nishikawa, H. Washizaki, Y. Fukazawa, Y. Shinohara, K. Oshima, and R. Mibe, "Recovering transitive traceability links among various software artifacts for developers," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 9, pp. 1750–1760, 2019.

[32] D. Fuchß, T. Hey, J. Keim, H. Liu, N. Ewald, T. Thirolf, and A. Koziolek, "Lissa: Toward generic traceability link recovery through retrieval-augmented generation," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering. ICSE*, vol. 25, 2025.

[33] J. Keim, S. Corallo, D. Fuchß, T. Hey, T. Telge, and A. Koziolek, "Recovering trace links between software documentation and code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[34] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[35] Q. Guo, X. Xie, S. Liu, M. Hu, X. Li, and L. Bu, "Intention is all you need: Refining your code from your intention," *arXiv preprint arXiv:2502.08172*, 2025.

[36] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu, "Specgen: Automated generation of formal program specifications via large language models," *arXiv preprint arXiv:2401.08807*, 2024.

[37] Q. Guo, X. Li, X. Xie, S. Liu, Z. Tang, R. Feng, J. Wang, J. Ge, and L. Bu, "Ft2ra: A fine-tuning-inspired approach to retrieval-augmented code completion," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 313–324.

[38] D. Katz and D. Ward, "Bidirectional forwarding detection (bfd)," Tech. Rep., 2010.

[39] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 933–944.

[40] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, L. Bai, W. Dong, and X. Liao, "degraphcs: Embedding variable-based flow graph for neural code search," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–27, 2023.

[41] "Spec2code: Code and data archive," https://sites.google.com/view/spec2code/.

[42] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[43] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[44] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270.

[45] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 631–642.

[46] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[47] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[48] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.

[49] S. Bradner, "Rfc2119: Key words for use in rfcs to indicate requirement levels," 1997.

[50] R. Fielding, M. Nottingham, and J. Reschke, "RFC 9110: HTTP Semantics," Internet Engineering Task Force (IETF), Request for Comments RFC 9110, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9110

[51] R. Fielding, M. Nottingham, and J. Reschke, "RFC 9111: HTTP Caching," Internet Engineering Task Force (IETF), Request for Comments RFC 9111, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9111

[52] R. Fielding, M. Nottingham, and J. Reschke, "RFC 9112: HTTP/1.1," Internet Engineering Task Force (IETF), Request for Comments RFC 9112, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9112

[53] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Tech. Rep., 2018.

[54] "Cve-2016-8743 detail," 2025. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2016-8743

[55] "Cve-2020-13950 detail," 2025. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2020-13950

[56] A. HTTPD, "Bug 40232 - date header replaced by apache's server date," 2025. [Online]. Available: https://bz.apache.org/bugzilla/show_bug.cgi?id=40232

[57] "Cve-2013-2028 detail," 2025. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2013-2028

[58] "Cve-2013-4547 detail," 2025. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2013-4547

[59] "Publicly invalid ffdhe and ecdhe key shares are rejected with internal_error alert," 2024. [Online]. Available: https://github.com/openssl/openssl/issues/25402

[60] "Incorrect alert description for clienthello with no cipher_suites," 2024. [Online]. Available: https://github.com/openssl/openssl/issues/25309

[61] "Receives any other change_cipher_spec value must abort the handshake with an unexpected_message alert in tls1.3," 2024. [Online]. Available: https://github.com/openssl/openssl/issues/25086

[62] "Tls1.3 client does not reject hrr without supported_versions field," 2024. [Online]. Available: https://github.com/openssl/openssl/issues/25041

[63] "In tls1.3, the client provides the psk_dhe_ke mode. when the server selects the psk_ke mode, the alert description is not illegal_parameter," 2024. [Online]. Available: https://github.com/openssl/openssl/issues/25040

[64] "Client allows more than one helloretryrequest," 2024. [Online]. Available: https://github.com/openssl/openssl/issues/17934

[65] "new_session_ticket message may contain invalid ticket_lifetime when using tlsv1.3," 2024. [Online]. Available: https://github.com/openssl/openssl/issues/17948

[66] "Bfd: Missing authentication section in bfd control packet," 2023. [Online]. Available: https://github.com/FRRouting/frr/issues/13085

[67] "bfdd: Add check for flag multipoint (m)," 2024. [Online]. Available: https://github.com/FRRouting/frr/issues/15697

[68] "bfdd: Miss check whether (p) and final (f) bits are both set," 2023. [Online]. Available: https://github.com/FRRouting/frr/issues/13432

[69] "bfdd: Initialize desiredmintxinterval to the required 1 second minimum," 2024. [Online]. Available: https://github.com/FRRouting/frr/issues/18588