RFCDiff

网络驱动的 RFC 文档匹配与不一致查找

谢远峰，王跃林，徐一洋

天津大学-智能与计算学部-网络安全

2024-09-04

1. 场景选择
2. 参考工作梳理
3. 网络协议测试挑战
4. 网络协议测试统一框架
5. TLS 协议测试框架
6. RFC 规则构建
7. CVE 样例详解

1. 场景选择

目标场景

- **SSL/TLS 协议实现 RFC 合规性模糊测试**
  - 协议：TLS(1.0,1.2,1.3)-公钥加密
  - 目标：找出 TLS 代码实现和 RFC 文档（TLS1.3-RFC8446）的不一致
    1. RFC 中使用强语气词（如"must"、"required"等）来描述实现协议时必须遵守的要求或事项
- motivation example:RFC 规则合规性样例说明
  1. 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).** There MUST NOT be more than one extension of the same type in a given extension block.
  2. 处理 message 的 target: C:\Windows\System32\schannel.dll–CVE-2023-2823*
  3. 基于 **RFC** 文档构建状态机和消息语法，构建的消息文法和状态机以及消息所处的状态作为生成根据，抽取强语气词规则作为变异根据，选取 **windows** 驱动作为目标，预设状态机违反作为反馈
- 挑战
  - 测试用例构造：了解协议状态转换和消息序列的流程
  - 状态维护：确保接收方（被测试的协议实现）能够维持在期望的状态接收测试
  - 加密会话解析：不能仅依靠 RFC 生成会话消息，依赖于协议具体实现
  - 异常行为检测：违反 RFC 规则不会导致明显的错误，需要开发更加敏感和准确的方法来检测协议实现中的异常行为

2. 参考工作梳理

工具现状梳理

- Survey:
  - **A Survey of Network Protocol Fuzzing: Model, Techniques and Directions**
    - On the (in)efficiency of fuzzing network
  - **Fuzzers for Stateful Systems: Survey and Research Directions**
  - Survey of Code Search Based on Deep Learning
  - Large Language Models Based Fuzzing Techniques: A Survey
- Baseline:
  - AFLNET: A Greybox Fuzzer for Network Protocols
  - HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations
- LLM application
  - State Machine
    - (ProtocolGPT)Inferring State Machine from the Protocol Implementation via Large Language Model
    - PROSPER: Extracting Protocol Specifications Using Large Language Models
  - Message Syntax
    - MSFuzz: Augmenting Protocol Fuzzing with Message Syntax Comprehension via Large Language Models
    - Large Language Model guided Protocol Fuzzing
  - Program Static Analysis
    - GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis
  - Fuzz Driver
    - How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation
- TLS
  - **DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing**

3. 网络协议测试挑战

- 参考：
  1. A Survey of Network Protocol Fuzzing: Model, Techniques and Directions（arxiv-2024.2.）
  2. Fuzzers for Stateful Systems: Survey and Research Directions（CCS 2024.4. ）
  3. On the (in)efficiency of fuzzing network protocols（2024.8）
- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战条目）
  - Reliance On Network Links(依赖于网络链接)
  - Statefulness: Network protocol software is typically stateful.(网络协议软件通常是状态化的)
  - the highly structured nature of inputs（测试输入的高度结构化）
  - Non-uniformity.（网络协议无泛化性，无法用状态机表示）

依赖于网络链接

- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战条目）
  - **Reliance On Network Links(依赖于网络链接)**
  - Statefulness: Network protocol software is typically stateful.(网络协议软件通常是状态化的)
  - the highly structured nature of inputs（测试输入的高度结构化）
  - Non-uniformity.（网络协议无泛化性，无法用状态机表示）
- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战）
  1. Reliance On Network Links(依赖于网络链接)
     - 挑战描述：网络协议处理的网络流量是随着时间依次到达的，数据包是这些流量中的最小单位。模糊测试工具需要构建数据包并通过网络接口将其传输到协议软件。由于依赖网络链接，协议模糊测试工具必须能够使用标准网络接口来发送和接收网络数据包。
     - 工具现状：大多数现代协议模糊测试工具都具备发送和接收网络数据包的能力。

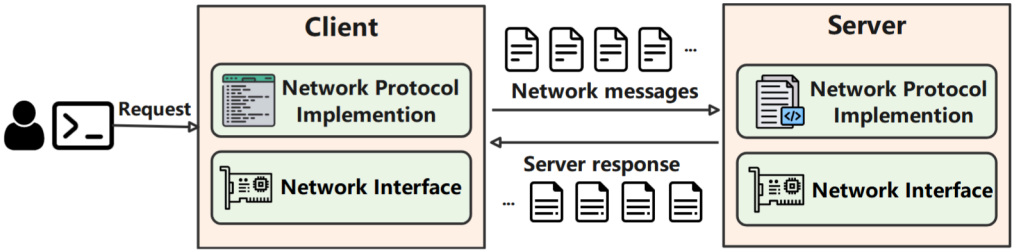


Figure 1: 网络协议软件交互过程

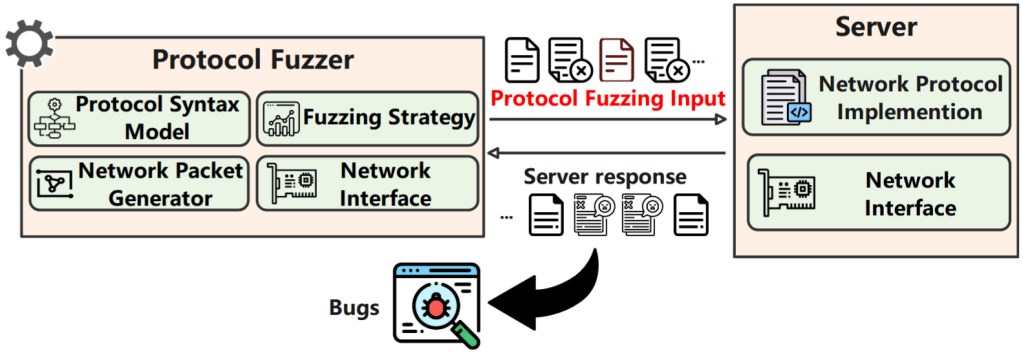


Figure 2: 网络协议软件 server 端模糊测试过程

网络协议软件状态化

- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战条目）
  - Reliance On Network Links(依赖于网络链接)
  - **Statefulness: Network protocol software is typically stateful.(网络协议软件通常是状态化的)**
  - the highly structured nature of inputs（测试输入的高度结构化）
  - Non-uniformity.（网络协议无泛化性，无法用状态机表示）
- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战）
  2. Statefulness: Network protocol software is typically stateful.(网络协议软件通常是状态化的)
     - 挑战描述：网络协议在通信过程中处于确定的状态，并在接收到特定消息时改变状态。在不同软件状态下，相同的输入可能导致不同的程序行为。
     - 工具现状：常用的方法是有限状态机来描述状态及其转换逻辑。协议交互的输入必须是严格按时间顺序的消息序列。
     - 为了模糊测试协议软件的庞大状态空间，模糊测试工具需要精心构建消息序列。首先通过传输相关前缀消息将协议软件转到目标状态，然后发送随机生成的数据包来模糊测试该状态。

