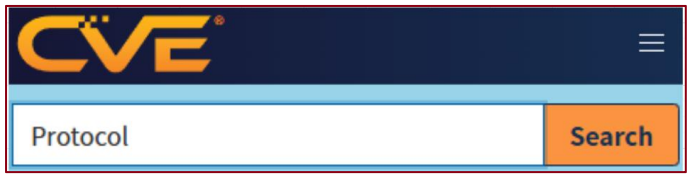
ProtocolGuard: Detecting Protocol Non-compliance Bugs via LLM-guided Static Analysis and Dynamic Verification

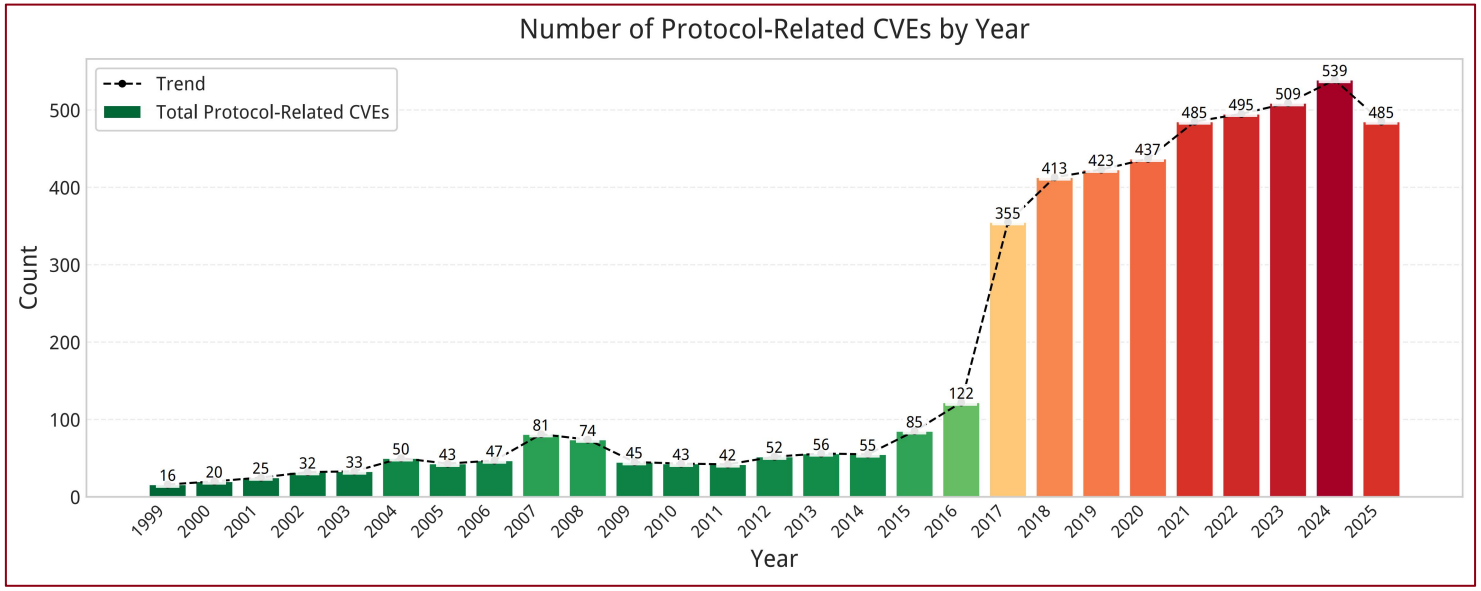
Xiangpu Song[1], Longjia Pei[1], Jianliang Wu[2], Yingpei Zeng[3], Gaoshuo He[1], Chaoshun Zuo[4], Xiaofeng Liu[1], Qingchuan Zhao[5] and Shanqing Guo[167]

[1]School of Cyber Science and Technology, Shandong University
[2]Simon Fraser University, [3]Hangzhou Dianzi University, [4]Independent Researcher, [5]City University of Hong Kong
[6]Shandong Key Laboratory of Artificial Intelligence Security,
[7]State Key Laboratory of Cryptography and Digital Economy Security, Shandong University

Network Protocols Under Threat

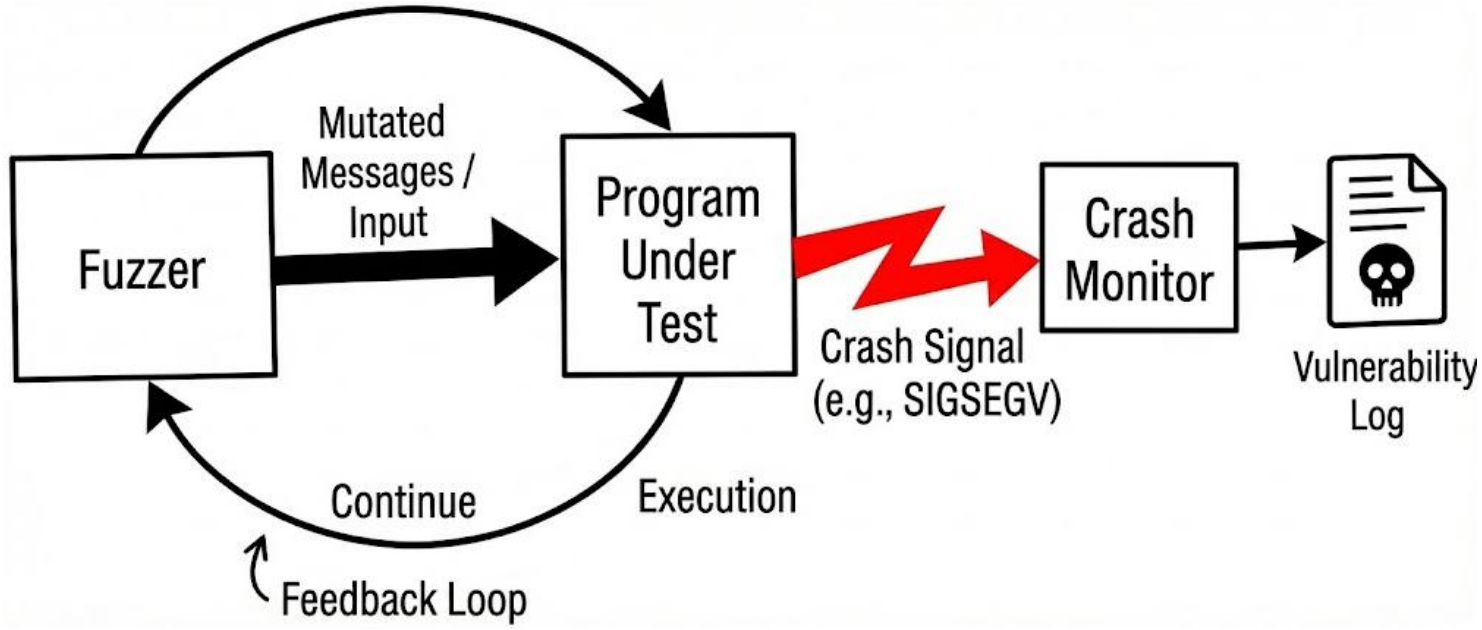
- Network protocols underpin modern critical infrastructure (e.g., IoT, and 5G networks).
- A single flaw in the protocol implementation may lead to serious problems.
- Vulnerabilities in protocol implementations are widespread and continue to be reported over time.





