RFCFuzz: Fuzzing for RFC-violations in SSL/TLS Protocol

RFCFuzz：基于 RFC 约束违反的 SSL/TLS 协议模糊测试

谢远峰

天津大学-智能与计算学部
网络安全-漏洞挖掘实验室

2024-09-25

1. 论文挑战与工作
2. 挑战一：有限状态机构建
3. 报文生成的讨论
4. 挑战二：消息匹配规则
5. 挑战三：规则指导变异
6. 论文工具框架参考
7. 论文验证实验
8. 研究案例
9. 相关工作：TLS Fuzz 框架
10. 网络协议测试挑战梳理

1. 论文挑战与工作

- survey 统一框架：
  1. protocol syntax acquisition and modeling（协议语法获取与建模）
  2. testcase generation（测试样例生成）
  3. test execution and monitoring（测试执行与状态检测）
  4. feedback information acquisition and utilization（反馈信息获取与利用）
- 实际挑战的提出
  - 挑战一：TLS 协议状态机的完整性与复杂性（补全状态转移）☑
    - 当前工作现状：针对具体实现进行状态机抽取；直接采用已有状态机；学习状态反馈利用算法推导
    - 论文工作 1：
      1. 由于协议状态性复杂，状态机不完整，需要补充状态转移，依据是 RFC 文档中自然语言描述。需要将文档自然语言描述转化为状态转移范式（符合状态机语法：现态，次态，条件，动作）
      2. 将状态转换范式嵌入到常规的有效状态机中，补充有限状态机
  - 挑战二：消息高度依赖上下文（正确解析-加密协议特有）☑
    - 当前工作现状：生成式（工具规范和协议语法），变异式种子文件
    - 论文工作 2：
      1.（wireshark）收集 TLS 常规报文序列构成 seed 集
      2. 解析：实现 **seed** 集里的每个消息序列（包含多条消息）与状态转换范式的匹配
      3. 框架：利用 LLM 解析状态转换范式，构建出报文消息框架（CHATAFL-NDSS-2024）
  - 挑战三：报文变异 Oracle 的精确性
    - 当前工作的现状：基于情感关键词的报文变异；手动定义可修改的字段内容
    - 论文工作 3：针对单条消息进行规则导向的精确变异
      1. 针对规则进行依赖性分析，筛选出不受依赖性影响的规则
      2. 需要参考逻辑攻击，将攻击方法形式化（绕过等）指导变异，实现变异精确性。（变异 trick）

2. 挑战一：有限状态机构建

实验一:规则抽取与范式转换

- 实验一：RFC 文档的规则抽取
  - 规则的定义？
    - 能够指导写代码，描述状态转换，可以匹配状态机范式（现态，次态，条件，动作）
    - 状态机范式的定义-（将 RFC 规则和实际报文类型字段进行绑定）
      - 现态：状态转移的起点；次态：状态转移的终点
      - 条件：（输入；判断）构成的集合
        输入：（输入消息类型+消息字段），判断：（对于消息类型+字段的约束）
      - 动作：（输出；约束）构成的集合
        输出：（输出消息类型+消息字段），判断：（对于消息类型+字段的约束）
  - 规则的分类？
    - 文本描述（自然语言描述）能完整匹配状态机范式的文本片段（1-n 句话）
    - 非文本描述（状态机流程图，代码，计算表达式）
  - 目标：构建一个规则数据集合
    - 单条文本规则的编码：（编号，所属章节，文本片段，现态，次态，条件，动作）
    - 单条非文本规则的编码->单条文本规则的编码：
      - 状态机流程图（是对消息序列的描述，是状态机的基本框架，可拆分成多条文本规则）
      - 代码（是对单条消息的结构描述，属于状态机中的条件/动作的最小(输入/输出)单位）
      - 计算表达式（是单条消息的字段约束，属于状态机中的条件/动作的判断操作）
  - 实验一具体流程-(输入 **RFC8446** 文档，产出 **8446** 文档状态转移范式数据集)
    - 针对 RFC 文档进行切块，基于完整语义进行文段切分
    - 判断切分片段是否包含规则语义（能否指导写代码，还是形式化定义）（人工+LLM）
    - 针对片段进行范式（起点，终点，条件，动作，包含情感关键词(MUST)）匹配（人工+LLM）
  - 阶段结果: 233 条规则-56 非规则-51 无关键词-带有 MUST/MUST NOT 规则占据很大一部分）
    - 126 带关键词-(101(MUST),21(MUST NOT),11(SHOULD),5(SHOULD NOT))

状态转换规则

- 能够指导写代码，描述状态转换，可以匹配状态机范式（现态，次态，条件，动作）,包含情感关键词；
  - 核心元组（Start, End, Action, Guards）
    - Start/End: 状态起点/状态终点; Action:对于消息的处理，Guards: 判断条件
    - （START, RECVD_CH, Check ClientHello Format and Parse,ClientHello Legal Format）
- 样例一：client 发送报文的约束（server 解析报文的约束）
  - {4.1.1. Cryptographic Negotiation}[If there is no overlap between the received “supported_groups” and the groups supported by the server, then the server MUST abort the handshake with a “handshake_failure” or an “insufficient_security” alert.]
  - ClientHello 的 supported_groups 扩展与 server 的 supported_groups 没有交叉，则报错。
  - **(START: RECVD_CH, END: ALERT, ACTION: [abort the handshake and send alert], GUARDS: [no overlap between the received “supported_groups” and the groups supported by the server])**
- 样例二：server 发送报文的约束
  - {4.1.3. Server Hello}[A client which receives a legacy_session_id_echo field that does not match what it sent in the ClientHello MUST abort the handshake with an “illegal_parameter” alert.]
  - RECVD_SH 节点接收到 server 发送的不符合规范的 ServerHello 报文
  - **(START: RECVD_SH, END: ALERT, ACTION: [abort the handshake and send alert], GUARDS: [a legacy_session_id_echo field that does not match what it sent in the ClientHello])**

实验一回顾

- 为什么要构建这样一个状态机？
  - 运用已有状态机，在进行 FUZZ 时会有 False Positive（以为非法实际合法）的情况
- 如何构建一个状态机？
  - 输入为 RFC 文档
  - 方法一：基于专家知识的构建，手动查看，并进行规则的构建（Ground Truth 搭建）
  - 方法二：LLM 的构建，设计 Prompt 进行 RFC 文档的切分和范式构建
- 如何验证构建的状态机相比其它工具是更加有效的？
  - 状态机抽取的已有工作，做对比
    - 专家知识人工抽取/自动化抽取（我们用 LLM 所以跟自动化抽取比,人工构建用来做检验?）
      - RESTler（ICSE 2019）, RFCNLP(S&P 2022), ChatAFL(ndss 2024)
      - benchmark? PROFUZZBENCH(ISSTA 2021) (benchmark for 网络协议状态 Fuzzer)
    - 怎么对比？
      - 状态机的构建效果：RFCNLP(S&P 2022)- 与基于专家知识构建的进行比较
      - 状态机的使用效果：ChatAFL(ndss 2024) 比用于 Fuzz 后最终的状态覆盖（benchmark）
- 产出内容：
  1. 基于专家知识(3 个人独立构建并针对不一致沟通)构建的规则集合（基于状态转移范式）
  2. 基于 LLM 构建的规则集合（包含相关 Prompt 设计）
  3. 1.和 2.构建的规则集合进行对比，计算 2.方法的 F1 score 等指标
- 规则数据集字段内容
  - 编号，章节编号，章节名称，自然语言描述
  - 是否是规则，包含的关键词个数，关键词名称
  - 端点（server/client/both），（服务器/客户端）-（现态/次态），条件，动作