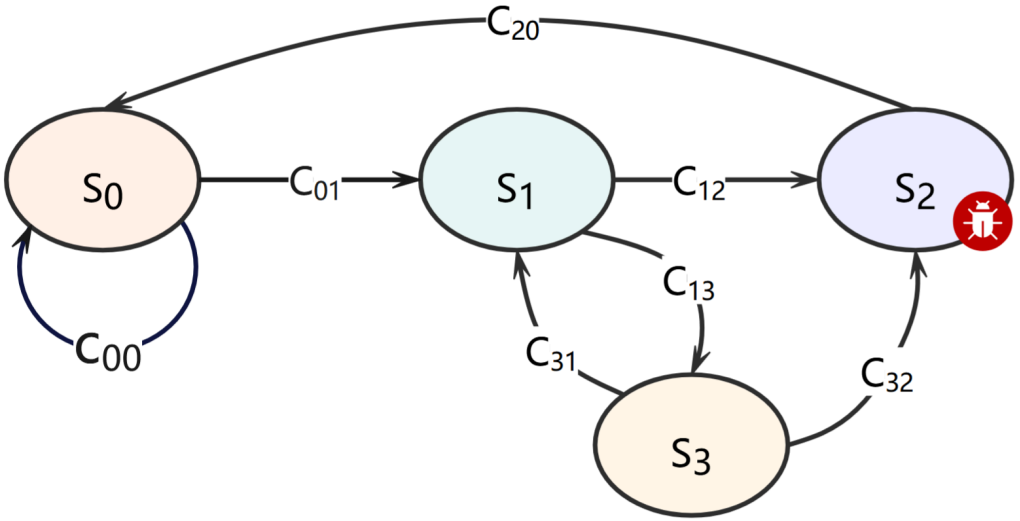


Figure 3: 有限状态机，S_0状态接收消息C_{01}后进入S_1状态

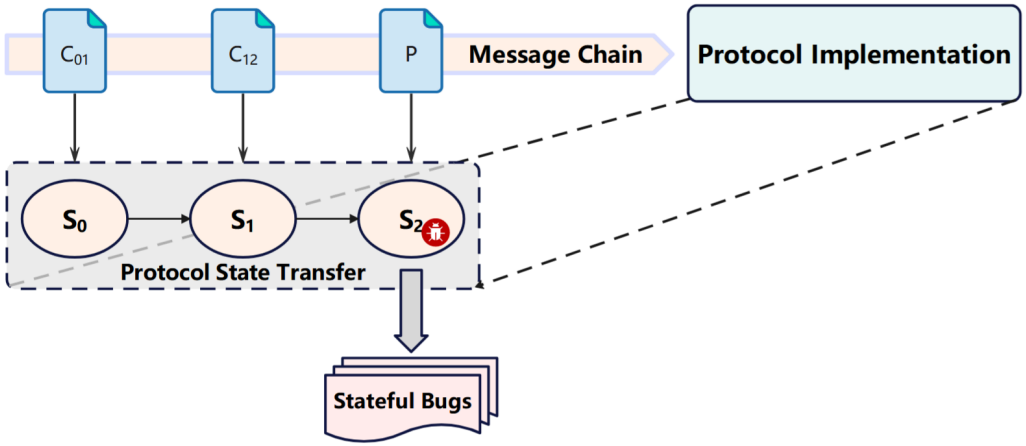


Figure 4: 状态 bug, 服务端接收消息 P 后触发到 S2 bug

测试输入的高度结构化

- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战条目）
  - Reliance On Network Links(依赖于网络链接)
  - Statefulness: Network protocol software is typically stateful.(网络协议软件通常是状态化的)
  - **The highly structured nature of inputs（测试输入的高度结构化）**
  - Non-uniformity.（网络协议无泛化性，无法用状态机表示）
- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战）
  3. the highly structured nature of inputs（测试输入的高度结构化）
     - 挑战描述：消息可以分为具有严格语法约束的位或字节字段，每个字段都有明确的类型和有效值范围。如果违反基本语法约束，如校验和字段中缺少预期的固定字节，协议软件将丢弃数据包并终止连接。这使得模糊测试工具难以发现深层次的程序漏洞。
     - 工具现状：对输入结构敏感的模糊测试工具在协议模糊测试中通常表现更好。

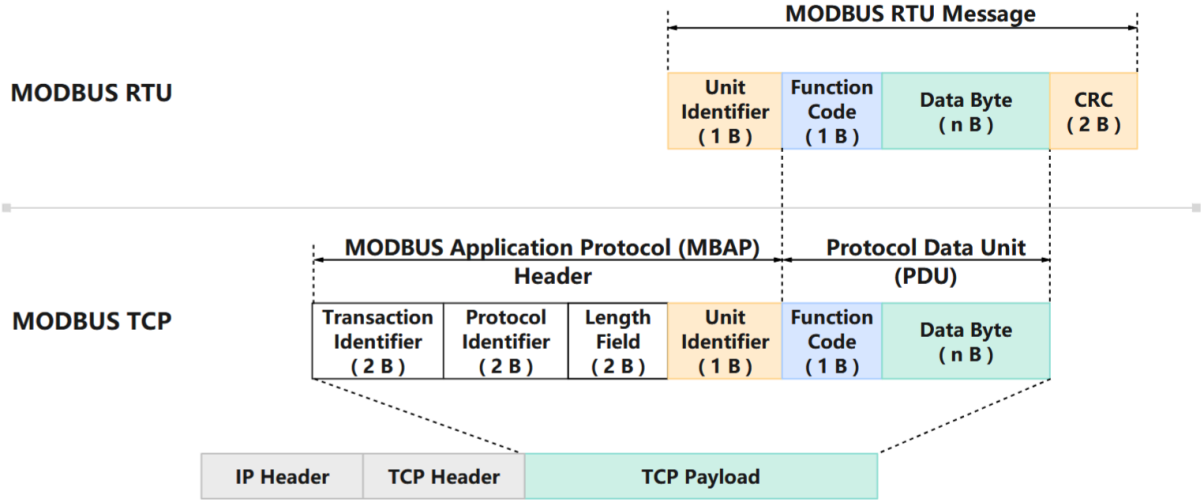


Figure 5: modbus 协议的部分构成

无泛化性

- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战条目）
  - Reliance On Network Links(依赖于网络链接)
  - Statefulness: Network protocol software is typically stateful.(网络协议软件通常是状态化的)
  - the highly structured nature of inputs（测试输入的高度结构化）
  - **Non-uniformity.（网络协议无泛化性，无法用状态机表示）**
- network protocol fuzzing unique challenge（网络协议模糊测试特有挑战）
  3. Non-uniformity.（网络协议无泛化性，无法用统一状态机表示）
     - 挑战描述：不同的协议在消息语法和状态机上缺乏统一性，不同的协议可能具有完全不同的消息语法和内部状态机(http/tls)，从而限制了协议模糊测试工具的通用性
     - 工具现状：添加额外的处理来确保可以进行模糊测试的协议的覆盖