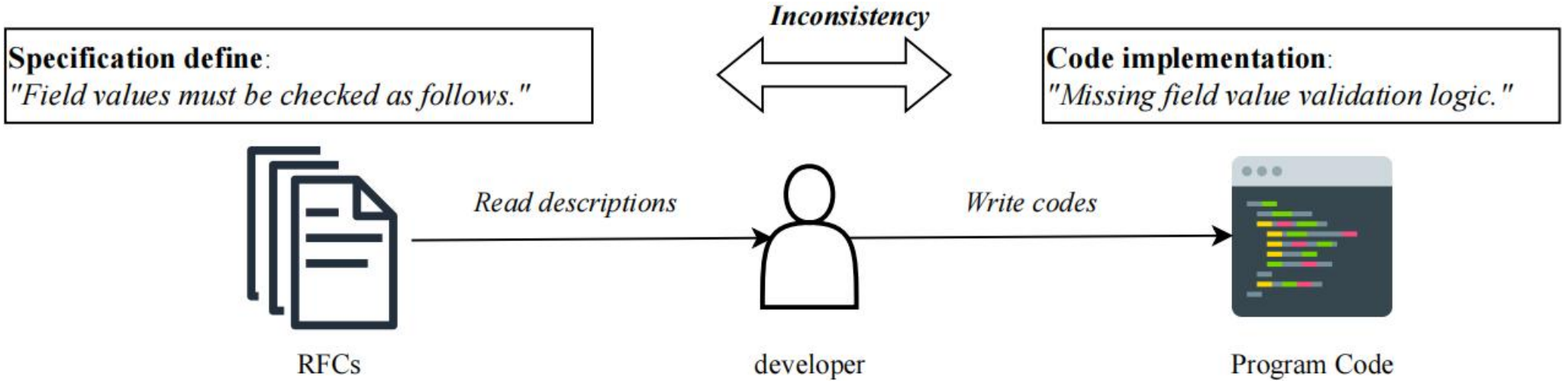
A Silent but Serious Issue: Protocol Non-compliance Bugs

- Many efforts have been made to detect protocol vulnerabilities, e.g., fuzzing.
- Most detected vulnerabilities are memory-related, with explicit signals (e.g., crashes).



A Silent but Serious Issue: Protocol Non-compliance Bugs

- Many efforts have been made to detect protocol vulnerabilities, e.g., fuzzing.
- Most detected vulnerabilities are memory-related, with explicit signals (e.g., crashes).
- Protocol non-compliance bugs are semantic and often silent, making them easy to overlook.
- These bugs stem from inconsistencies between protocol specifications (e.g., RFCs) and implementations.



A Silent but Serious Issue: Protocol Non-compliance Bugs

- Many efforts have been made to detect protocol vulnerabilities, e.g., fuzzing.
- Most detected vulnerabilities are memory-related, with explicit signals (e.g., crashes).
- Protocol non-compliance bugs are semantic and often silent, making them easy to overlook.
- These bugs stem from inconsistencies between protocol specifications (e.g., RFCs) and implementations.
- Such bugs are extremely common in the real world, but their consequences may be dangerous: ranging from service disruptions to security vulnerabilities.

Detecting protocol non-compliance bugs has become imperative.

Motivating Case: ClientId Truncation in a MQTT broker

- **Bug Description**: This bug stems from the broker copying the client identifier into a fixed-size buffer without validating its length, causing silent truncation of oversized client IDs.
- **Bug Impact**: By exploiting silent truncation, an attacker can cause client ID collisions and impersonate legitimate clients, leading to DoS attacks.

MQTT Specification v3.1.1
The Server MUST allow ClientIds which are between 1 and 23 UTF-8 encoded bytes in length, and MAY allow ClientIds that contain more than 23 encoded bytes.

```
1  struct client {
2    ...
3    char client_id[MQTT_CLIENT_ID_LEN];
4  }
5  static int connect_handler(struct io_event *e) {
6    struct mqtt_connect *c = &e->data.connect;
7    struct client      *cc = e->client;
8
9    // BUG: Client ID truncation without validation
10   snprintf(cc->client_id, MQTT_CLIENT_ID_LEN, "%s
         ", c->payload.client_id);
11   ...
12 }
```

Bug

Motivating Case: ClientId Truncation in a MQTT broker

- **Bug Description**: This bug stems from the broker copying the client identifier into a fixed-size buffer without validating its length, causing silent truncation of oversized client IDs.
- **Bug Impact**: By exploiting silent truncation, an attacker can cause client ID collisions and impersonate legitimate clients, leading to DoS attacks.

MQTT Specification v3.1.1
The Server MUST allow ClientIds which are between 1 and 23 UTF-8 encoded bytes in length, and MAY allow ClientIds that contain more than 23 encoded bytes.

The code violates an implicit protocol semantic: ClientIds must uniquely and consistently identify clients.

```
1  struct client {
2    ...
3    char client_id[MQTT_CLIENT_ID_LEN];
4  }
5  static int connect_handler(struct io_event *e) {
6    struct mqtt_connect *c = &e->data.connect;
7    struct client      *cc = e->client;
8
9    // BUG: Client ID truncation without validation
10   snprintf(cc->client_id, MQTT_CLIENT_ID_LEN, "%s
         ", c->payload.client_id);
11   ...
12 }
```

Limitations of Existing Work

- **Fuzzing based on memory sanitizers.**

  Relies on crashes or sanitizer signals, and fails to detect such bugs that do not trigger explicit failures.

Limitations of Existing Work

- **Fuzzing based on memory sanitizers.**

  Relies on crashes or sanitizer signals, and fails to detect such bugs that do not trigger explicit failures.

- **Differential testing.**

  Compares responses across implementations, but fails in this case because both vulnerable and correct brokers return the identical CONNACK response, making the bug behavior indistinguishable.

Limitations of Existing Work

- **Fuzzing based on memory sanitizers.**

  Relies on crashes or sanitizer signals, and fails to detect such bugs that do not trigger explicit failures.

- **Differential testing.**

  Compares responses across implementations, but fails in this case because both vulnerable and correct brokers return the identical CONNACK response, making the bug behavior indistinguishable.

- **Heuristic-based static detection.**

  Depends on predefined rules extracted from specifications, but lacks the semantic understanding needed to assess whether implementation logic truly preserves protocol invariants.

Limitations of Existing Work

- **Fuzzing based on memory sanitizers.**

  Relies on crashes or sanitizer signals, and fails to detect such bugs that do not trigger explicit failures.

- **Differential testing.**

  Compares responses across implementations, but fails in this case because both vulnerable and correct brokers return the identical CONNACK response, making the bug behavior indistinguishable.