3. 报文生成的讨论

消息报文构建讨论

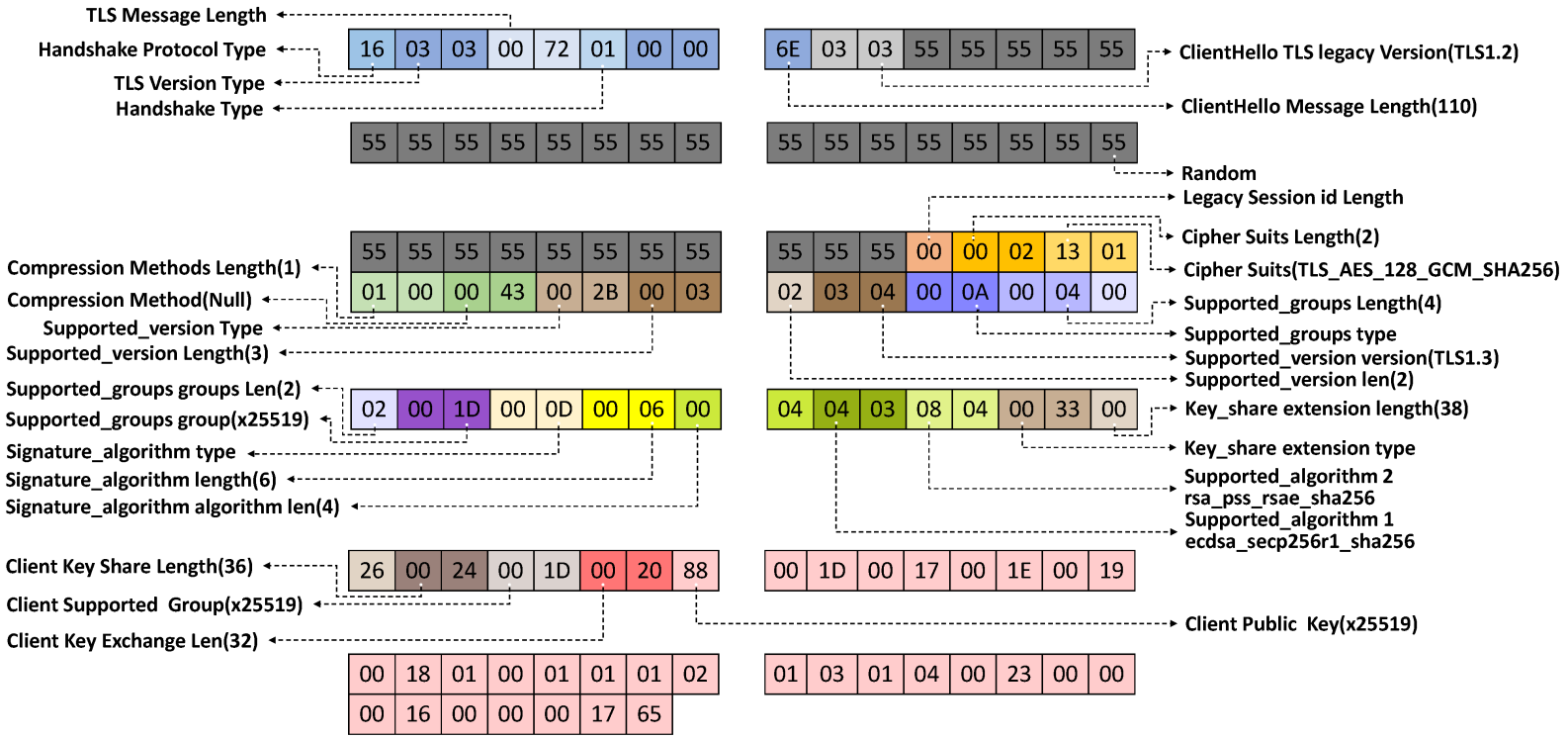
- 测试用例生成方法：
  1. 生成式（generation-based）：采用特定语法生成测试样例
  2. 变异式（mutation-based）：使用各种策略对用户提供的种子进行变异（种子质量，语法规范）
     - FITM（fuzzer-in-middle）：充当中间人拦截客户端和服务器之间的通信流量-捕获服务器和客户端之间的网络流量，并有选择地对其进行篡改和重放。
  3. 生成式（generation-based）+变异式（mutation-based）-（工具适配+详尽的协议语法规范）
- 已经进行的实验☑
  - 用 LLM 进行 CVE 漏洞报文的解析☑
  - 基于 RFC8446 文档进行简易 TLS ClientHello 报文构建☑
    - 报文的长度字段 LLM 无法有效理解，需要多轮反馈指导
  - 基于 RFC8446 文档进行复杂 TLS 报文构建，难以有效通过（长度，加密协议算法）✗
  - 截取的 TLS 报文可以通过特定方式解析加密部分☑
- 思考
  - 基于 RFC 全部文档进行学习构建，学习到的知识浓度低，规则指向性弱（预训练学习效果差）
  - 基于已有报文结构进行学习，LLM 可以快速理解报文结构（Example Learn 效果好）
  - 选取的样例也决定了学习到的消息结构
- 方法
  - 选取变异式的生成方法
  - 基于 RFC 文档进行规则蒸馏（过滤），告诉 LLM 哪些是有效规则，可以用于指导报文解析/生成(提升预训练学习效果、增强 Example 学习效果)
  - 作为 MITM 捕获客户与服务器间的所有 TLS 通讯消息（加密的部分通过特定方式解密）

TLS 简易报文构造

- 构建最小的 ClientHello 报文(LLM 构建+脚本修正)

```
// 1. 输入附录中报文的伪代码（B.1-B.4）基于以上补充信息，构建最小 TLS1.3Clienthello 消息结构，并逐步验证，确保除字段长度，需要算法计算的字段外合法，random 字段也随机填充。
//  --->(输出最小的 clienthello 消息，确保除长度，公钥外合法)
// 2. （补充背景知识，介绍 supported_groups，key_share 算法计算，书写代码，生成公钥填充并替换）
//  --->(对于明文发送的 ClientHello 消息结构进行了删减，确保结构最简，缺省部分按照数字顺序填充，输出也说明了公钥的填充方法 x25519)

// (开启新的一轮聊天，输入获得消息字节流，给出计算 extension length 的方法，给出计算 clienthello length 的方法)，作为一个网络协议专家，请你查看以上 clienthello 消息，判断是否合法，如果不合法，给出修改，最后输出修改后完整的 clienthello 消息。
// --->(输出了修改的完整 clienthello 消息并带有注释)
// 针对以上文本，删除注释和多余空格，生成完整连续的字节流
// --->(生成了完整连续的字节流)
```