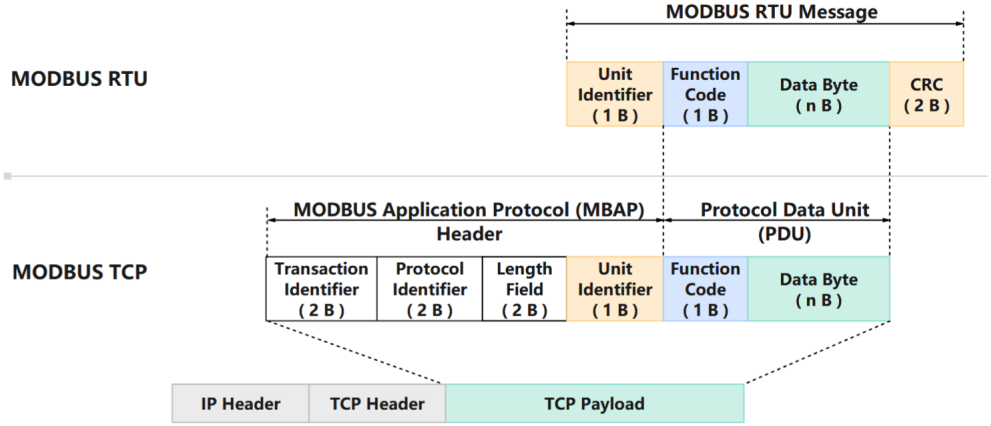


Figure 6: modbus 协议的部分构成

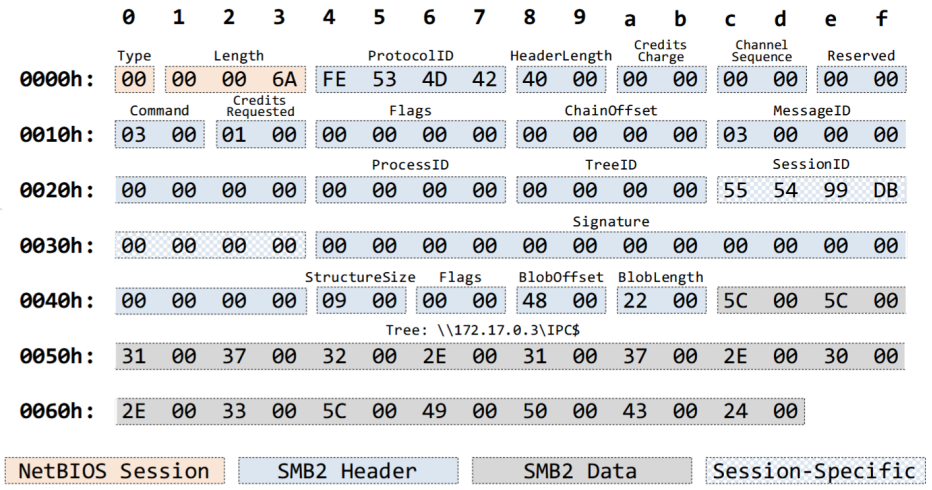


Figure 7: SMB2 request 消息构成

4. 网络协议测试统一框架

- 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）

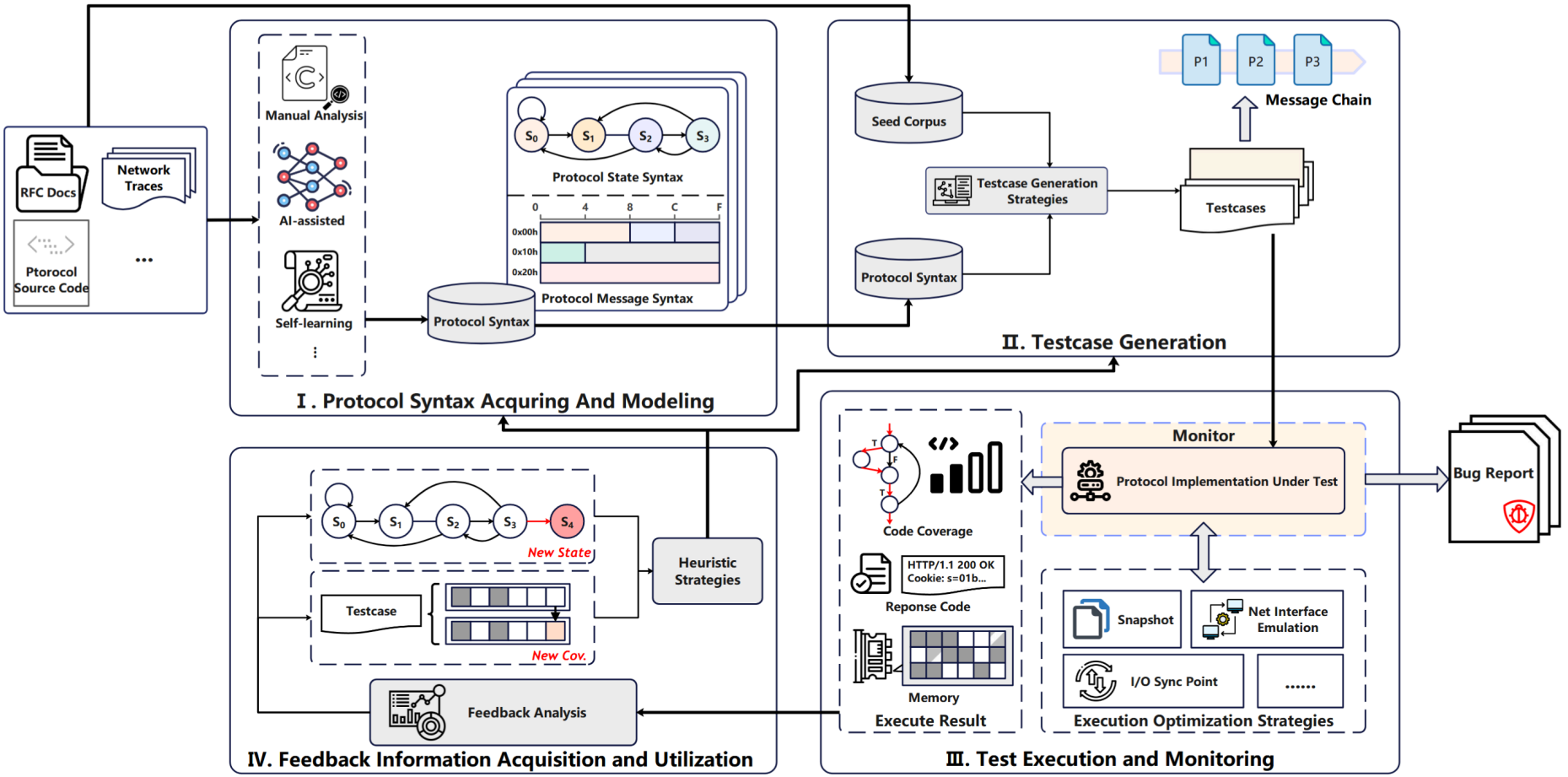
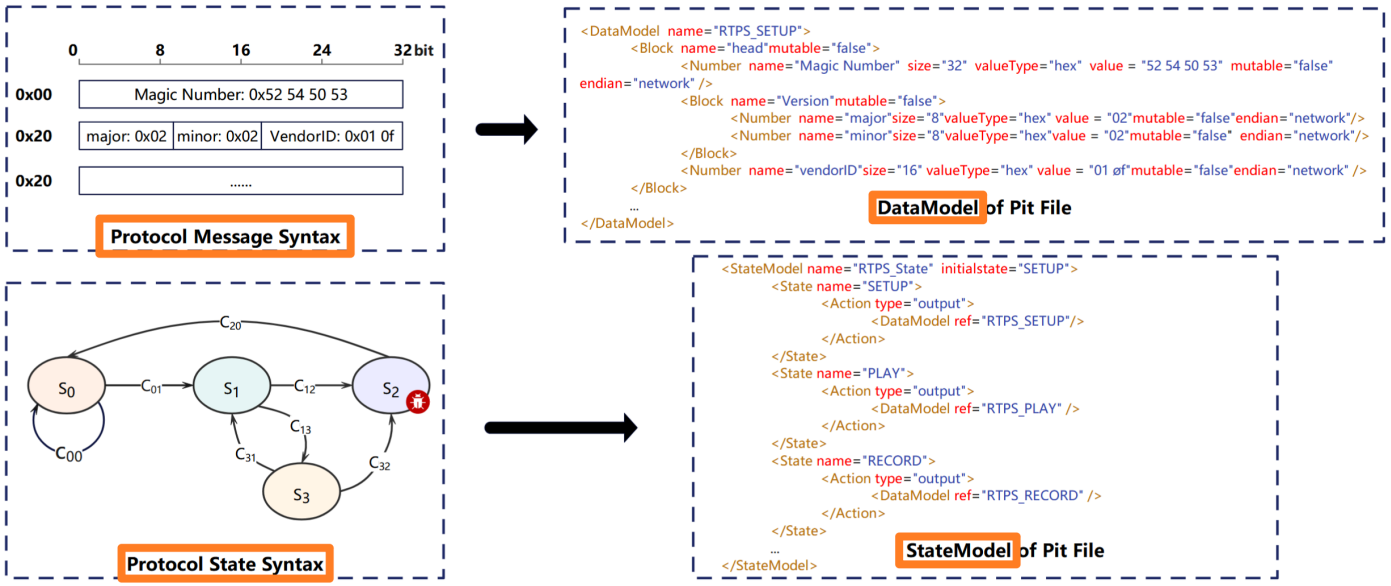