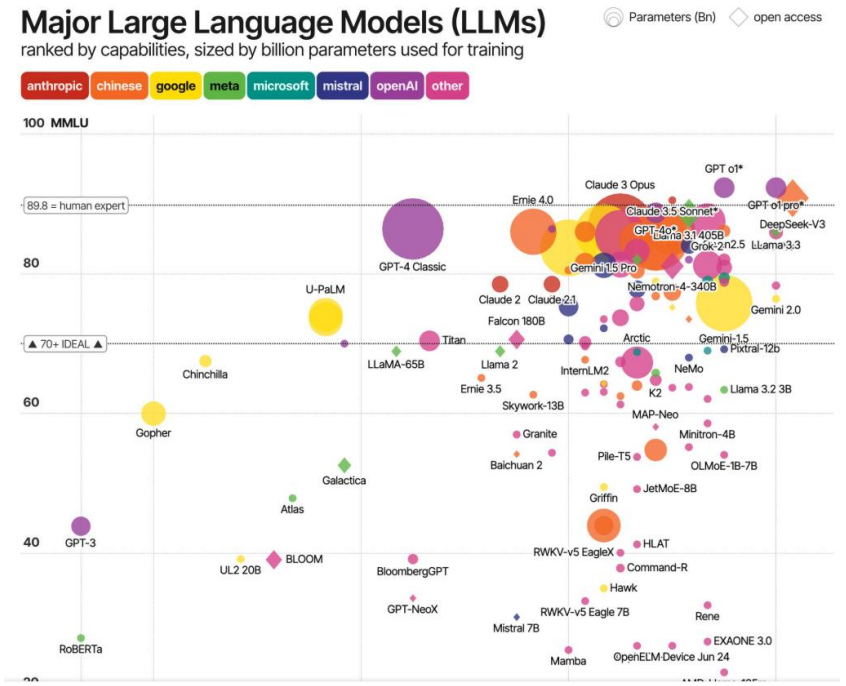
- **Heuristic-based static detection.**

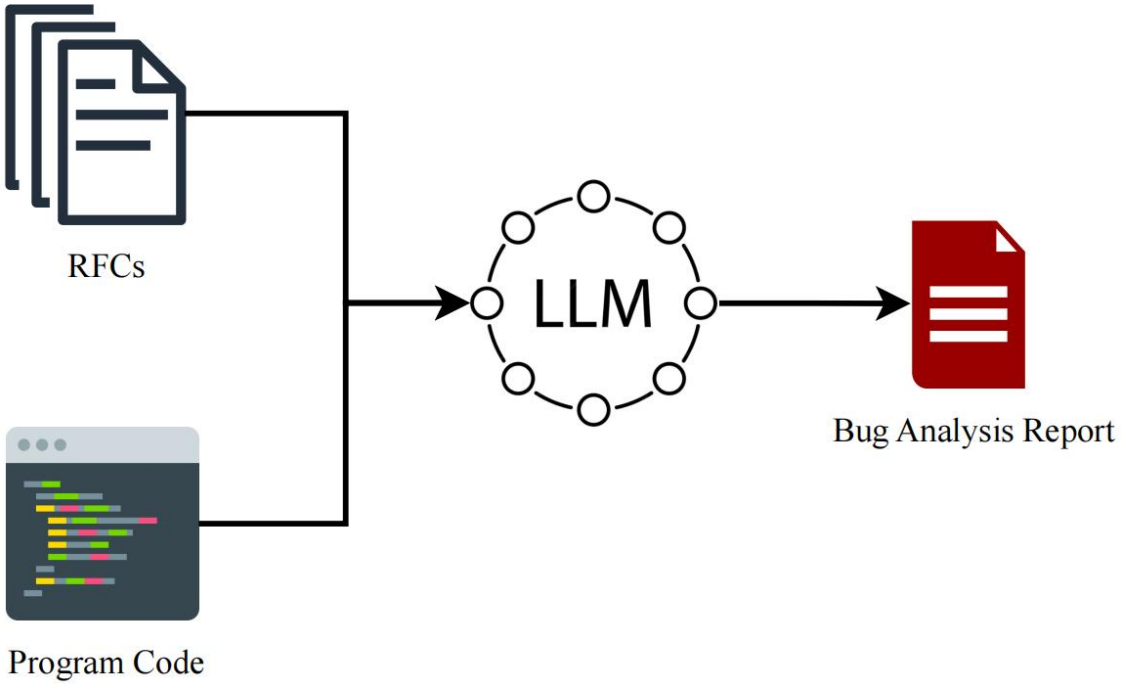
  Depends on predefined rules extracted from specifications, but lacks the semantic understanding needed to assess whether implementation logic truly preserves protocol invariants.

These limitations highlight the need for a new approach that can accurately detect silent but seriouse protocol non-compliance bugs.

Challenge & Solution

- LLMs have demonstrated strong capabilities in understanding both natural language and program semantics, providing a promising foundation for detecting protocol non-compliance bugs.

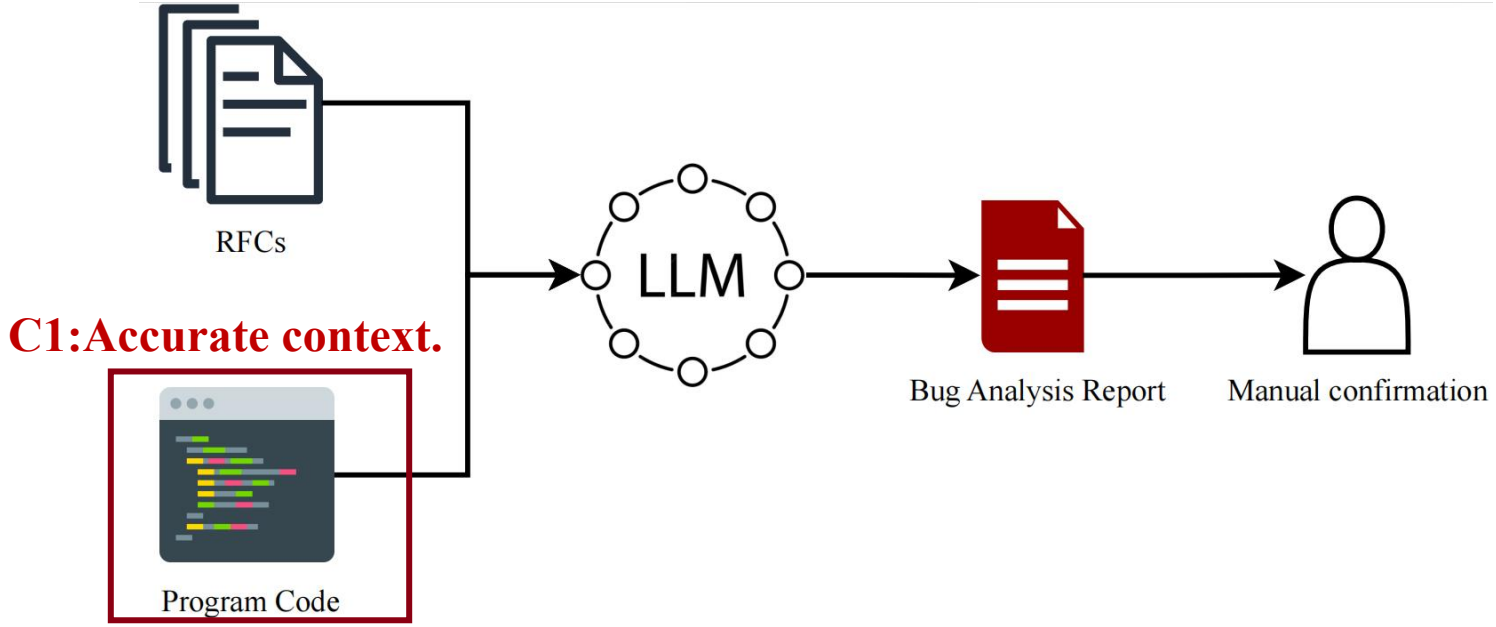




Challenge & Solution

- **C1: How to provide LLMs with appropriate rule-relevant code implementations?**

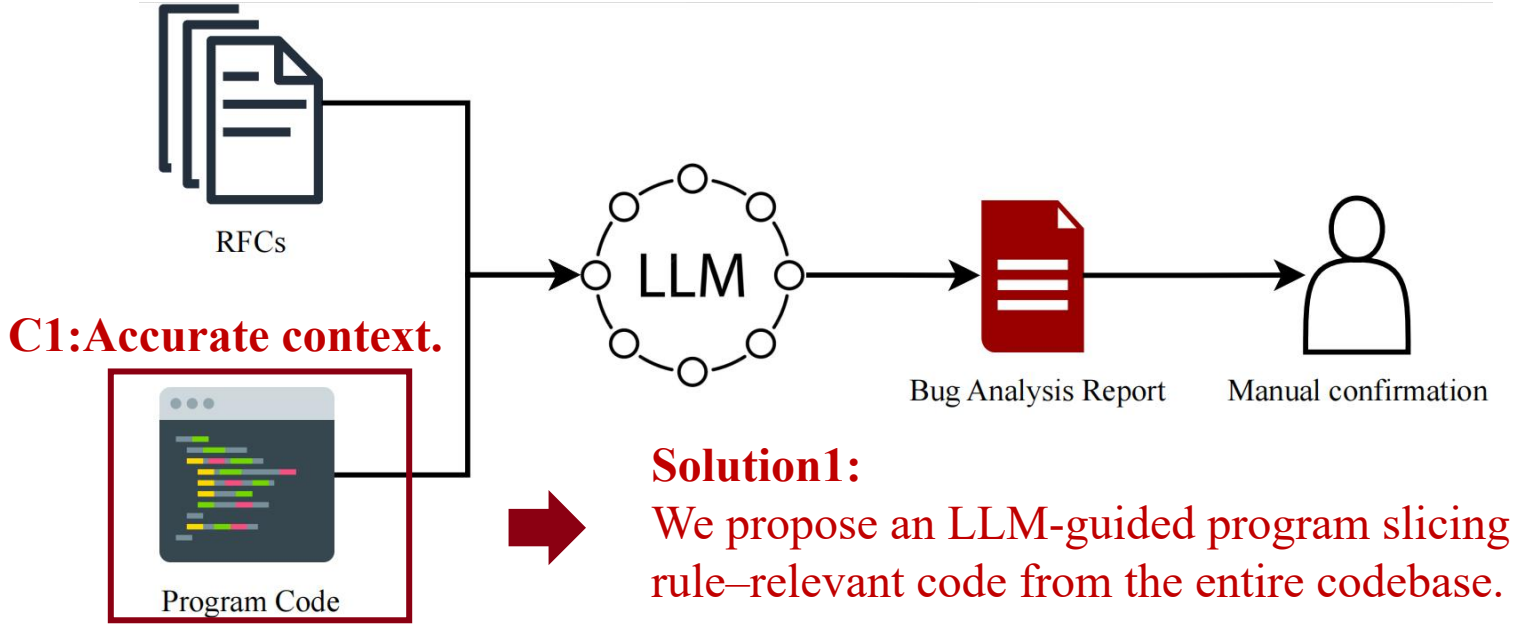
  Directly applying LLMs to an entire codebase is ineffective, as their reasoning ability is highly influenced by the relevance and focus of the provided input context.