4. 挑战二：消息匹配规则

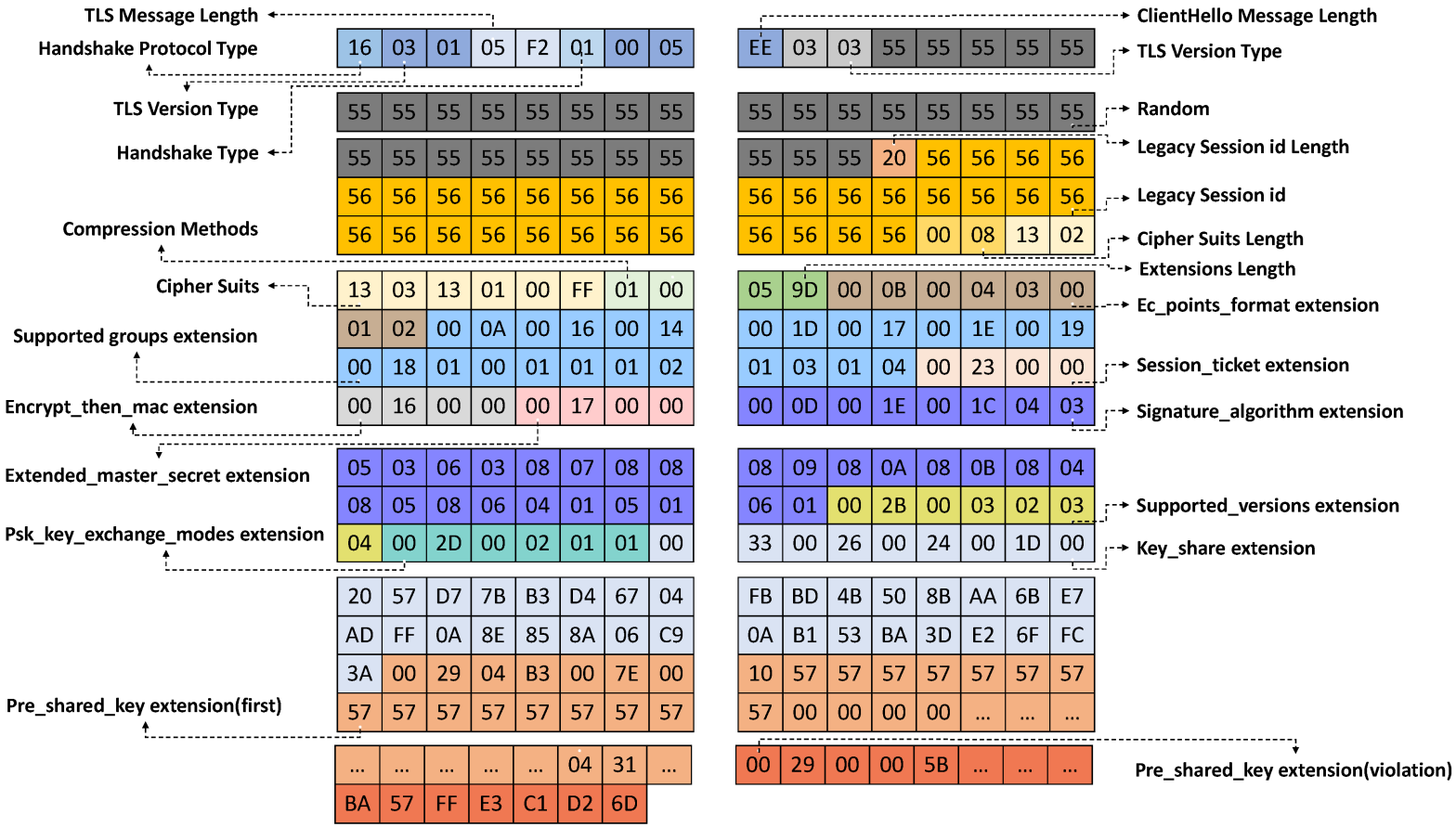
实验二：报文规则匹配

- 实验二：报文消息规则匹配
  1. 报文消息的收集：收集 Client 和 Server 沟通的 TLS 报文
  2. 报文消息的规则匹配：将 TLS 报文序列和状态转移范式进行匹配
- 报文消息的收集：收集 Client 和 Server 沟通的 TLS 报文，利用浏览器进行报文加密部分的解密
- 报文消息的状态转移范式规则（现态，次态，条件，动作）匹配：
  - 报文消息序列可以理解为单向链表;状态转换范式规则可以理解为有向边（包含起点，终点,）
  - 将抽象的状态转移规则应用到具体的 TLS 报文序列上（MAP-1）
    1. 获取完整的 TLS 报文序列,比如一次完整的 TLS 握手过程。
    2. 实验一构建的完整状态机图（由状态转换规则集构成）
    3. 逐一检查报文序列中的每个报文,尝试将其与状态转移规则中的节点匹配。
    4. 匹配成功后,建立了报文和状态节点之间的对应关系。
  - 将长的报文序列分解成有意义的小单元（MAP-2）
    1. 利用 MAP-1 中建立的对应关系,我们知道每个报文对应的起止状态节点。
    2. 将报文的起止状态映射成状态转换规则中的起止点,即"现态"和"次态"。
    3. 根据这些状态对,我们可以在报文序列中划定边界,将整个序列切分成多个小段。
       每个小段代表了从一个状态到另一个状态的转换过程。
  - 匹配单个报文需要的满足的约束转换条件（MAP-3）
    1. 对于 MAP-2 中切分出的每个报文段,我们匹配出对应的状态转移规则。
    2. 每个状态转移规则都有一些特定的条件,这些条件定义了何时可以从现态转移到次态。
    3. 这些特定的条件存在交叉，需要针对这些条件进行划分，具体到消息类型+消息字段
    4. 检查报文的内容,确认它是否满足这些条件。如果满足条件,则确认这个报文确实发生了符合该条件状态转移
- 验证实验：人工构建+LLM 匹配（人工匹配的结果去校验 LLM 输出结果，计算 F1 分数等）
- 产出内容：针对 MOTIVATION EXAMPLE 的规则进行人工匹配，LLM 实现的匹配结果

1. target: C:\Windows\System32\schannel.dll
2. 违反规则： RFC8446-4.2Extensions

> When multiple extensions of different types are present, the extensions MAY appear in any order, **with the exception of "pre shared key" (Section 4.2.11) which MUST be the last extension in the ClientHello** (but can appear anywhere in the ServerHello extensions block).

3. 涉及 TLS1.3 协议 client 端状态转换： START →WAIT_SH(发送 clienthello 消息)
4. 涉及的报文结构(应用层报文):有 2 个 psk,第二个 psk 使第一个 psk 非法



5. 挑战三：规则指导变异

实验三：规则指导的精确变异

- 实验三：把特定匹配规则作为变异策略指导变异
  - 针对实验二匹配的规则做分类（验证性规则，声明性规则）

    为了降低实验难度，只收集针对单个节点，包含(MUST (NOT), REQUIRED, SHALL (NOT))

    1. 验证性规则 (Verifiable Enforcement Rules)：

       这类规则可以被明确验证，并且违反时会触发可预见的执行机制。"验证性"表明这些规则可以被程序或协议实现直接检查，"执行"暗示违反时会有明确的后果。

       The “pre_shared_key” extension MUST be the last extension in the ClientHello (this facilitates implementation as described below). Servers MUST check that it is the last extension and otherwise fail the handshake with an “illegal_parameter” alert.
    2. 声明性规则 (Declarative Compliance Rules)：

       这类规则是协议声明的要求，但可能没有明确的验证或执行机制。"声明性"表示这些规则是协议明确陈述的，"遵循"暗示这些规则应该被遵守，但可能没有明确的验证或违反后果。

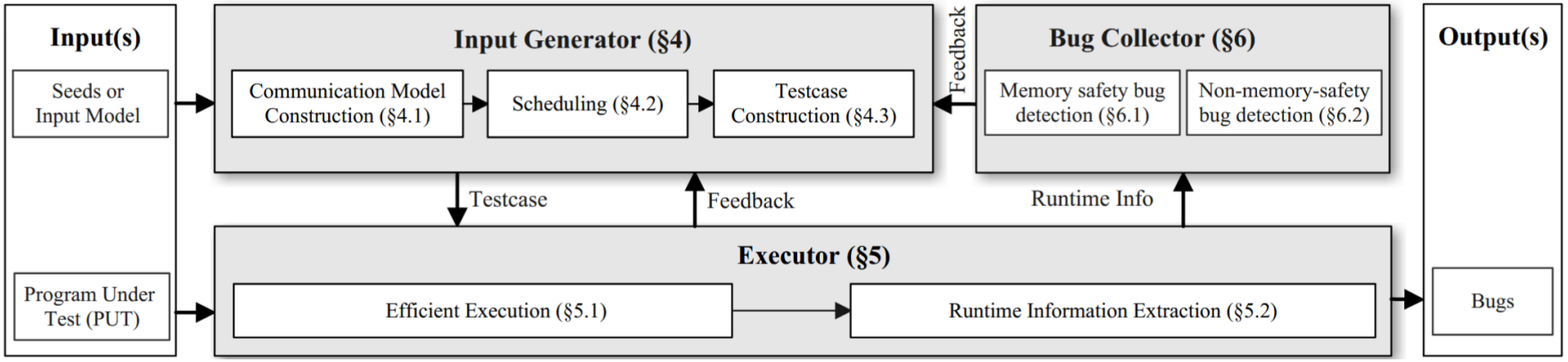
       legacy_version: In TLS 1.3, the client indicates its version preferences in the “supported_versions” extension (Section 4.2.1) and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2.
  - 声明性规则作为 Oracle 指导报文变异（从 Fuzz 工具角度，我们希望做的：）
    1. 种子消息序列（~~满足验证性规则和声明性规则~~）–TLS-Anvil
    2. 覆盖常规的通信情况（满足声明性规则）–已有的工具实现部分 — 罗权提供的 CVE
    3. 覆盖非常规的通信情况（不满足声明性规则）

    – 以声明性规则作为变异 Oracle，构造出违反这些声明性规则的测试样例

    (分层次，一部分用脚本，一部分用 LLM)