Figure 8: Unified Process Model for Network Protocol Fuzzing

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。

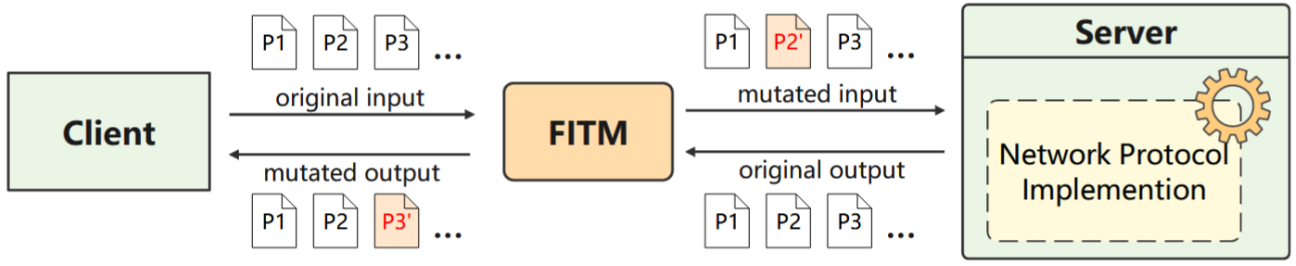


协议语法获取与建模

- 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：状态响应代码用于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
    6. 状态机违反：违反了预定义状态机状态转换过程，出现未定状态或行为

- 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）
- 已有工具现状
  1. Pre-processing of raw network traffic（网络流浪原始数据的预处理）
     利用原始网络流量数据获取协议的消息语法,以此构建消息流/基于正常网络流量数据变异
  2. Using snapshots（快照的使用）
     快照可以加速测试速度，使程序快速恢复到指定状态（系统快照、虚拟机快照、进程快照等）
  3. Mutation primitives and heuristics （突变策略和启发式方法）
     （消息结构内容变异(位变异)，消息序列变异(增删重复)；状态机违规，响应代码等）
- 自身工作的一些思考
  - 挑战一：协议的状态性（确保消息的正确性和方向性-protocol fuzzing general challenge）☒
  - 挑战二：确保测试程序维持特定状态-protocol fuzzing general challenge（快照使用）☒
  - 挑战三：消息的构造高度依赖上下文（正确解析+构造-加密协议特有）☑
  - 挑战四：检测异常行为（反馈的设置-protocol fuzzing general challenge）☑☒

5. TLS 协议测试框架

DY Fuzzing 框架

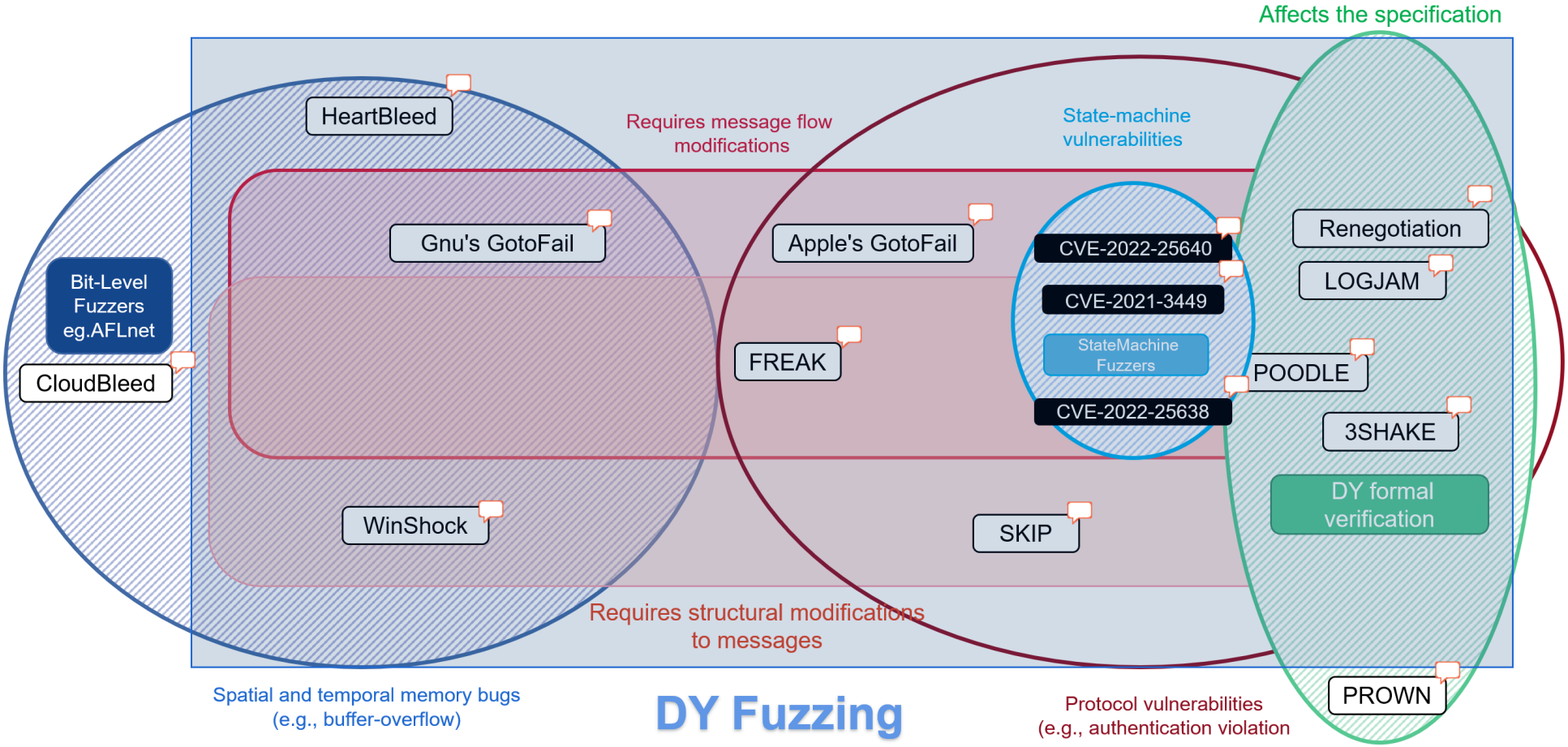
- 参考内容（2024 S&P）
  - **DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing**
  - 论文讲解：Secure Cryptographic Protocols Bug Hunt
- 前置概念：DY Attack
  - 基于 Dolev-Yao 攻击者模型的一类攻击。这个模型由 Danny Dolev 和 Andrew Yao 在 1980 年代提出，它描述了一个在网络中具有强大能力的主动攻击者
  - DY 攻击者能力：拦截、读取、修改、伪造、重放、延迟
  - 特点：将消息视为符号化的术语而非比特串、无法破解加密算法且只能通过已知的密钥进行解密、可以利用密码学操作的代数属性（加解密的互逆操作）
  - 类型：中间人攻击，重放攻击，类型混淆攻击，并行会话攻击，反射攻击
  - 应用领域：密码协议分析、安全验证、漏洞发现
- TLS 协议核心：
  - 身份验证：服务器始终进行身份验证，客户端可选择性地进行身份验证。身份验证通过非对称加密（如 RSA）或预共享密钥（PSK）实现。
  - 完整性和机密性：应用数据始终使用会话密钥进行加密和完整性保护。
- TLS1.3 协议组成：
  - 握手协议：用于协商密码套件、验证终端身份并建立共享的会话密钥
  - 记录层协议：利用建立的安全通道（基于会话密钥）交换应用数据
- 目标：
  - OpenSSL 是使用最广泛的 TLS 实现，拥有近 25 年的历史。
  - LibreSSL 是 OpenSSL 的一个分支，旨在提高安全性但功能较少。
  - wolfSSL 是一个轻量级实现，广泛用于物联网和嵌入式设备，能够在其他不支持的操作系统和 CPU 上运行。