Challenge & Solution

- **C1: How to provide LLMs with appropriate rule-relevant code implementations?**

  Directly applying LLMs to an entire codebase is ineffective, as their reasoning ability is highly influenced by the relevance and focus of the provided input context.

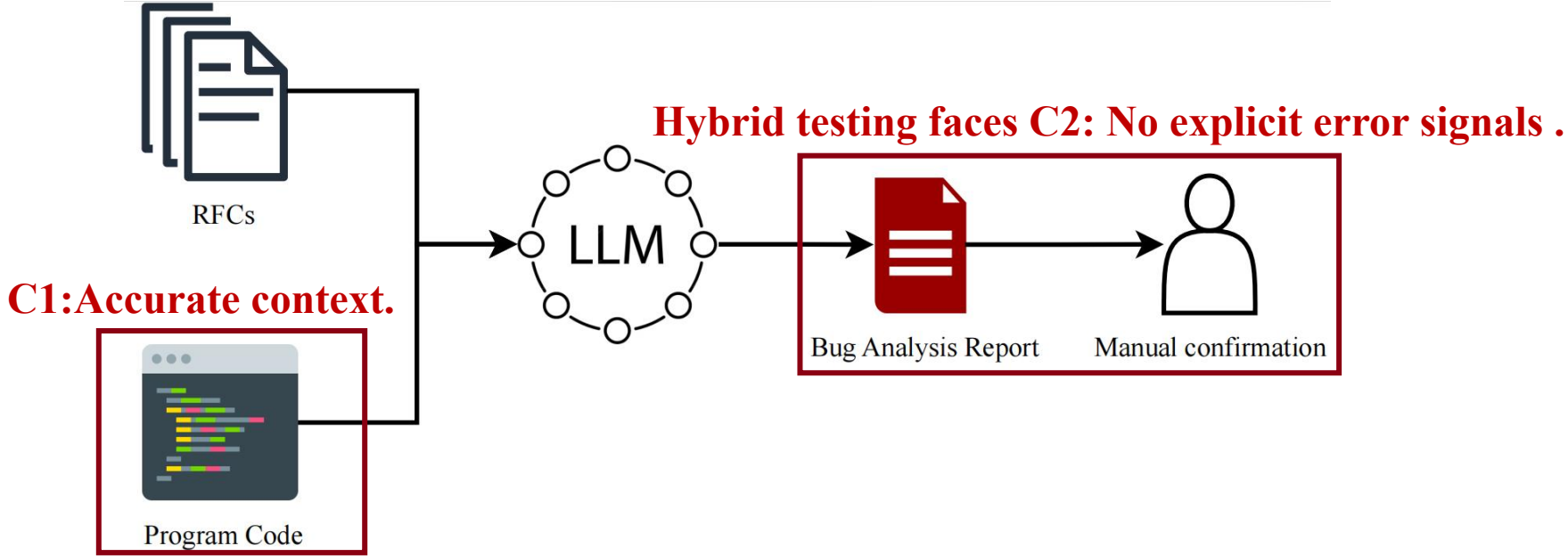


Solution1:
We propose an LLM-guided program slicing approach to extract rule–relevant code from the entire codebase.

Challenge & Solution

- C1: How to provide LLMs with appropriate rule-relevant code implementations?
- **C2: How to effectively verify non-compliance bugs without explicit error signals?**

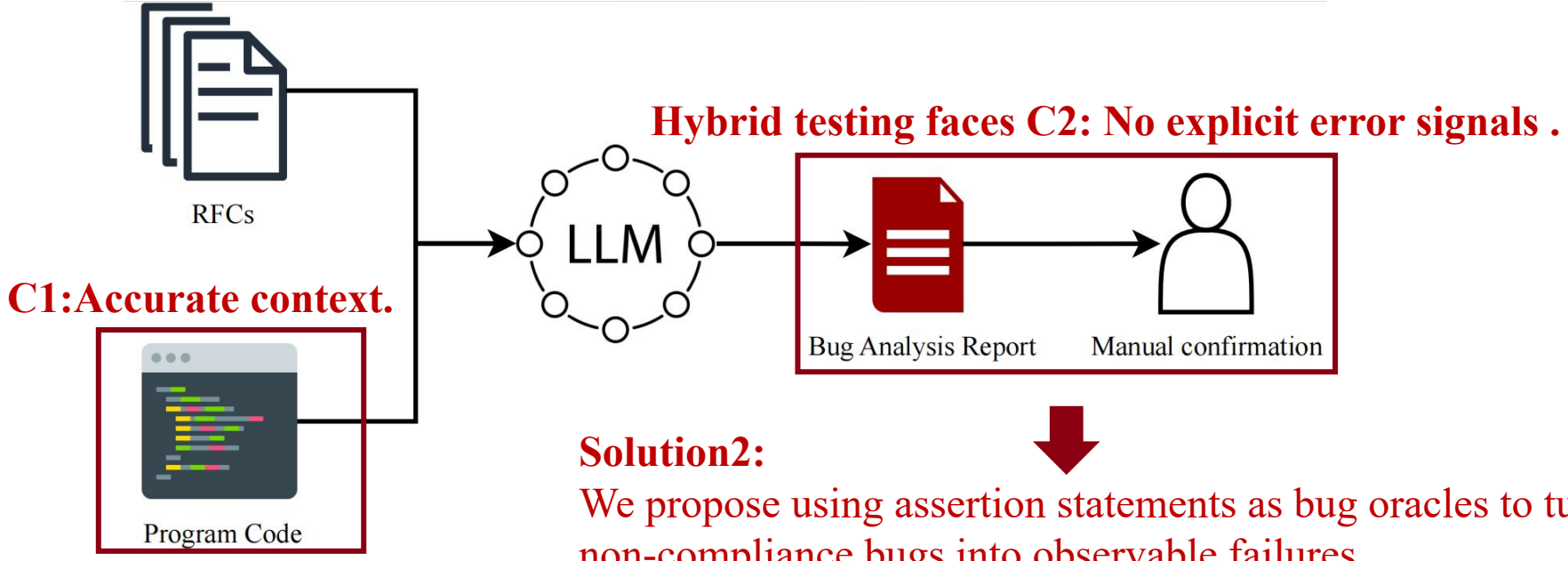
  Once bugs are identified by LLMs, existing approaches rely on manual confirmation. Hybrid testing (static analysis -> dynmaic fuzzing) is a promising workflow, but lacks of explicit signals for fuzzing.



Challenge & Solution

- C1: How to provide LLMs with appropriate rule-relevant code implementations?
- **C2: How to effectively verify non-compliance bugs without explicit error signals?**

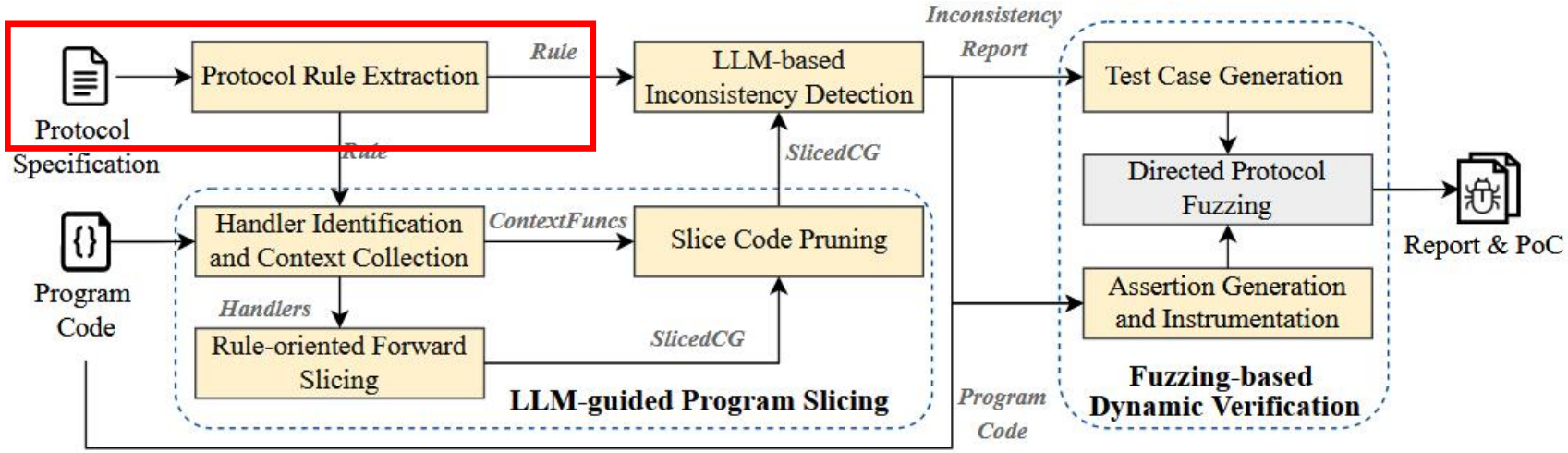
  Once bugs are identified by LLMs, existing approaches rely on manual confirmation. Hybrid testing (static analysis -> dynmaic fuzzing) is a promising workflow, but lacks of explicit signals for fuzzing.