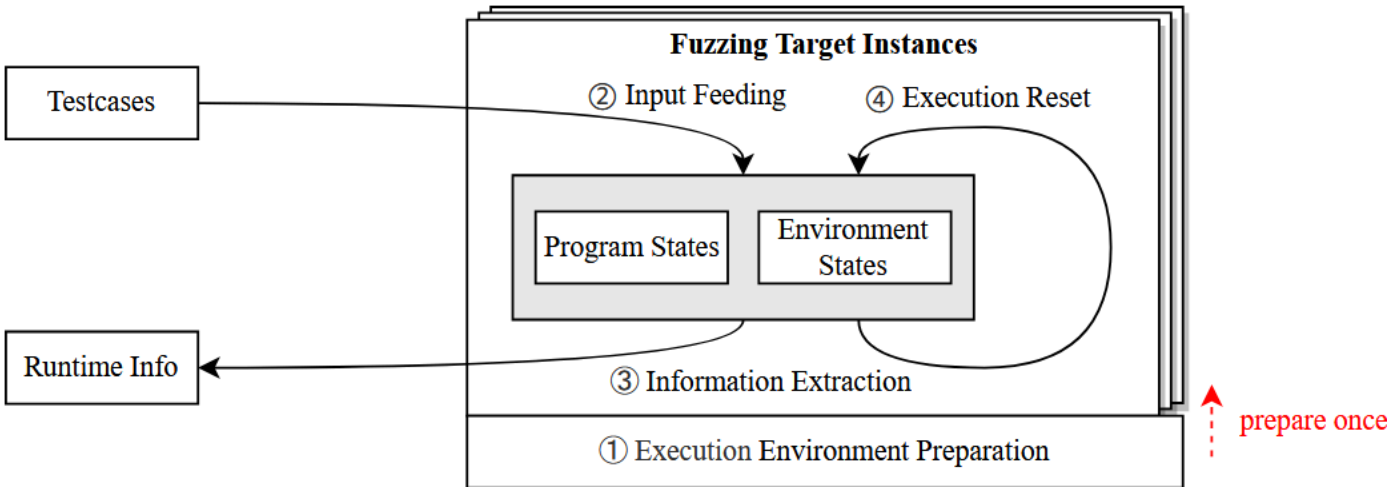
6. 论文工具框架参考

框架组件-Input Generator



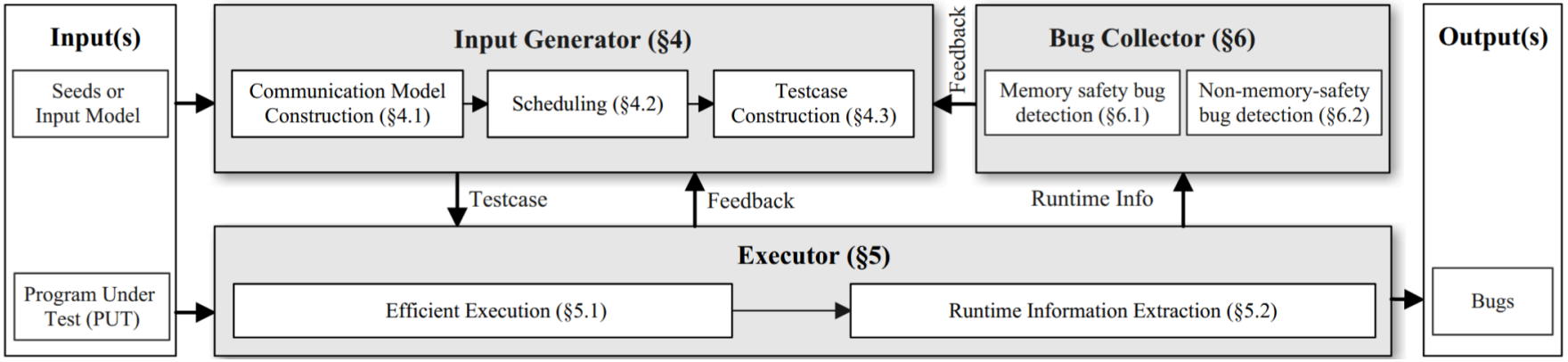
- Communication Model Construction:
  - Top-Down-Automatic-LLM(基于 RFC 用 LLM 自动化抽取)
  - 对比：RFCNLP, ChatAFL
- Task Scheduling：
  - Hierarchical Approaches - 分层方法（针对单个状态）
    1. Rarity-preferred：状态执行次数 - (AFLNET, StateAFL)
    2. Complexity-preferred：基本块链接次数, 状态深度, 可变异策略数量 - (ICSFuzzer, Pulsar)
    3. Performance-preferred: 提示代码覆盖率，状态覆盖率，发现更多 BUG - (AFLNET, StateAFL)
  - Monolithic Approaches - 单片方法（状态间转移）
    1. Rarity-preferred：状态执行次数 - (SGFuzz)
- Testcase Construction:
  - Packet-Level Construction Strategy
    - reducing input space：专家知识引导/报文片段拆解/报文结构语法解析
    - improving the effectiveness of triggering bugs：专家知识引导/Study 批量漏洞结论指导
  - Sequence-Level Construction Strategy
    - ~~Generation-based：利用既定的协议知识（如标准状态机和消息间的依赖关系）构建消息序列~~
    - mutation-based：智能管理消息序列中特定字段的相关性，如会话号、计数器（AFLNET）

框架组件-Executor



- key techniques and improvements in efficient execution
  1. Execution Environment Preparation - Parallel testing(Emulation) - (闭源/模拟只支持部分)
  2. Input Feeding - (~~OTA based~~, Socket-Based, ~~shared memory based~~, File-Based, ~~others~~)
     - Socket-Based(P2P, Proxy(MITM))
     - File-Based(利用插桩将网络接口替换为文件接口)
  4. Execution Reset - (reset time selection, execution state reset, execution environment reset)
     1. Time Selection - (固定时间间隔/代码插桩/忽略) - (AFLNET/AMPFuzz/SGFuzz)
     2. Execution State - (~~message-based reset, process restart~~, snapshot & recovery)
        - snapshot & recovery - (Process-Level/Virtual-Machine-Level)
          - Process-Level - (fork-based/ptrace-based(CRIU/DMTCP)) - (AFL-based/SNPSFuzzer)
          - Virtual-Machine-Level - (virtual machine hypervisors) - (Nyx-net)
     3. Execution Environment - (脚本/内存文件系统/虚拟机快照) - (.../SnapFuzz/Nyx-Net)
- key techniques and improvements in runtime information extraction
  3. Information Extraction
     - Hardware-Assisted - (Intel PT) - (Nyx-net)
     - Software-Based - (SSI/SDI 静态/动态插桩) - (...)
     - Externally-Observable-Behavior-Based - (状态码/报文/日志/旁路信息) - (AFLNET/SnipFuzz/...)

框架组件-Bug Collector



- Bug Oracle(Memory-Safety/Non Memory-Safety)
  - Memory-Safety
    (Fatal Signals and Sanitizers/~~Crash Logs and Debug Information~~/Error-Signaling Messages/)
    (~~Abnormal Physical Behaviors~~/~~Timeout and Liveness Checks~~)
    - Fatal Signals and Sanitizers - (SIGSEGV,SIGABRT,.../sanitizer)
    - Error-Signaling Messages - (special responses or status codes)
  - Non-Memory-Safety
    (Incorrect Message Content & State Transitions/~~Message Inconsistency in Transmission~~)
    (~~Abnormal Performance Indicators~~/~~Differences in Execution~~/~~Timeout and Liveness Checks~~)
    - Incorrect Message Content & State Transitions
      - canonical state machines（状态转移）
      - linear-temporal properties - (中间人攻击、信息泄露、拒绝服务、协议状态混乱) - 逻辑攻击
      - constraints of response messages - (消息字段约束违反)

框架组件汇总

1. 参考论文框架：A Survey of Protocol Fuzzing
2. Input Generator
   - Communication Model Construction: Top-Down-Automatic-LLM(RFCNLP, ChatAFL)
   - Task Scheduling：Hierarchical Approaches - 分层方法（针对单个状态）
     1. Rarity-preferred：状态执行次数 - (AFLNET, StateAFL)
     2. Complexity-preferred：可变异策略数量 - (ICSFuzzer, Pulsar)
     3. Performance-preferred: 提示代码覆盖率，状态覆盖率，发现更多 BUG - (AFLNET, StateAFL)
   - Testcase Construction: (Packet-Level/Sequence-Level)
     1. 压缩输入控件：专家知识引导/报文片段拆解/报文结构语法解析
     2. 提升触发 BUG 的有效性：专家知识引导/Study 批量漏洞结论指导
     3. 消息序列变异：智能管理消息序列中特定字段的相关性
3. Executor: (execution/runtime info extraction)
   1. 执行环境准备 - Parallel testing(Emulation) - (闭源/模拟只支持部分)
   2. 消息输入延迟 - Socket-Based(P2P, Proxy(MITM)); File-Based(利用插桩将网络接口替换为文件接口)
   3. 反馈信息抽取: Externally-Observable-Behavior-Based - (状态码/报文) - (AFLNET/SnipFuzz)
   4. 执行环境重置 - (reset time selection, execution state reset, execution environment reset)
      - 重置时间选取 - (固定时间间隔/代码插桩/忽略) - (AFLNET/AMPFuzz/SGFuzz)
      - 执行状态重置 - snapshot & recovery - Virtual-Machine-Level - (Nyx-net)
      - 执行外部环境重置 - (虚拟机快照) - (Nyx-Net)
4. Bug 收集: Bug Oracle(Memory-Safety/Non Memory-Safety)
   - Fatal Signals and Sanitizers - (SIGSEGV,SIGABRT,.../sanitizer)
   - Error-Signaling Messages - (special responses or status codes)
   - Incorrect Message Content & State Transitions
     (canonical state machines/linear-temporal properties/constraints of response messages)

7. 论文验证实验

规则抽取对照及消融实验

1. 参考论文框架：A Survey of Protocol Fuzzing
2. Input Generator
   - Communication Model Construction: (RFC 规则/标准抽取历史工作)
     Top-Down-Automatic (RESTler, RFCNLP, ChatAFL, HDiff)
     - Control Experiments: 对照实验, 说明方法的正确性和准确性
     - Ablation Experiments: 消融实验, 说明规则抽取方法内部的有效性

Year	Journal	Work	Target	Control Experiments	Ablation Experiments
2019	ICSE	RESTler	HTTP	-	Code Cov Num of bug
2022	DSN	HDiff	HTTP	-	-
2022	S&P	RFCNLP	TCP,SCTP,PPTP,...	预定义规则,Bert, designed-Model	文本分割策略, 预训练帮助, 手动修正
2024	NDSS	ChatAFL	RTSP,FTP,SMTP,...	LLM 抽取; 人工抽取	-
-	-	RFCFuzz	SSL/TLS	LLM 抽取; 人工抽取	文本分割策略; (全文本,章节,段落)

	Total/Precision	MUST (NOT)	REQUIRED	SHALL (NOT)
Maual	212	175(33)	1	3(0)
LLM				

	Total/Precision	MUST (NOT)	REQUIRED	SHALL (NOT)
LLM(Full)				
LLM(Chapter)				
LLM(Paragraph)				
Maual	212	175(33)	1	3(0)

测试样例构建历史工作

1. 参考论文框架：A Survey of Protocol Fuzzing
2. Input Generator- Testcase Construction: (Packet-Level-Mutation-Based/Sequence-Level-Mutation-Based)
   - 基于专家知识(抽取规则)进行敏感字段的修改(Packet-Level-Mutation-Based)
   - 基于专家知识(抽取规则)进行种子数据的规则匹配(Sequence-Level)
3. Executor: (execution/runtime info extraction)
   - 沿用旧的方法，没有对比，不是本文贡献点