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

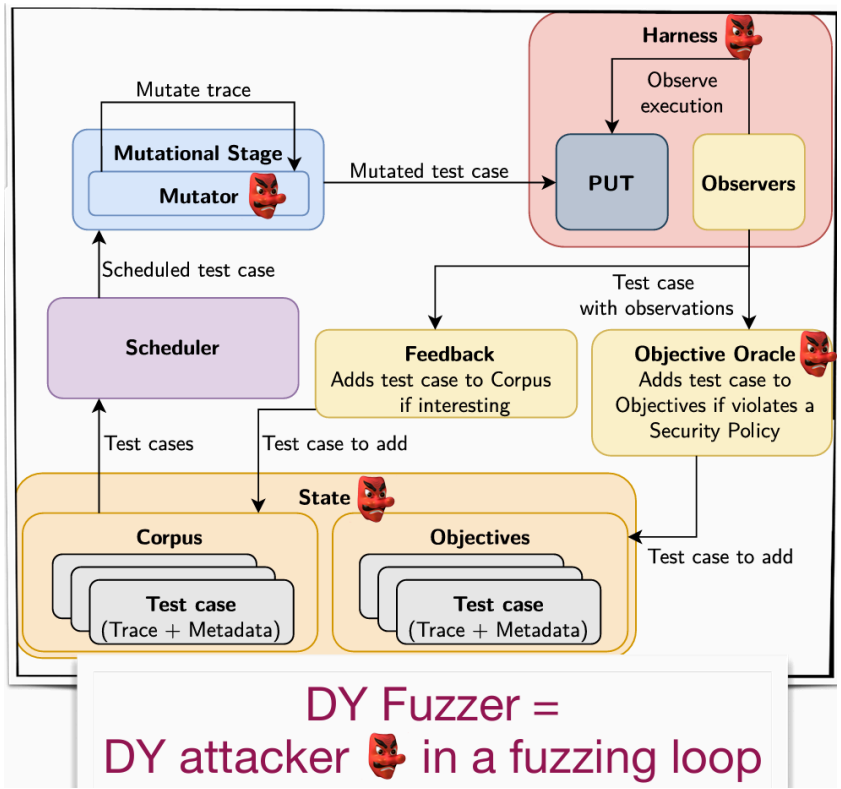
已有模糊测试工具的局限

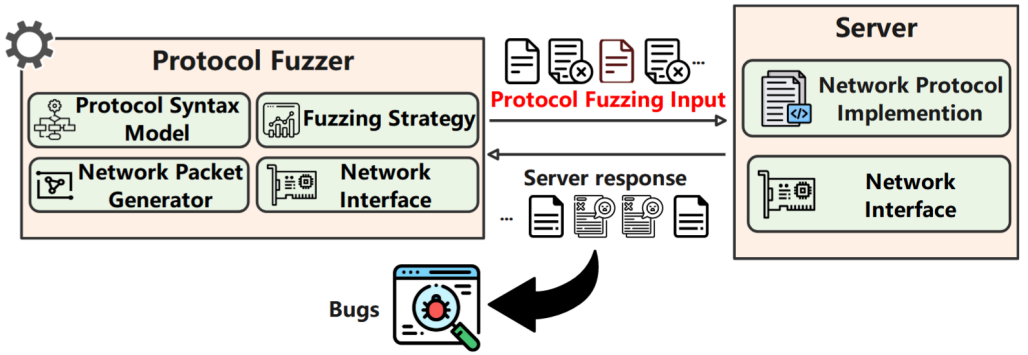
- Fuzzing protocols at the bit-level（比特级模糊测试工具）
  1. 可达性问题：无法处理多轮交互；难以实现复杂逻辑变异；难以达到触发 DY 攻击的深层状态
  2. 检测问题：主要专注于内存错误；无法检测协议级别的漏洞（如身份验证绕过）
- Model-based protocols fuzzing（基于状态的协议模糊测试工具）
  - 模型局限性：使用有限状态机(FSM)抽象协议行为，但不够全面
  - 攻击覆盖不足：可检测特定类型攻击（如身份验证绕过）；无法全面捕获 DY 攻击
  - 消息变异能力不足：不能有效篡改消息内容；仅能修改有限的预定义值
  - 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，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。