Solution2:
We propose using assertion statements as bug oracles to turn silent non-compliance bugs into observable failures.

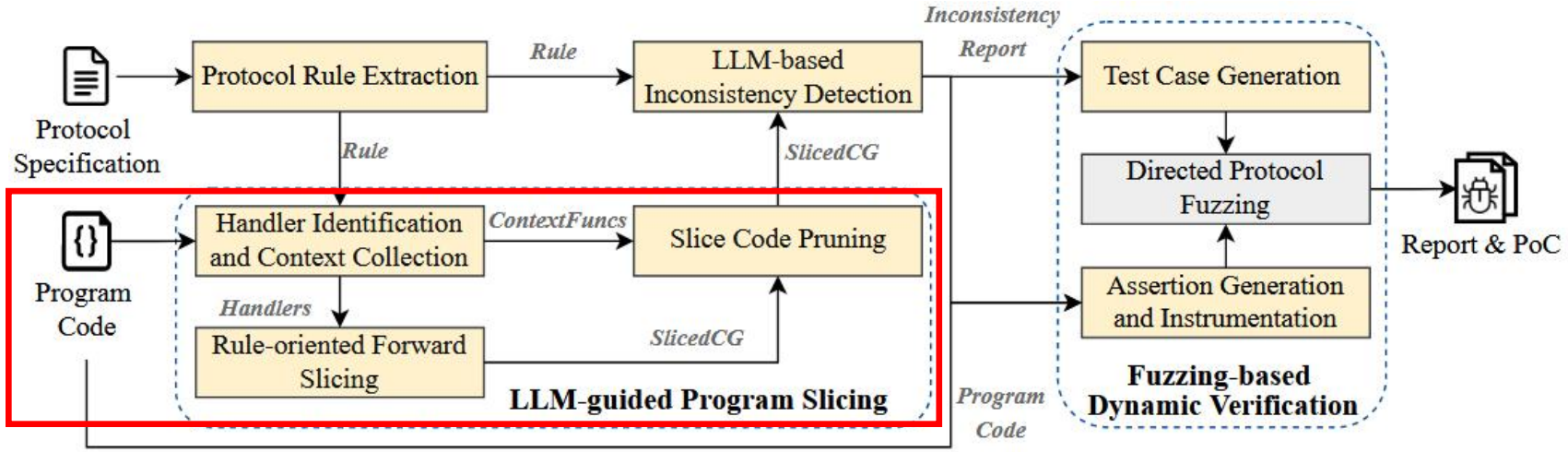
Our Approach: ProtocolGuard

- **Protocol Rule Extraction:** specification -> rules
- **LLM-guided Program Slicing:** rule + program code -> relevant code slice
- **LLM-based Inconsistency Detection:** rule + code slice -> inconsistency report
- **Fuzzing-based Dynamic Verification:** code + report + assertions + seeds -> PoCs



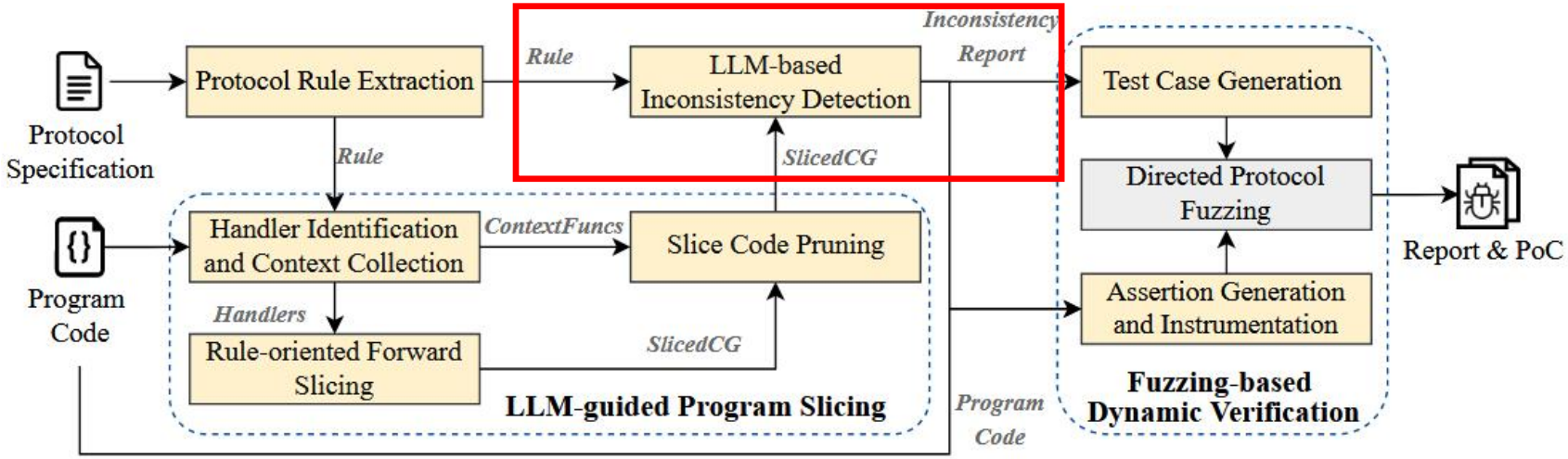
Our Approach: ProtocolGuard

- **Protocol Rule Extraction:** specification -> rules
- **LLM-guided Program Slicing:** rule + program code -> relevant code slice
- **LLM-based Inconsistency Detection:** rule + code slice -> inconsistency report
- **Fuzzing-based Dynamic Verification:** code + report + assertions + seeds -> PoCs



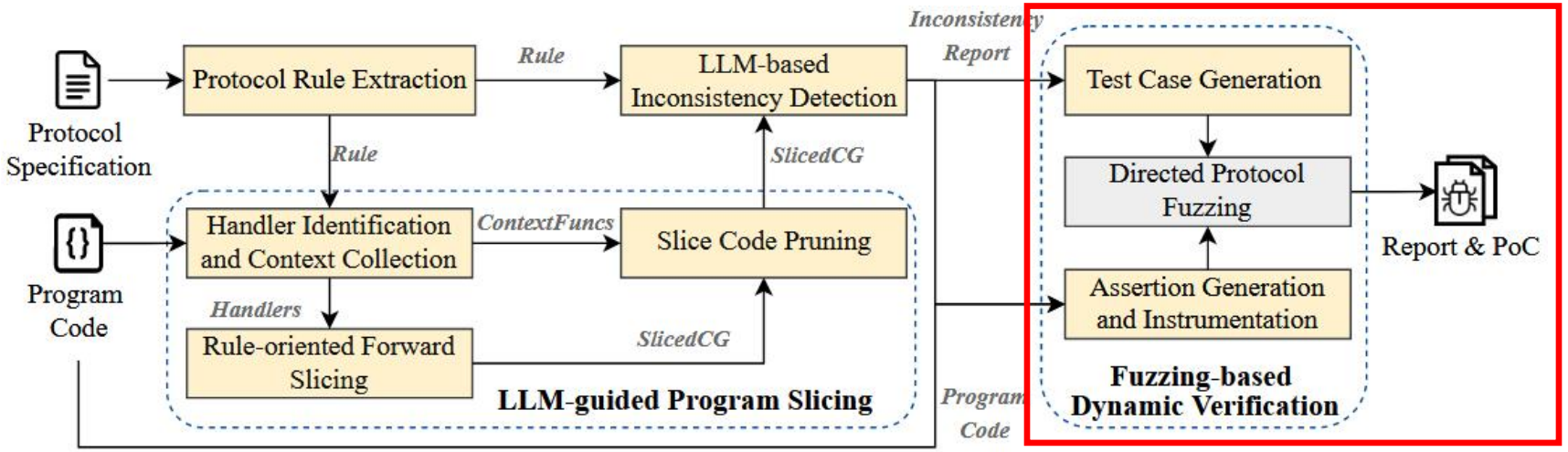
Our Approach: ProtocolGuard

- **Protocol Rule Extraction:** specification -> rules
- **LLM-guided Program Slicing:** rule + program code -> relevant code slice
- **LLM-based Inconsistency Detection:** rule + code slice -> inconsistency report
- **Fuzzing-based Dynamic Verification:** code + report + assertions + seeds -> PoCs