- 历史工作(AFLnet, StateAFL, NSFuzz, DY-Fuzzing)- 比代码覆盖度，发现的 **CVE** 数量
- 目标工具 RFCDiff - 拟改进 **StateAFL/AFLNET** 工具实现预期效果
  - Execution State Reset: 执行状态重置;
  - Execution Env Reset: 执行环境重置;
  - Runtime Info: 执行环境的信息;
  - Monitoring Method: Fuzzer 监控的信息;
  - MiTM: Man-in-The-Middle-based packet injection;中间人报文注入
  - PSR: Process-level Snapshot and Recovery mechanism;进程级快照回复技术
  - MR: Message-based reset;重置类型消息重置执行状态到起点
  - UPSR: User-Provided Script Reset;用户定义脚本重置执行环境外部变量
  - SSI: Software Static Instrumentation;软件静态插桩

Year	Journal	Work	Target	Env Prep	Input Feeding	Execution State Reset	Execution Env Reset	Runtime Info	Monitoring Method
2020	ICST	AFLNET	General	-	Socket(P2P)	MR	UPSR	State & Code Cov	Resp & SSI
2021	ESE	StateAFL	General	-	Socket(P2P)	PSR	UPSR	State & Code Cov	SSI
2023	ToSEM	NS-Fuzz	General	-	Socket(P2P)	PSR	UPSR	State & Code Cov	SSI
2024	NDSS	DYFuzzing	SSL/TLS	-	Socket(MITM)	PSR	UPSR	Code Cov &# of Bugs	SSI
2024	NDSS	ChatAFL	RTSP,...	-	Socket(P2P)	PSR	UPSR	State & Code Cov	Resp
-	-	RFCFuzz	SSL/TLS	-	Socket(P2P)	PSR	UPSR	State Cov &# of Bugs	Resp

模糊测试对比与消融实验

- 对照试验
  - (规则抽取)-(规则匹配)-规则指导变异)
  - VMI:Validity of Mutated Inputs;单位时间生成语义正确样例的个数/比例
  - State Cov:覆盖到的协议节点状态
  - State Trans Cov:覆盖到的协议节点边转换的情况
  - Code Cov:函数行/边覆盖度
  - # of CVEs:发现 CVE 的数量

Year	Journal	Work	VMI	State Cov	State Trans Cov	Code Cov	# of CVEs
2020	ICST	AFLNET					0
2021	ESE	StateAFL					0
2024	NDSS	DYFuzzing					4
2024	NDSS	ChatAFL					4
-	-	RFCFuzz					2

- 消融实验
  - VMI:Validity of Mutated Inputs;单位时间生成语义正确样例的个数/比例
  - RE:Rule Extraction;规则抽取,抽取出了 TLS 消息结构,但是不包含情景约束
  - RM:Rule Mapping;规则匹配,将种子与规则匹配,可以包含情景约束
  - PM:Precised Mutation;精确变异,利用单一规则构建语义合法消息序列报文

Work Construction	VMI	State Cov	State Trans Cov	Code Cov	Note
AFLNET(Baseline)					AFL Mutation Strategy(bit flip...)
AFLNET+RE					符合 TLS 协议结构的消息
AFLNET+RE+RM					符合 TLS 状态转换的消息序列
RFCFuzz(RE+RM+PM)					语义正确的 TLS 消息序列

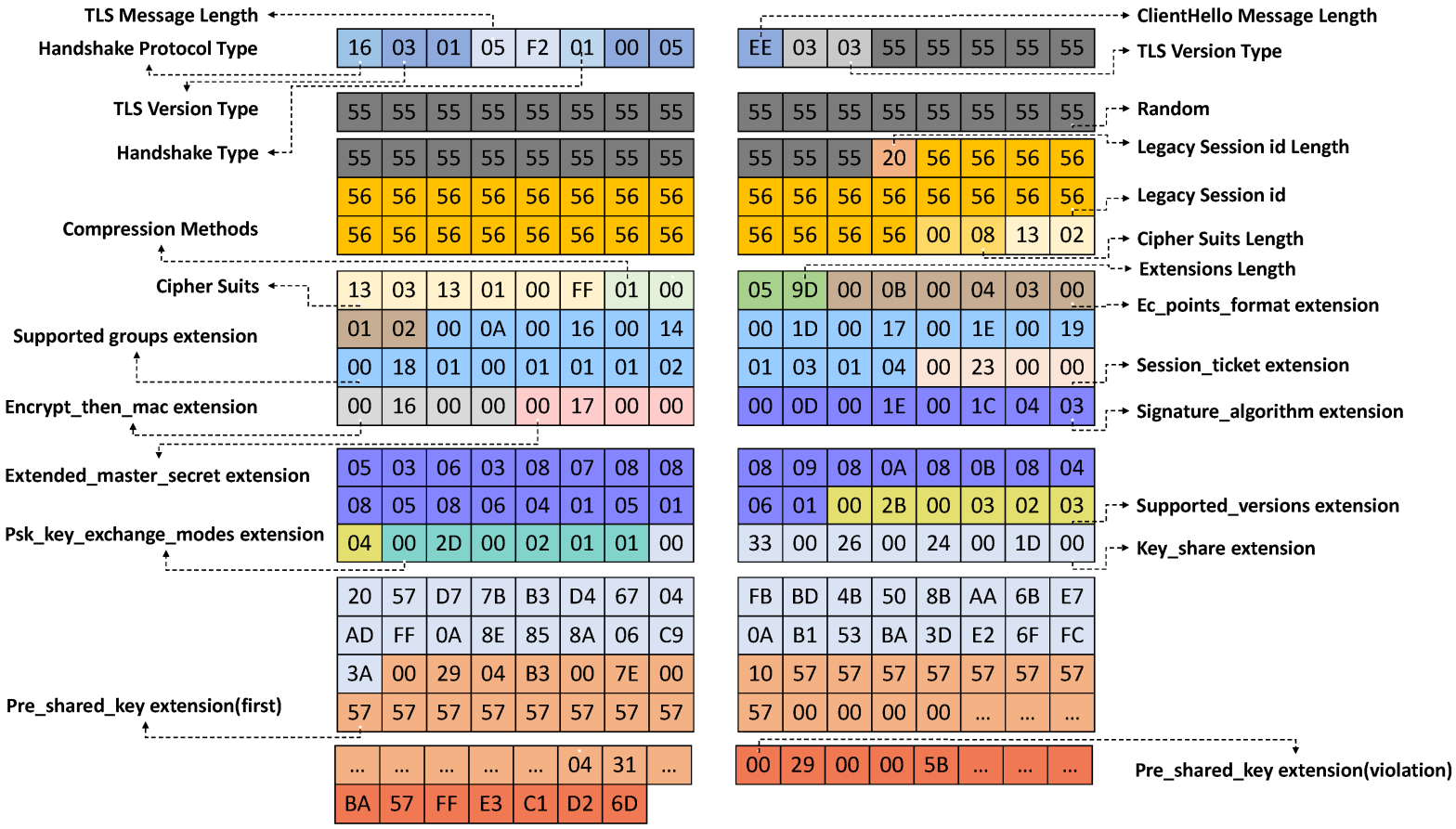
8. 研究案例

CVE-2023-28233(28234)

1. target: C:\Windows\System32\schannel.dll
2. 违反规则：RFC8446-4.2Extensions

> When multiple extensions of different types are present, the extensions MAY appear in any order, **with the exception of "pre shared key" (Section 4.2.11) which MUST be the last extension in the ClientHello** (but can appear anywhere in the ServerHello extensions block).

3. 涉及 TLS1.3 协议 client 端状态转换：START →WAIT_SH(发送 clienthello 消息)
4. 涉及的报文结构(应用层报文):有 2 个 psk,第二个 psk 使第一个 psk 非法



漏洞代码原理-引用错误

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Object {
    int value;
    int refCount;
} Object;

Object* createObject(int value) {
    Object* obj = (Object*)malloc(sizeof(Object));
    obj->value = value;
    obj->refCount = 1;  // 初始引用计数为 1
    return obj;
}

void incrementRefCount(Object* obj) {
    if (obj) {
        obj->refCount++;
    }
}

void decrementRefCount(Object* obj) {
    if (obj) {
        obj->refCount--;
        if (obj->refCount == 0) {
            free(obj);
        }
    }
}
```

引用计数是一种内存管理技术,用于跟踪对象被引用的次数。当引用计数降至零时,对象通常会被删除。然而,如果引用计数的管理不当,就会导致各种错误。

主要的引用计数错误包括:

1. 忘记增加引用计数
2. 忘记减少引用计数
3. 引用计数溢出
4. 循环引用

```c
int main() {
    // 正确使用
    Object* obj1 = createObject(10);
    incrementRefCount(obj1);  // refCount = 2
    decrementRefCount(obj1);  // refCount = 1
    decrementRefCount(obj1);  // refCount = 0, 对象被释放

    // 错误 1: 忘记增加引用计数
    Object* obj2 = createObject(20);
    Object* obj2Alias = obj2;  // 应该增加引用计数，但忘记了
    decrementRefCount(obj2);   // 对象被释放
    // 使用 obj2Alias 将导致未定义行为