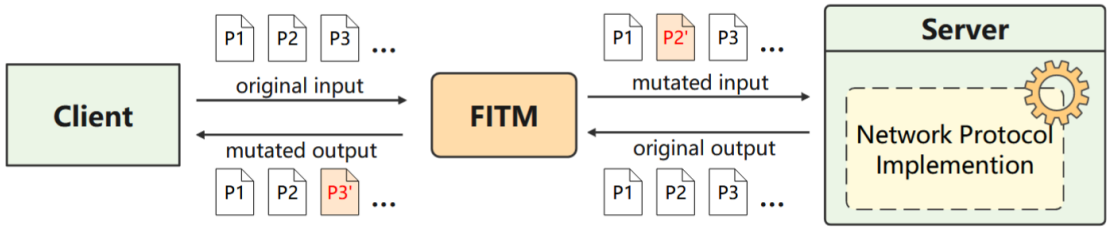


除了CloudBleed（在边缘服务器上），其他漏洞（如内存溢出、权限提升）可模拟检测

DY Fuzzing 框架构建

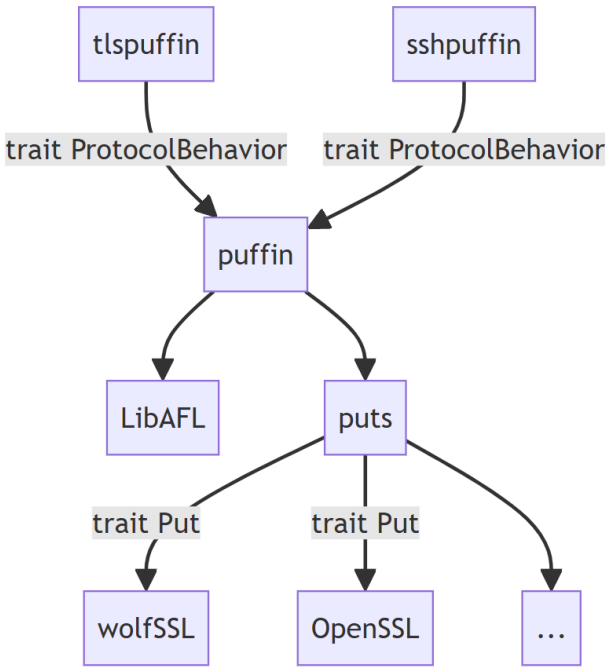


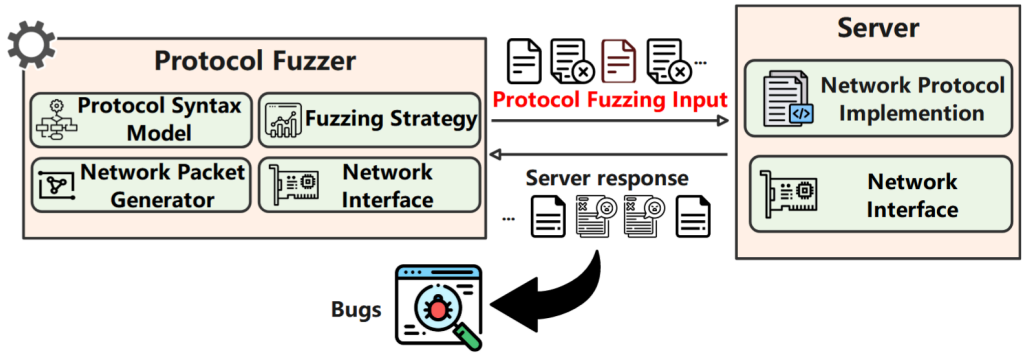


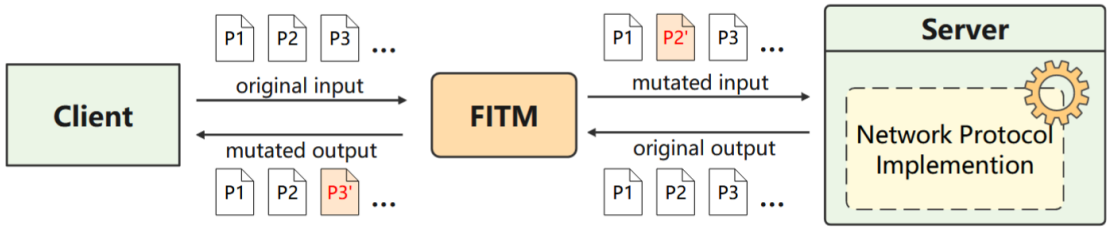


- State: 由"语料库"和"目标"组成，存储所有测试用例及其跟踪信息和元数据。其中，Objectives 特别关注那些触发了安全违规的用例。
- Scheduler: 根据各种策略性标准，从语料库中选择并安排测试用例进行变异和重新测试。
- Mutational Stage: 修改测试用例的 trace，创建一个变异后的测试用例，然后将其发送至测试框架。
- Harness: 执行变异后的测试用例，在被测程序中运行并观察其执行过程。(server, client, MITM)
- Feedback: 评估测试用例执行的观察结果，将有趣的用例添加到语料库中以进行进一步测试。
- Objective Oracle: 检查测试用例是否违反安全策略，将违反策略的用例添加到 Objectives 重点分析

tlspuffin: TLS 协议模糊测试工具







- puffin - Core fuzzing engine which is protocol agnostic.
- tlspuffin - TLS fuzzer which uses puffin and is implementation agnostic.
- sshpuffin (WIP) - SSH fuzzer which uses puffin and is implementation agnostic.
- puts - Linkable Programs Under Test that can be linked with tlspuffin or sshpuffin.

1. State: 由"语料库"和"目标"组成，存储所有测试用例及其跟踪信息和元数据。其中，Objectives 特别关注那些触发了安全违规的用例。
2. Scheduler: 根据各种策略性标准，从语料库中选择并安排测试用例进行变异和重新测试。
3. Mutational Stage: 修改测试用例，创建一个变异后的测试用例，然后将其发送至测试框架。
4. Harness: 执行变异后的测试用例，在被测程序中运行并观察其执行过程。(server, client, MITM)
5. Feedback: 评估测试用例执行的观察结果，将有趣的用例添加到语料库中以进行进一步测试。
6. Objective Oracle: 检查测试用例是否违反安全策略，将违反策略的用例添加到 Objectives 重点分析

- 已有工具现状
  1. Pre-processing of raw network traffic（网络流浪原始数据的预处理）
     利用原始网络流量数据获取协议的消息语法,以此构建消息流/基于正常网络流量数据变异
  2. Using snapshots（快照的使用）
     快照可以加速测试速度，使程序快速恢复到指定状态（系统快照、虚拟机快照、进程快照等）
  3. Mutation primitives and heuristics （突变策略和启发式方法）
     （消息结构内容变异(位变异)，消息序列变异(增删重复)；状态机违规，响应代码等）
- tlspuffin: TLS 协议模糊测试工具
  1. DY 模糊测试框架: 针对加密协议的全新模糊测试方法，首次成功捕捉了 DY 攻击者模型和 DY 攻击类别漏洞。提出了一个全新且完整的系统设计。
  2. tlspuffin：这是一个用 Rust 语言实现的功能完备、模块化且高效的模糊测试工具，它完全遵循我们提出的设计理念
  3. 对多个 TLS 1.2 和 TLS 1.3 库使用 tlspuffin 进行了评估和比较，(重新)发现了七个其他方法未能发现的漏洞，其中包括四个新发现的漏洞（一个严重级别，两个高级别，一个中级别）
- 自身工作的一些思考
  - 挑战一：协议的状态性（确保消息的正确性和方向性-protocol fuzzing general challenge）☒
  - 挑战二：确保测试程序维持特定状态-protocol fuzzing general challenge（快照使用）☒
  - 挑战三：消息的构造高度依赖上下文（正确解析+构造-加密协议特有）☑☒
  - 挑战四：检测异常行为（反馈的设置-protocol fuzzing general challenge）☑☒
- 下一步规划：
  - 细读论文(形式化说明和建模部分)，关注论文的不足
  - 关注消融实验，说明工具有效性

工具有效性验证-part1

- 捕获漏洞的种类：逻辑漏洞
- 变异策略消融实验(same seeds, harness, and detection capabilities (only ASAN))
  1. 消息流修改（skip,repeat）, 消息结构修改（generate,...）
  2. “all"列显示了启用所有突变时的基准性能。
  3. 红色"X"表示在禁用某种突变时无法发现相应的漏洞。
  4. 红色数字突出了某些情况下禁用特定突变导致发现漏洞所需的程序执行次数显著增加
  5. 有些突变的禁用对某些漏洞的发现影响较大，而对其他漏洞影响较小。

	all	*Skip*	*Repeat*	*Swap*	*Generate*	*Replace-Match*	*Replace-Reuse*	*Remove-and-Lift*
SDOS1	2,3E+07	✗	✗	2,4E+07	3,6E+06	1,6E+07	1,7E+07	2,1E+07
SDOS2	6,3E+03	8,7E+03	4,0E+06	2,7E+04	2,4E+03	**1,2E+04**	2,0E+03	4,4E+03
BUF	9,8E+04	6,1E+04	3,4E+04	9,1E+04	**2,8E+05**	1,1E+05	5,1E+04	3,5E+04
HEAP	1,3E+05	1,9E+05	4,5E+04	**2,0E+05**	1,1E+05	**1,9E+05**	**3,0E+05**	1,8E+04