Our Approach: ProtocolGuard

- **Protocol Rule Extraction:** specification -> rules
- **LLM-guided Program Slicing:** rule + program code -> relevant code slice
- **LLM-based Inconsistency Detection:** rule + code slice -> inconsistency report
- **Fuzzing-based Dynamic Verification:** code + report + assertions + seeds -> PoCs



Protocol Rule Extraction

- **Input:** Protocol Specification
- **Output:** Structural Rule

➢ **RFC (TLS1.3)**

Modal keyword

If this extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the "supported_versions" extension to determine client preferences.

1.1. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Protocol Rule Extraction

- **Input:** Protocol Specification
- **Output:** Structural Rule

➢ **RFC (TLS1.3)**

Modal keyword

If this extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the "supported_versions" extension to determine client preferences.

```
{
  "rule": "If the supported_versions extension is present in the
  ClientHello, servers MUST NOT use the ClientHello.legacy_version
  value for version negotiation and MUST use only the \"
  supported_versions\" extension to determine client preferences.",
  "req_type": "ClientHello",
  "req_fields": [
      "legacy_version",
      "extensions",
      "extensions.supported_versions"
  ],
  "res_type": "",
  "res_fields": []
}
```

Rule description

Protocol Rule Extraction

- **Input:** Protocol Specification
- **Output:** Structural Rule

➢ **RFC (TLS1.3)**

Modal keyword

If this extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the "supported_versions" extension to determine client preferences.

```
{
  "rule": "If the supported_versions extension is present in the
  ClientHello, servers MUST NOT use the ClientHello.legacy_version
  value for version negotiation and MUST use only the \"
  supported_versions\" extension to determine client preferences.",
  "req_type": "ClientHello",
  "req_fields": [
      "legacy_version",
      "extensions",
      "extensions.supported_versions"
  ],
  "res_type": "",
  "res_fields": []
}
```

Request packet types and fields constrained by rules

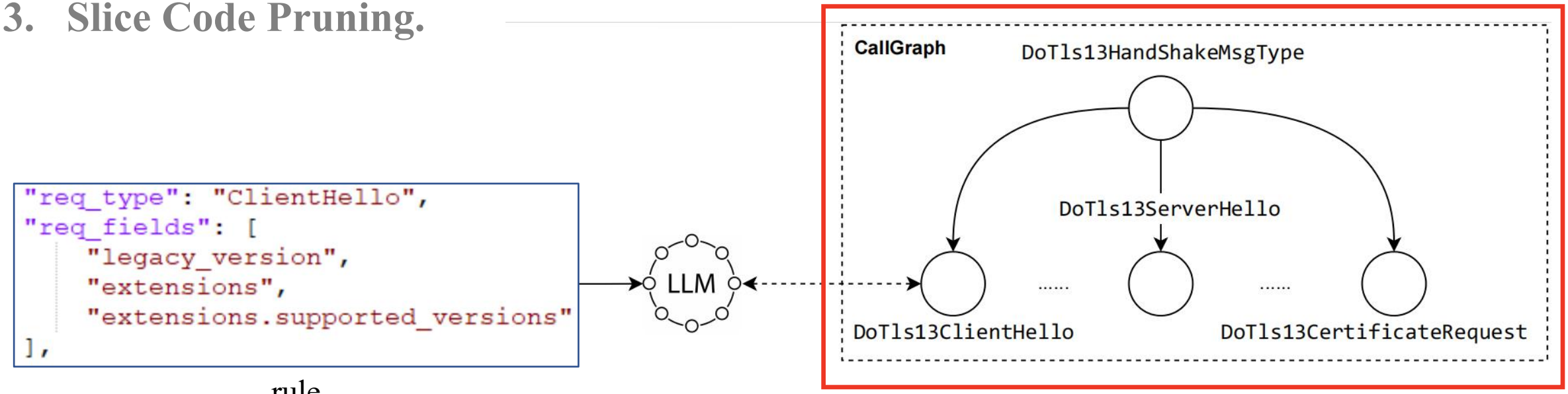
LLM-guided Program Slicing

1. **Handler Function Identification.**

   Insight: Message handling logic resides in handler function's call subgraphs

2. **Rule-oriented Forward Slicing.**

3. **Slice Code Pruning.**



rule

LLM-guided Program Slicing

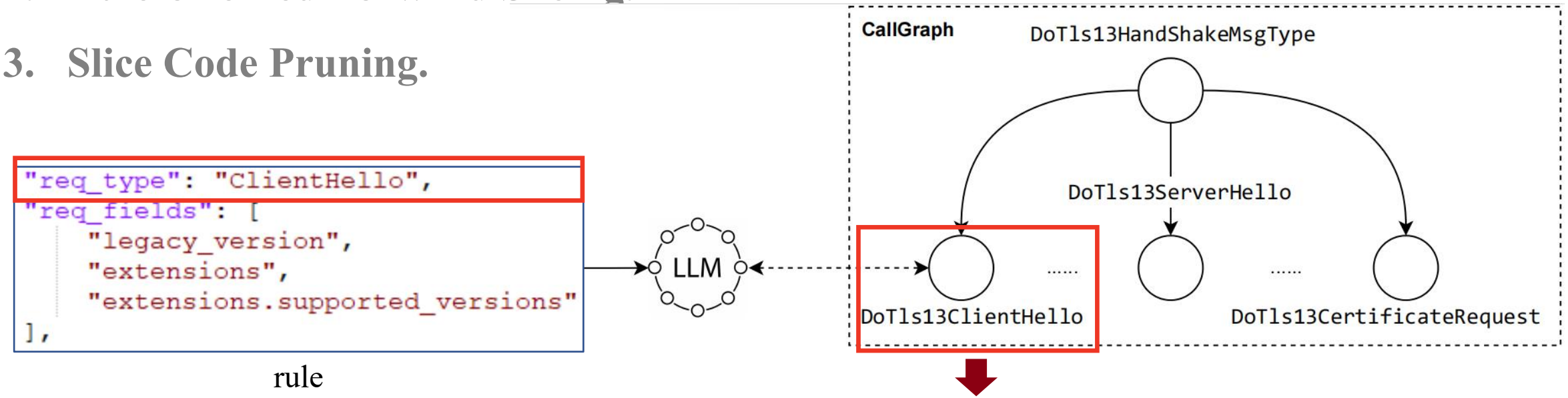
1. **Handler Function Identification.**

   Insight: Message handling logic resides in handler function's call subgraphs

   （1）LLVM-based call graph traversal -> （2）LLM-based handler function identification

2. **Rule-oriented Forward Slicing.**

3. **Slice Code Pruning.**



rule

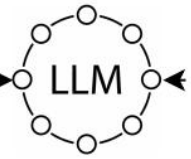
It is the entry function for processing ClientHello message.