    // 错误 2: 忘记减少引用计数
    Object* obj3 = createObject(30);
    incrementRefCount(obj3);  // refCount = 2
    // 忘记调用 decrementRefCount(obj3)
    // 即使不再使用，obj3 也不会被释放，导致内存泄漏

    // 错误 3: 引用计数溢出
    Object* obj4 = createObject(40);
    for (int i = 0; i < 1000000; i++) {
        incrementRefCount(obj4);  // 可能导致 refCount 溢出
    }
    // 如果 refCount 溢出回到 0，下一次 decrementRefCount 将错误地释放对象

    // 错误 4: 循环引用 (在 C 中较少见，但在更复杂的系统中可能发生)
    // 这通常需要更复杂的数据结构来演示，这里只是一个概念说明
    Object* objA = createObject(50);
    Object* objB = createObject(60);
    // 假设 objA 和 objB 相互引用，但我们没有适当的机制来打破这个循环
    // 即使外部不再引用 objA 和 objB，它们的引用计数也不会降到 0，导致内存泄漏

    return 0;
}
```

漏洞代码实例-引用错误

```
// CTls13ExtServer::ParsePreSharedKeyExtension 函数
int64_t CTls13ExtServer::ParsePreSharedKeyExtension(/* 参数 */) {
    // ... (前面的代码省略)
    // 漏洞：此函数可能被多次调用，导致引用计数错误或内存泄漏
    if (identityLength == 32) {
        // 情况 1：使用 session_id (32 位) 作为身份标识
        // 这通常用于在同一台机器上进行会话恢复
        sessionCacheItem = CSessionCacheManager::LookupCacheForServerItem(
            CSessionCacheManager::m_pSessionCacheManager,
            (struct CSslContext *)sslContext,
            identityData,
            0x20u,
            (struct CSessionCacheServerItem **)(sslContext + 0x3E0));
             // 在这里接收 CSessionCacheItem 指针

        // 漏洞：偏移量 0x3E0 处的 CSessionCacheItem 指针未被清除
        // 多次调用可能导致引用计数从 2 增加到 0xffffffff 再到 0，潜在地导致 UAF（使用后释放）
    }
    else {// 情况 2：使用 session_ticket 作为身份标识
        // 这用于分布式系统中跨不同服务器的会话恢复
        sessionCacheItem = CSsl3TlsServerContext::UnprotectAndDeserializeSessionState(
            (CSsl3TlsServerContext *)sslContext,
            identityData,
            identityLength,
            (unsigned __int8 *const )(sslContext + 0x4E1),
            *(unsigned __int8 *)(sslContext + 0x501));

        // 此函数内部调用 CSessionCacheManager::CacheRetrieveNewServerItem
        // 漏洞：多次调用可能导致内存泄漏，最终导致 lsass 进程崩溃
    }

    // ... (中间的代码省略)

    // CacheRetrieveNewServerItem 的另一个调用实例
    newSessionCacheItem = CSessionCacheManager::CacheRetrieveNewServerItem(
        sessionCacheManager,
        *((_DWORD *)context + 16),
        someData,
        (struct CSessionCacheServerItem **)context + 124); // 在这里接收 CSessionCacheItem 指针

    // 漏洞：最后生成的 CSessionCacheItem 未被清除
    // 没有调用相应的清理函数来处理 CSessionCacheItem 指针
    // ... (剩余代码省略)
}
```

```
// TLS 1.3 实现中的额外漏洞：
// 如果提供多个身份标识且只有最后一个身份标识是正确的，
// 同时精心构造 binders 长度，可以绕过 binder 验证过程。
// 这允许多次进入 ParsePreSharedKeyExtension 函数
// 而无需对所有 binder 执行哈希碰撞。

// CTls13ServerContext::VerifyBinder 函数 (简化版)
bool CTls13ServerContext::VerifyBinder(/* 参数 */) {
    // ... (前面的代码省略)

    // 漏洞：如果提供多个身份标识且只有最后一个是正确的，
    // binder 验证可能被绕过
    comparisonResult = RtlCompareMemory(/* 参数 */);