- 覆盖率对照实验
  - 对照目标（AFLnwe,StateAFL,AFLNET）
  - 分支覆盖率，函数覆盖率，代码行覆盖率超过已有工具（wolfSSL-5.3.0）
    - 种子设置（基于变异，tls1.3,tls1.2 消息流，（无加密，作为 server，client，中间人）
    - 分支覆盖度：（tlspuffin 8000/ available 5500）+45%
    - 函数覆盖度：（tlspuffin 1000/ available 900）+11.1%
    - 代码行覆盖度：（tlspuffin 19000 / available 16000）+18.8%
  - 与针对 TLS 库的组合测试集(人工构建)对比: 仍有差距，仍有改进空间

	OpenSSL		*wolfSSL*	
	TLS-Anvil	*tlspuffin*	TLS-Anvil	*tlspuffin*
l_p	25.8%	19.9%	29.0%	24.2%
l_a	28074	21664	23421	19581
b_p	20.2%	16.0%	19.9%	16.9%
b_a	12363	9806	9434	8025
f_p	30.1%	23.9%	31.9%	27.0%
f_a	2370	1885	1225	1037

工具有效性验证-part2

- Oracle
  - Memory-related(ASAN-内存相关)
  - Logic-related(逻辑错误,逻辑攻击进行形式化)在 TLS 中,这可能对应于以下场景:
    - 安全属性定义：当一个代理(由 pk 标识)相信它已经成功地与代理 pk′就 m 达成一致时,代理 pk′确实先前就 m 开始了与 pk 的会话。
    - 满足属性的正常情况:在正常的 TLS 握手中:
      - 客户端发送 ClientHello,产生 Run(pkClient, pkServer, clientRandom)声明
      - 服务器完成握手后,产生 Agr(pkServer, pkClient, sessionData)声明
      - 这满足了属性,因为对于每个 Agr 声明,都存在一个先前的相应 Run 声明。
    - 违反属性的情况:假设一个恶意攻击者能够欺骗服务器,使其认为它已经与一个特定的客户端完成了握手,而实际上该客户端从未参与这个会话。在这种情况下:
      - 服务器会产生一个 Agr(pkServer, pkClient, sessionData)声明
      - 但客户端从未产生相应的 Run(pkClient, pkServer, sessionData)声明
      - 这将违反上述属性,因为存在一个 Agr 声明,但没有相应的先前的 Run 声明。
  - 当客观预言机检测到违反这种属性的情况时,它会将相应的测试用例标记为潜在的攻击。这可能表明存在中间人攻击、会话重放攻击或其他破坏 **TLS** 预期安全属性的漏洞。
- 设计的工具是否有效找到漏洞（漏洞探究）
  - （[SDOS1、SIG、SKIP]、SDOS2）、（CDOS、BUF、HEAP）
  - 大多数漏洞针对服务器端，只有 CDOS 针对客户端。漏洞目标主要集中在 WolfSSL 上
  - ClientHello 消息是最常见的攻击载体，可能因为它是握手过程中的第一条消息，包含了大量可配置的选项。
  - 多数攻击由恶意客户端发起，反映了客户端在 TLS 协议中的主动角色。
  - 消息扩展的处理是许多漏洞的关键点，特别是与签名算法和密钥共享相关的扩展。
  - 一些漏洞（如 SDOS2 和 BUF）涉及会话恢复机制(易出错的复杂功能)
  - SKIP、SDOS2、CDOS、BUF、HEAP 漏洞在运行几十分钟后找到，SDOS1、SIG 漏洞 24H 内找到