LLM-guided Program Slicing

1. **Handler Function Identification.**
2. **Rule-oriented Forward Slicing.**
   （1）LLM-guided field identification
   （2）LLVM data dependency analysis
3. **Slice Code Pruning.**

```
"req_type": "ClientHello",
"req_fields": [
    "legacy_version",
    "extensions",
    "extensions.supported_versions"
],
```

rule



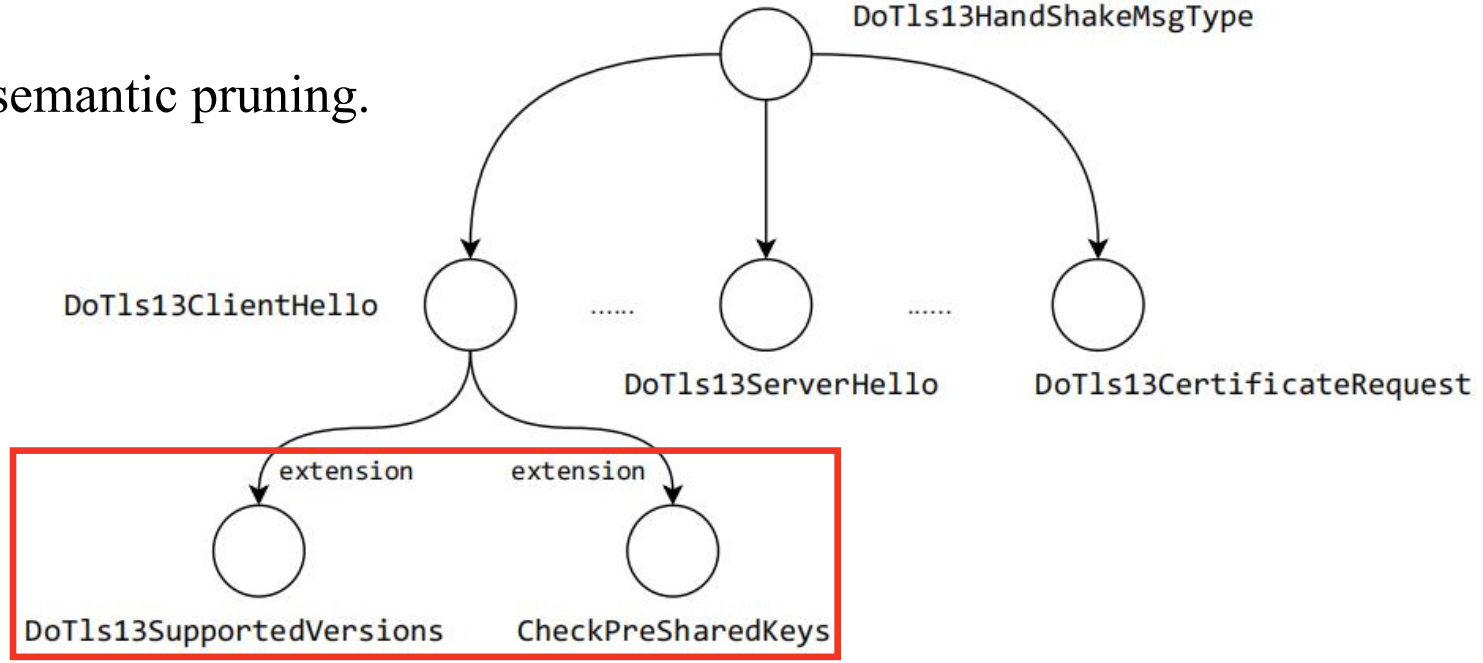
➢ **DoTLS13ClientHello:**

```
if (FindSuite(ssl->clSuites, 0, TLS_EMPTY_RENEGOTIATION_INFO_SCSV) >=
        0) {
    TLSX* extension;

    /* check for TLS_EMPTY_RENEGOTIATION_INFO_SCSV suite */
    ret = TLSX_AddEmptyRenegotiationInfo(&ssl->extensions, ssl->heap);
    if (ret != WOLFSSL_SUCCESS) {
        ret = SECURE_RENEGOTIATION_E;
        goto out;
    } else {
        ret = 0;
    }

    extension = TLSX_Find(ssl->extensions, TLSX_RENEGOTIATION_INFO);
    if (extension) {
        ssl->secure_renegotiation =
                                (SecureRenegotiation*)extension->data;
        ssl->secure_renegotiation->enabled = 1;
    }
}
```

LLM-guided Program Slicing

1. **Handler Function Identification.**
2. **Rule-oriented Forward Slicing.**
3. **Slice Code Pruning.** LLM-guided semantic pruning.

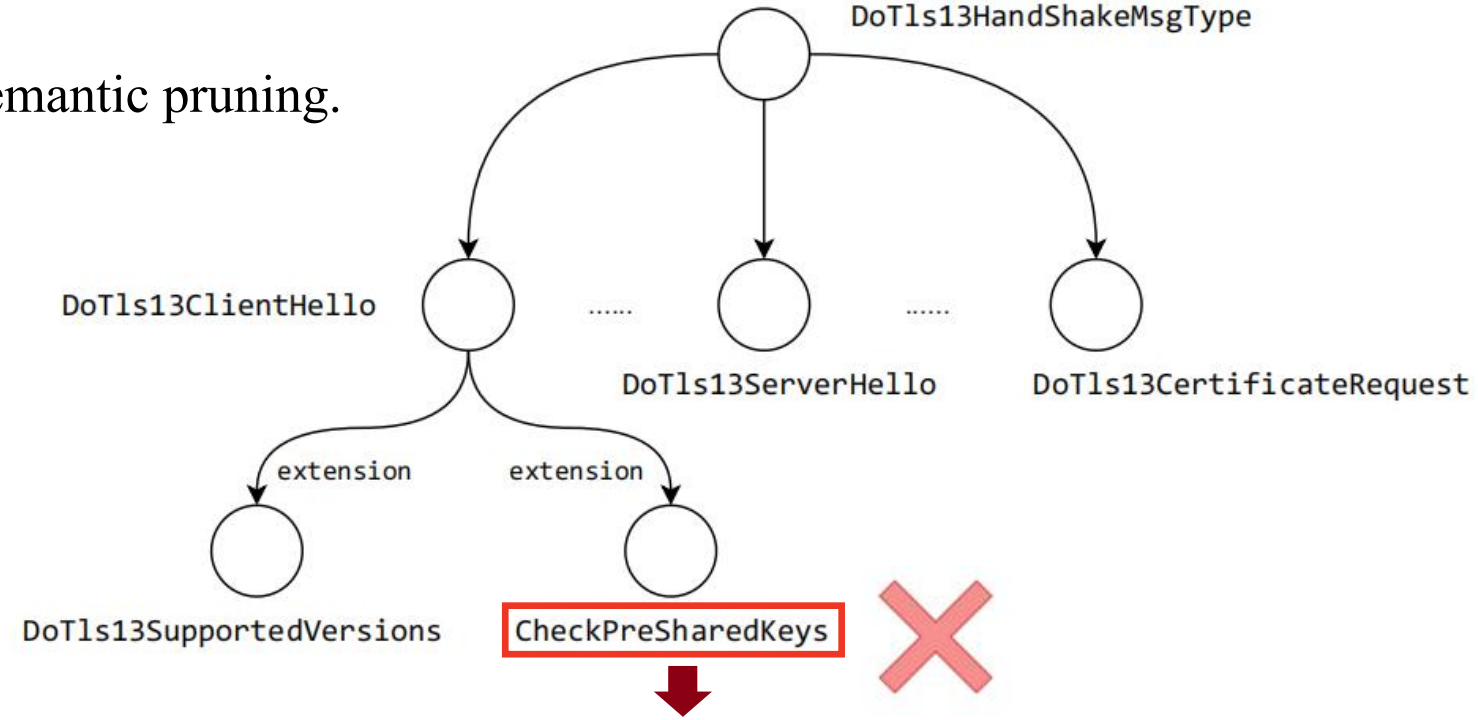
```
{
  "rule": "If the supported_versions extension is present in the
  ClientHello, servers MUST NOT use the ClientHello.legacy_version
  value for version negotiation and MUST use only the \"
  supported_versions\" extension to determine client preferences.",
  "req_type": "ClientHello",
  "req_fields": [
      "legacy_version",
      "extensions",
      "extensions.supported_versions"
  ],
  "res_type": "",
  "res_fields": []
}
```



LLM-guided Program Slicing

1. **Handler Function Identification.**
2. **Rule-oriented Forward Slicing.**
3. **Slice Code Pruning.** LLM-guide semantic pruning.

```
{
  "rule": "If the supported_versions extension is present in the
  ClientHello, servers MUST NOT use the ClientHello.legacy_version
  value for version negotiation and MUST use only the \"
  supported_versions\" extension to determine client preferences.",
  "req_type": "ClientHello",
  "req_fields": [
      "legacy_version",
      "extensions",
      "extensions.supported_versions"
  ],
  "res_type": "",
  "res_fields": []
}
```



Fuzzing-based Dynamic Verification

- **Assertion Generation and Instrumentation.**

  Bug Oracle: Assertions turn silent logic flaws into observable failures, serving as bug oracles for fuzzing.