    // 在实际攻击场景中，可能通过在调试器中修改返回值
    // 强制使此比较始终成功

    return (comparisonResult == expectedLength);
}
```

Memory-Oracle 可以触发（引用错误/内存泄漏）
触发方式：包含两个 PSK 信息的 ClientHello 报文
目标：schannel.dll
构造报文：确保只违反一个规则，构造多个 identity 和 binder，让 binderlength 字段内容变大，将 binder 字段跳过，从而让服务器接收报文，而非 binder 字段不匹配就丢弃。

9. 相关工作：TLS Fuzz 框架

TLS1.3 通信过程

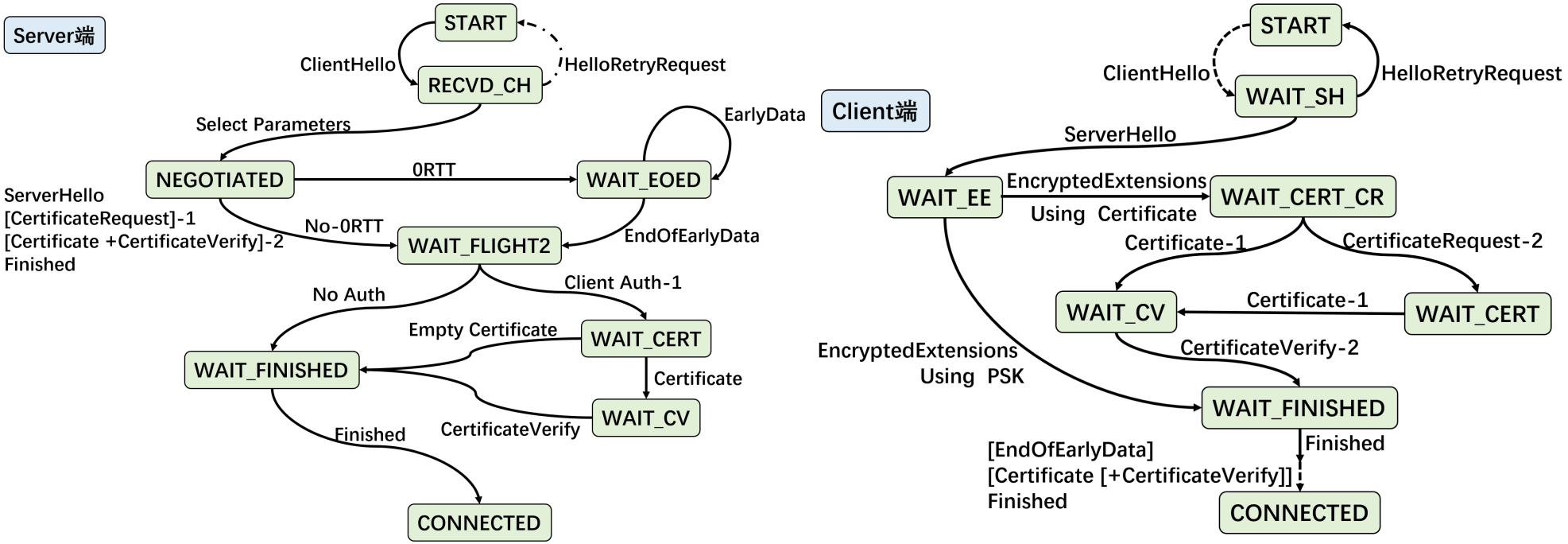
- TLS1.3 通信过程：
  - 密钥交换：
    - 客户端发送 ClientHello 消息，包含：
      1. 建议的密码套件
      2. 临时的(EC)DH 密钥共享或 PSK（或两者）
    - 服务器回应 ServerHello 消息，包含：
      1. 协商的连接参数
      2. 如需要，服务器的临时(EC)DH 密钥共享
    - 如果客户端建议不适合（如不支持的密码套件），服务器可能发送 HelloRetryRequest
  - 身份验证：
    - 成功交换密钥后，后续所有消息都会加密
    - 如果不使用预共享密钥，为防止中间人攻击，服务器必须进行身份验证：
      1. 发送 Certificate（如 X.509 证书）
      2. 发送 CertificateVerify（用认证密钥对整个握手的签名）
    - 服务器可选择要求客户端也进行类似的身份验证
  - 完成握手：
    - 服务器和客户端都发送 Finished 消息：
      1. 这是对整个握手过程的消息认证码（MAC）
      2. 提供密钥确认
      3. 将参与者身份绑定到会话密钥

已有模糊测试工具的局限

- Fuzzing protocols at the bit-level（比特级模糊测试工具）
  1. 可达性问题：无法处理多轮交互；难以实现复杂逻辑变异；难以达到触发 DY 攻击的深层状态
  2. 检测问题：主要专注于内存错误；无法检测协议级别的漏洞（如身份验证绕过）
- **Model-based protocols fuzzing（基于状态的协议模糊测试工具）**–不够精确
  - 模型局限性：使用有限状态机(FSM)抽象协议行为，但不够全面
  - 攻击覆盖不足：可检测特定类型攻击（如身份验证绕过）；无法全面捕获 DY 攻击
  - 消息变异能力不足：不能有效篡改消息内容；仅能修改有限的预定义值
  - 安全性判断问题：FSM 模型不专门为安全设计
    - 检测到的违规不一定是真正的安全攻击；需要人工检查确认实际漏洞
  - 协议级漏洞检测不足：无法自动识别所有类型的协议级漏洞
- Program verification and secure compilation（程序验证和安全编译）
  - 实施成本高：需使用特定编程语言（如 F*）重写协议；需投入大量时间和精力进行形式化证明
    - 用 F*编写的 QUIC 协议记录层证明花费约 20 人月,但未覆盖更复杂的握手协议
    - TLS 1.3 的证明仅限于记录层。TLS 1.2 的证明(使用 F7)未覆盖完整握手。
  - 适用性问题：无法直接应用于现有的、已部署的代码库
  - 可扩展性问题：难以验证完整的大型协议实现
  - 实际应用受限：在加密原语实现上有成功，但未能应用于复杂加密协议；无法有效检测和排除广泛使用的协议实现（如 OpenSSL）中的 DY 攻击
- Advantages of DY Fuzzing
  - 更全面的测试覆盖：可以模拟攻击者、客户端和服务器的多种角色。
  - 更强大的变异能力：能够进行逻辑层面的消息转换，特别是涉及加密操作。
  - 更精确的漏洞检测：能够识别协议级别的漏洞(捕获逻辑错误)，而不仅限于内存错误。
  - 结构化的方法：通过使用会话、代理和声明的概念，提供了更深入的协议分析能力。

协议文本状态机形式化

- 现状：异常的错误状态没有体现，无法根据常规状态机确认边缘测例的反馈（异常状态机）
- 异常
  - 网络故障或系统问题：unexpected_message(10)；bad_record_mac(20)；record_overflow(22)；decode_error(50)；internal_error(80)
  - 字段错误：illegal_parameter(47)；decrypt_error(51)；protocol_version(70)；
  - 安全错误：handshake_failure(40)；insufficient_security(71)；inappropriate_fallback(86)
  - 证书和身份验证错误：bad_certificate(42)；unsupported_certificate(43)；certificate_revoked(44)；certificate_expired(45)；certificate_unknown(46)；access_denied(49)；unknown_ca(48)；unknown_psk_identity(115)；certificate_required(116)；
  - 扩展错误：missing_extension(109)；unsupported_extension(110)；unrecognized_name(112)；bad_certificate_status_response(113)；no_application_protocol(120)



10. 网络协议测试挑战梳理

- survey 统一框架：
  1. protocol syntax acquisition and modeling（协议语法获取与建模）
  2. testcase generation（测试样例生成）
  3. test execution and monitoring（测试执行与状态检测）
  4. feedback information acquisition and utilization（反馈信息获取与利用）

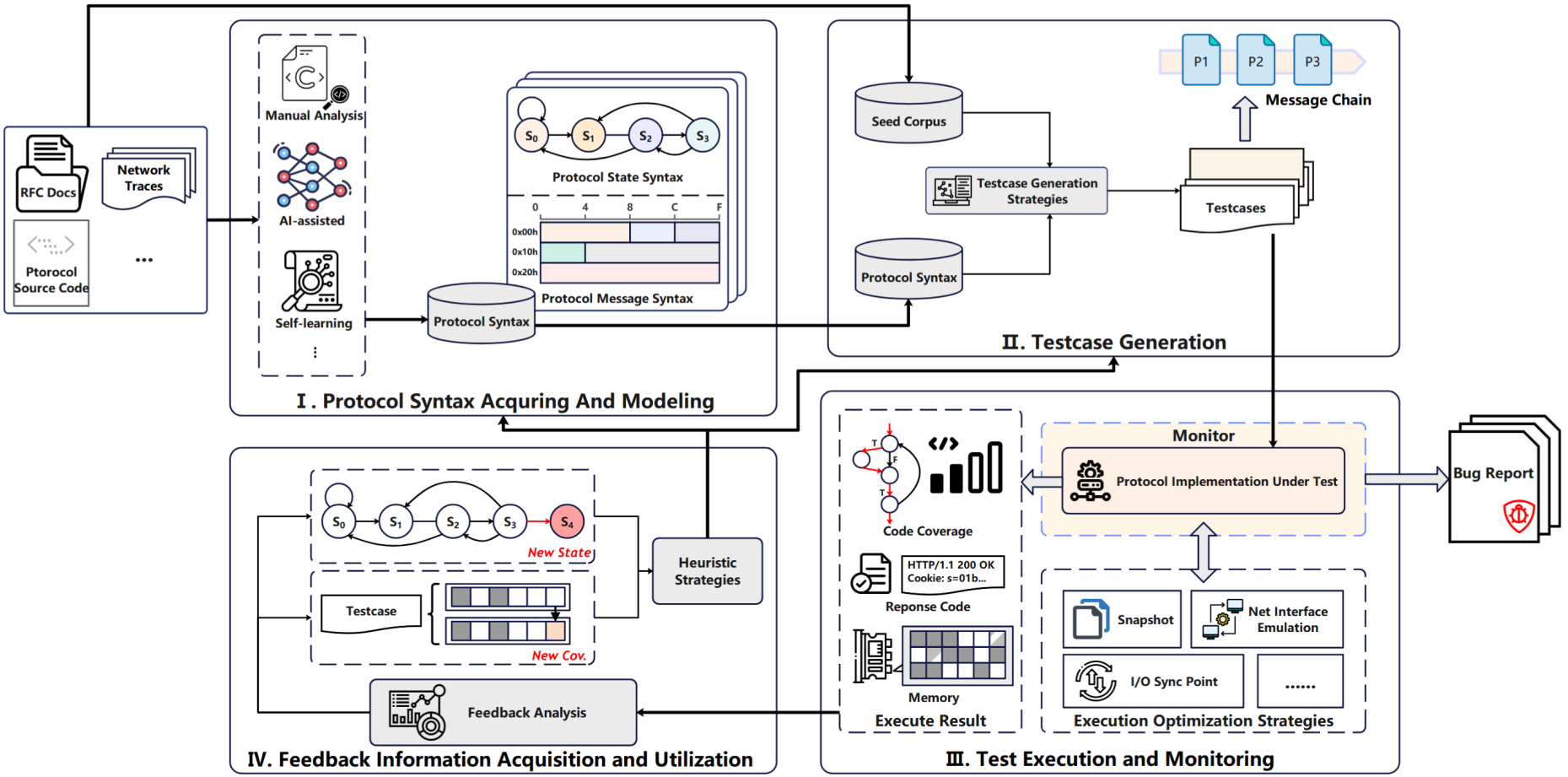
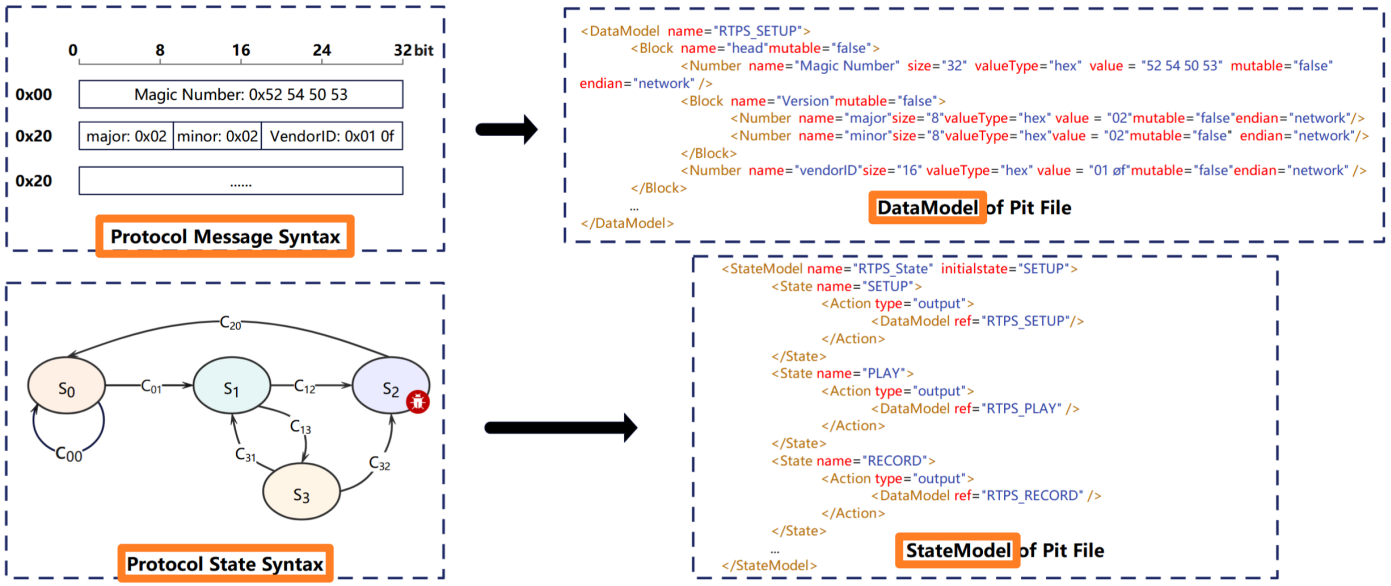


Figure 9: Unified Process Model for Network Protocol Fuzzing

协议语法获取与建模

- survey 统一框架：
  1. **protocol syntax acquisition and modeling（协议语法获取与建模）**
  2. testcase generation（测试样例生成）
  3. test execution and monitoring（测试执行与状态检测）
  4. feedback information acquisition and utilization（反馈信息获取与利用）
- protocol syntax acquisition and modeling（协议语法获取与建模）
  - 通过诸如网络流量、协议规范文档或协议源代码等信息源，对协议输入空间构建严格约束。
  - protocol message syntax and protocol state syntax（协议消息语法和协议状态语法）
    - protocol message syntax: 协议消息语法：用于在通信中传输数据和元数据
      规定消息中不同字段的划分、字段的组织顺序以及字段间的依赖关系（例如，某些字段的内容是其他字段的长度和校验和）。每个字段内部包含数据类型、长度和取值范围等属性。
    - protocol state syntax: 用于描述通信过程中系统的状态
      规定网络协议的状态集合及状态间转换的逻辑，通常由协议状态机描述。

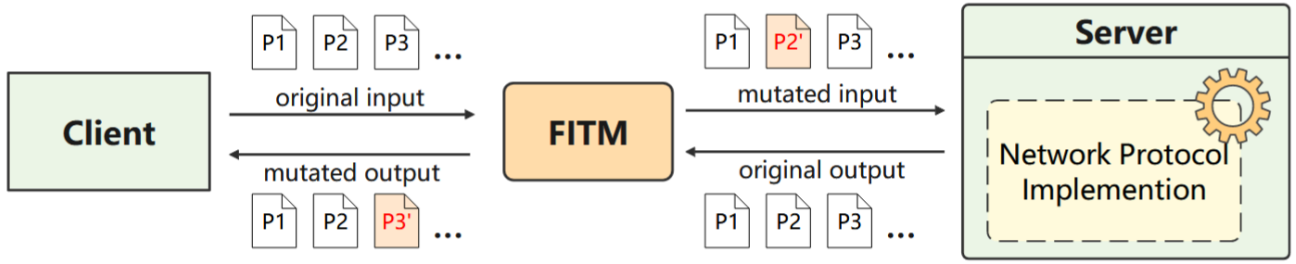