Fuzzing-based Dynamic Verification

- **Assertion Generation and Instrumentation.**

  Bug Oracle: Assertions turn silent logic flaws into observable failures, serving as bug oracles for fuzzing.

  1. Extract expected validation constraints from inconsistency reports and protocol rules.
  2. Identify missing or incorrect validation logic in the implementation.
  3. Synthesis and instrument assertions at identified location.
  4. Compile and refine assertions untile syntactically correct.

Fuzzing-based Dynamic Verification

- **Assertion Generation and Instrumentation.**

  Bug Oracle: Assertions turn silent logic flaws into observable failures, serving as bug oracles for fuzzing.

1. Extract expected validation constraints from inconsistency reports and protocol rules.
2. Identify missing or incorrect validation logic in the implementation.
3. Synthesis and instrument assertions at identified location.
4. Compile and refine assertions untile syntactically correct.

```
1  static int connect_handler(struct io_event *e) {
2    // ...
3    // Generated assertion to validate client ID
4    assert(client_id_length_validation(c));
5    snprintf(cc->client_id, MQTT_CLIENT_ID_LEN, "%s
         ", c->payload.client_id);
6  }
7  // Generated helper function
8  static int client_id_length_validation(struct
       mqtt_connect *c) {
9    size_t original_len = strlen((char *)c->payload
         .client_id);
10   return (original_len < MQTT_CLIENT_ID_LEN);
11 }
```

Assertion statement

Fuzzing-based Dynamic Verification

- **Assertion Generation and Instrumentation.**

  Bug Oracle: Assertions turn silent logic flaws into observable failures, serving as bug oracles for fuzzing.

1. Extract expected validation constraints from inconsistency reports and protocol rules.
2. Identify missing or incorrect validation logic in the implementation.
3. Synthesis and instrument assertions at identified location.
4. Compile and refine assertions untile syntactically correct.

```
1  static int connect_handler(struct io_event *e) {
2    // ...
3    // Generated assertion to validate client ID
4    assert(client_id_length_validation(c));
5    snprintf(cc->client_id, MQTT_CLIENT_ID_LEN, "%s
         ", c->payload.client_id);
6  }
7  // Generated helper function
8  static int client_id_length_validation(struct
       mqtt_connect *c) {
9    size_t original_len = strlen((char *)c->payload
       .client_id);
10   return (original_len < MQTT_CLIENT_ID_LEN);
11 }
```

Assertion statement

Helper function for validating the field length against the buffer size.

Evaluation Setting

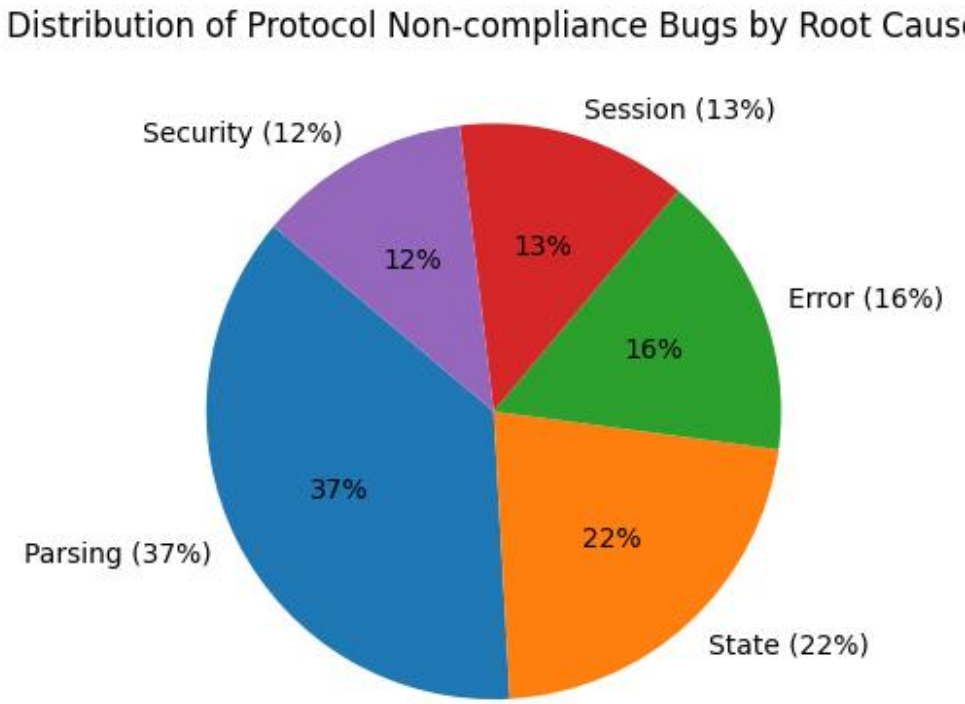
- **RQ1.** Can ProtocolGuard effectively detect non-compliance bugs in real-world protocol implementations? **(Bug Detection)**
- **RQ2**. How does ProtocolGuard compare to existing state-of-the-art tools in detecting protocol inconsistencies? **(Comparison)**
- **RQ3.** Can the generated assertions effectively serve as oracles for fuzzing to verify non-compliance bugs? **(Assertion Effectiveness)**

Evaluation Setting

- **Dataset: 11 open-source protocol implementations, covering six protocols.**
  - IoT communication: MQTTv3.1.1, MQTTv5.0, CoAP.
  - Secure transport: TLS 1.3
  - File transfer: FTP
  - Network services: DHCPv6
- **LLM Choice:**
  - DeepSeek series for program analysis
  - Claude for code generation

RQ1. Discovered Non-compliance Bugs

- ProtocolGuard extracted **420** rules from the offical specifications.
- It detected **181** inconsistencies with an overall precision rate **90.6%**.
- We confirm **158** unique non-compliance bugs.



- Parsing — 37%
- State — 22%
- Error — 16%
- Session — 13%
- Security — 12%

RQ2. Comparison with Existing Tools

- **Baselines**: Cursor with Claude 3.7、Cursor with DeepSeek R1
- ProtocolGuard achieves the best precision and recall
  - **ProtocolGuard**: **86.3%** precision, **81.3%** recall
  - **Cursor + Claude 3.7**: 71.7% precision, 76.8% recall
  - **Cursor + DeepSeek R1**: 49.3% precision, 52.0% recall

  Significant gap even when using the same LLM (DeepSeek R1)

Project	Cursor (Claude 3.7)			Cursor (DeepSeek R1)			ProtocolGuard (DeepSeek R1)		
	TP/FP/FN	Precision	Recall	TP/FP/FN	Precision	Recall	TP/FP/FN	Precision	Recall
Sol	37/3/7	92.5%	84.1%	16/5/28	76.2%	36.4%	**39/2/5**	**95.1%**	**88.6%**
pure-ftpd	16/9/4	64.0%	80.0%	10/14/10	41.7%	50.0%	**17/2/3**	**89.5%**	**85.0%**
libcoap	**5/3/2**	62.5%	**71.4%**	4/7/3	36.4%	57.1%	4/1/3	**80.0%**	57.1%
TLSE	21/5/7	80.8%	75.0%	18/11/10	62.1%	64.3%	**25/2/3**	**92.6%**	**89.3%**
Average	20/5/5	71.7%	76.8%	12/37/13	49.3%	52.0%	**21/2/4**	**86.3%**	**81.3%**

RQ3. Effectiveness of Assertions

- **Evaluation Setup**
  - Compilation check → semantic validation → 24h directed fuzzing
- **Overall Effectiveness**
  - **100%** syntactic correctness (all assertions compiled successfully)
  - **88.9%** semantic accuracy (correctly reflect protocol rules)
  - **68.4%** crash-triggering rate under directed fuzzing

The generated assertions effectively transform silent non-compliance bugs into observable assertion failures in most cases.

Conclusion

- We present ProtocolGuard, a novel hybrid framework for detecting protocol non-compliance bugs by combining **LLM-guided static analysis** and **fuzzing-based dynamic verification**.
- We implement a prototype and evaluate it on **11** real-world protocol implementations, uncovering **158** previously unknown non-compliance bugs, including multiple CVEs.

Conclusion

- We present ProtocolGuard, a novel hybrid framework for detecting protocol non-compliance bugs by combining **LLM-guided static analysis** and **fuzzing-based dynamic verification**.
- We implement a prototype and evaluate it on **11** real-world protocol implementations, uncovering **158** previously unknown non-compliance bugs, including multiple CVEs.

Thank you!

Q&A

songxiangpu@mail.sdu.edu.cn