协议语法获取与建模

- survey 统一框架：
  1. **protocol syntax acquisition and modeling（协议语法获取与建模）**
  2. testcase generation（测试样例生成）
  3. test execution and monitoring（测试执行与状态检测）
  4. feedback information acquisition and utilization（反馈信息获取与利用）
- protocol syntax acquisition and modeling（协议语法获取与建模）
  - 通过诸如网络流量、协议规范文档或协议源代码等信息源，对协议输入空间构建严格约束。
  - protocol message syntax and protocol state syntax（协议消息语法和协议状态语法）
    - protocol message syntax: 协议消息语法：用于在通信中传输数据和元数据
      1. 手动获取：从官方协议文档、源代码和捕获的网络流量中手动提取协议语法
      2. 网络流量分析：通过捕获协议正常运行期间的网络流量来构建语料库,并结合各种启发式算法从中提取协议的不同字段、分隔符和其他消息语法
      3. 程序行为分析：通过白盒方法分析程序在处理消息数据时的具体行为,获得大量关于协议消息格式的信息
      4. AI 辅助学习：通过 NLP 技术实现自动协议消息语法学习系统
      5. 消息片段推理：逐字节变异要发送的消息并收集变异消息产生的响应,根据获得的响应的一致性将触发相同响应的字节组合成一个片段,每个片段对应协议中的特定代码执行路径
    - protocol state syntax: 用于描述通信过程中系统的状态
      1. 手动学习：手动分析网络协议的状态转换,编写语法规则（**Pit** 文件）描述转换+ **LLM**
      2. 主动推断：主动生成一系列测试消息并发送给被测程序(PUT)以获得相应的输出,同时使用模型学习算法来推断和构建目标协议的完整状态机
      3. 被动学习：通过样本（协议软件在正常行为下生成的网络流量和并行进行的模糊测试期间使用的测试用例）来学习状态转换逻辑

测试样例生成

- survey 统一框架：
  1. protocol syntax acquisition and modeling（协议语法获取与建模）
  2. **testcase generation（测试样例生成）**
  3. test execution and monitoring（测试执行与状态检测）
  4. feedback information acquisition and utilization（反馈信息获取与利用）
- testcase generation（测试样例生成）
  - 测试用例生成方法：
    1. 生成式（**generation-based**）：采用特定语法生成测试样例
       - 以配置文件的形式为用户提供标准接口，用户需要根据工具规范和协议语法为目标协议编写测试配置文件。配置文件通常包含某种形式的描述协议语法的数据结构，因此编写配置文件的过程实际上是从用户角度对协议语法进行建模（配置文件书写）
    2. 变异式（mutation-based）：使用各种变异操作符对用户提供的种子文件进行变异
       - 使用协议语法来指导测试用例的变异，利用学习到的语法规范来保护待测输入的结构在变异过程中不被破坏（种子文件质量，语法规范）
    3. FITM（fuzzer-in-middle）：充当中间人拦截客户端和服务器之间的通信流量-变异式的变种
       - 捕获服务器和客户端之间的网络流量，并有选择地对其进行篡改和重放。需要专门的工具和技术来拦截和修改协议软件之间的通信。（详尽的协议语法规范）
    4. 生成式（generation-based）+变异式（mutation-based）
       - 将生成式和变异式两者结合，基于配置文件生成基础测例，再基于协议语法指导变异



测试执行与状态检测

- survey 统一框架：
  1. protocol syntax acquisition and modeling（协议语法获取与建模）
  2. testcase generation（测试样例生成）
  3. **test execution and monitoring**（测试执行与状态检测）-优化属性
  4. feedback information acquisition and utilization（反馈信息获取与利用）
- test execution and monitoring（测试执行与状态检测）-优化属性
  - 测试执行：被测程序（PUT）接收并执行由模糊器生成的测试用例。
  - 状态检测：监控器通过监视 PUT 的运行状态来感知 bug 的触发
  - 测试执行过程给协议模糊器带来了巨大的性能负担，严重影响了模糊测试的效率：
    - 协议软件通过网络接口传输数据；
    - 多线程服务器进程通常具有更高的启动时间成本；
    - 测试状态恢复时间成本
  - 测试执行解决方法：
    1. （**SnapShot**）快照：存储特定时刻操作系统或进程在物理内存和各种设备中状态的静态副本文件。通常，有系统快照、虚拟机快照、文件系统快照、进程级快照等（**FITM**）
    2. （Network Function Replacement or Emulation）网络功能替换或模拟：通过替换网络功能 API、文件系统 API 等，使用更高效的接口函数或自定义模拟方法（Desock+，SnapFuzz，FITM）
    3. （Protocol I/O Synchronisation Points）协议 I/O 同步点：基于协议事件循环设置 I/O 同步点以加速测试执行，标记输入消息时间点（NSFuzz）
    4. （Sending a sequence of messages in one go）发送消息序列：减少模糊测试过程中模糊器和 SUT 之间的上下文切换（context-switches）开销。

反馈信息获取与利用

- survey 统一框架：
  1. protocol syntax acquisition and modeling（协议语法获取与建模）
  2. testcase generation（测试样例生成）
  3. test execution and monitoring（测试执行与状态检测）
  4. **feedback information acquisition and utilization（反馈信息获取与利用）**
- feedback information acquisition and utilization（反馈信息获取与利用）
  - 软件系统产生的输出中包含的一类可用属性
  - 反馈信息分类：
    1. 响应代码(状态机违反)：状态响应代码确保客户端的请求得到确认，并通知客户端当前服务器的状态。通过观察状态响应代码或从响应中提取的一些信息，可以推断系统的状态轨迹
    2. 覆盖率：覆盖率包括代码覆盖率、函数覆盖率、分支覆盖率、路径覆盖率、状态覆盖率等，通常应用于灰盒模糊器。模糊器对被测试程序进行插桩，并使用位图记录测试用例执行期间的路径、分支等覆盖情况
    3. 分支：用于描述程序执行的特定位置或时刻。它可能指代代码中的特定行、函数调用的位置、发生特定状态或事件的位置等。分支反馈需要用户手动注释代码。用户通过观察来手动标记那些更可能触发漏洞的分支为感兴趣的分支
    4. 变量：程序变量指的是用于存储数据的标识符。变量可以包含各种类型的数据,如数字、字符串、布尔值等。通过观察这类过程变量来获取协议状态的信息,并更新和维护协议状态机以提高模糊测试的有效性。
    5. 内存：在计算机系统或电子设备中用于控制内存（如 RAM、ROM 等）操作和访问的电信号或信号组合。它们控制数据的读取、写入和处理。通过对内存区域进行快照并比较相关信息，来观察当前系统输入是否对这些内存区域产生影响