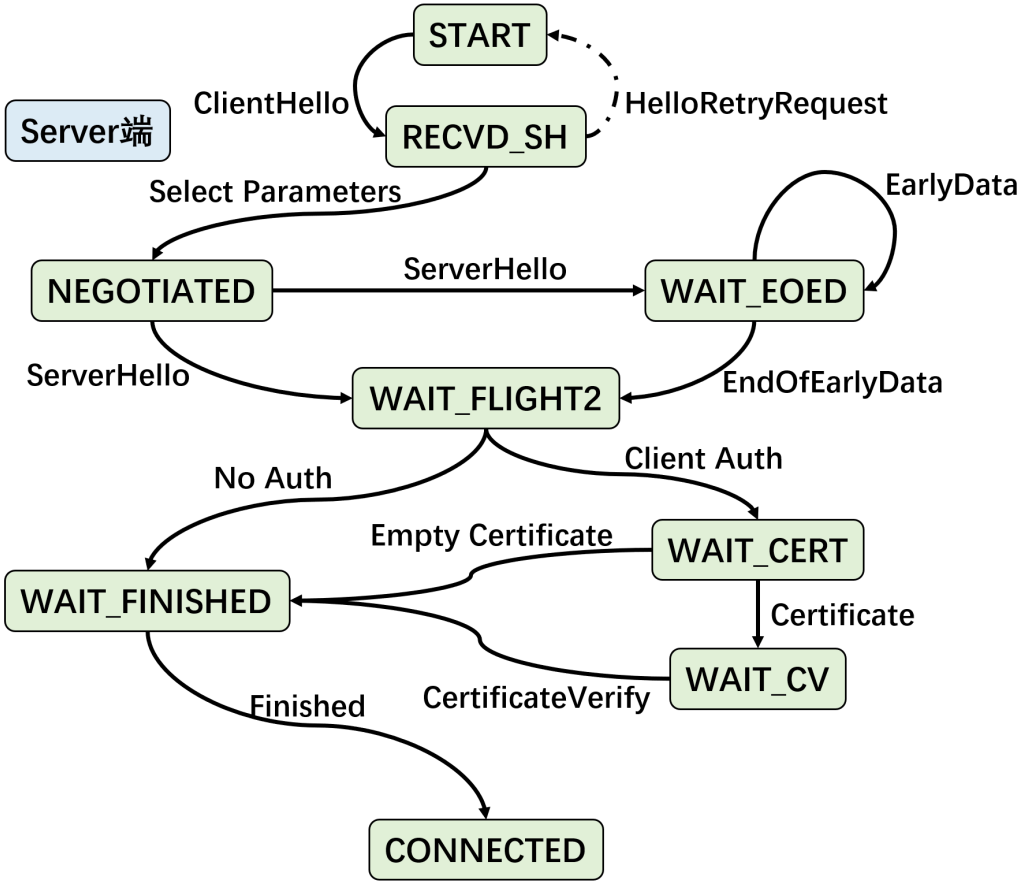
6. RFC 规则构建

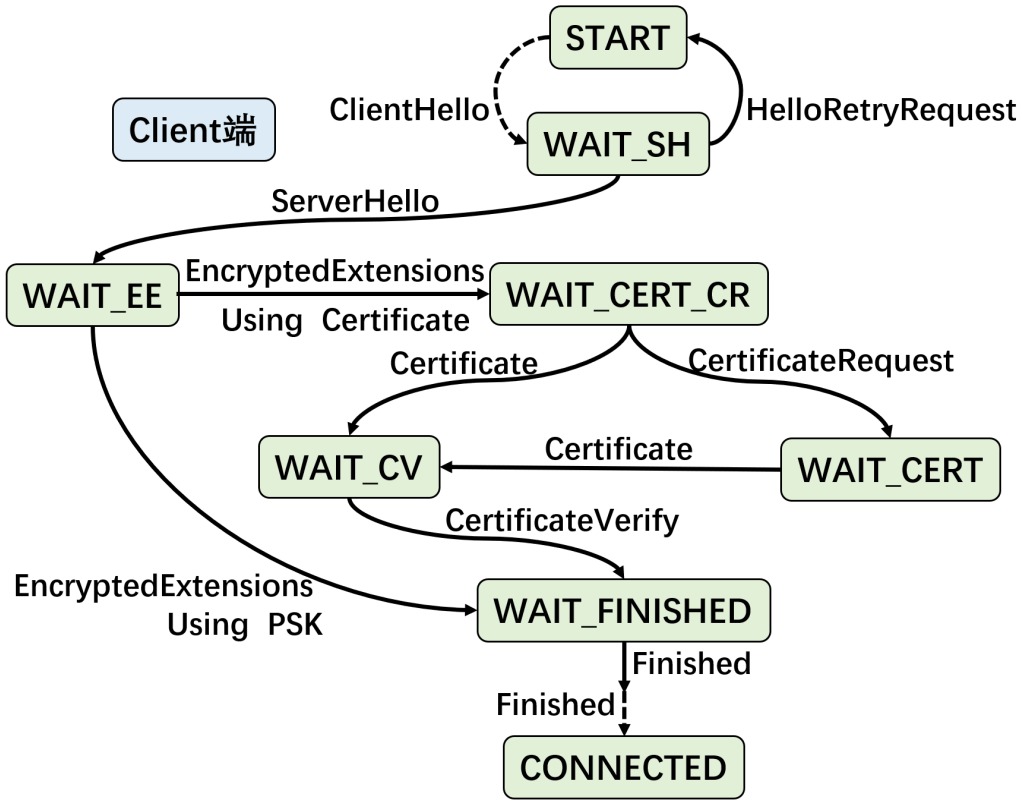
RFC 广义规则定义

- HDiff 中对于规则定义
  - 显式规则：情感词（must, is required, only）；隐式规则：条件语句（if）,强调词（In particular）
- 当前工作：
  - 针对终端进行分类：client, server, both
  - 针对消息类型进行分类：handshake/record/alert protocol：clienthelle/serverhello/...
  - 构造的规则：状态语法规则，消息语法规则(伪代码定义)
- protocol syntax acquisition and modeling（协议语法获取与建模）
  - 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+ **LLM**
      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
      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

TLS1.3 协议状态机构建-状态语法规则

- 前提条件 1：Regular Protocol- constraint-enhanced regular expressions（http,udp,tls）
- 前提条件 2：Already have state machine description (tls)





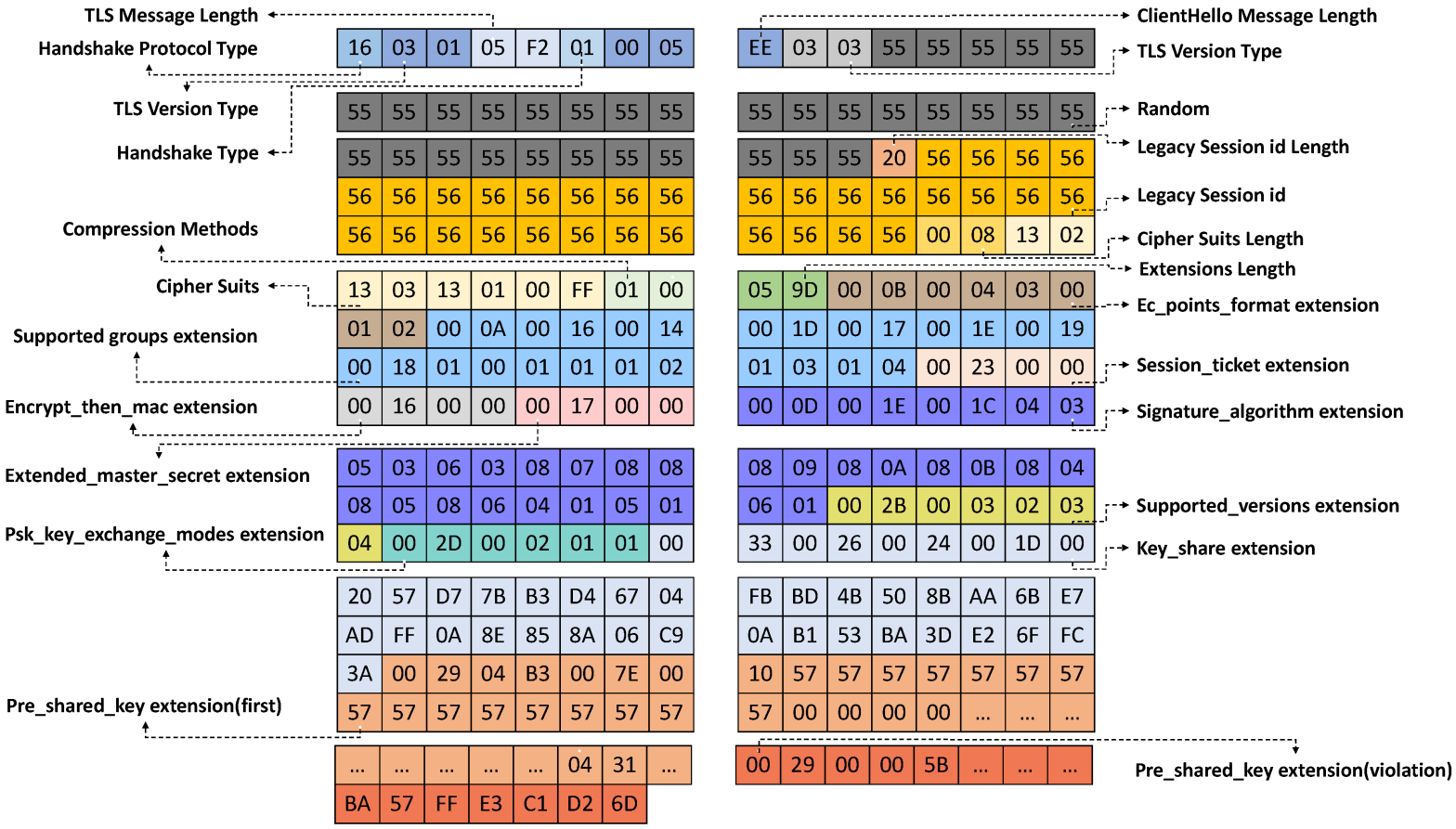
7. CVE 样例详解

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

恶意报文构造-Client-ClientHello 约束

- 检索方式：ClientHello + Client
- 手工看约束-client 发送 clienthello 消息满足的 must 约束-17 条
  - [1][clientANDserver][Sequence][Protocol messages MUST be sent in the order defined in Section 4.4.1 and shown in the diagrams in Section 2.]{**msg syntax**}
  - [2][client][Sequence][When a client first connects to a server, it is REQUIRED to send the ClientHello as its first TLS message.]{**state syntax**}
  - [3][client][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.]{**msg syntax**}
  - [4][client][legacy_session_id][In compatibility mode (see Appendix D.4), this field MUST be non-empty, so a client not offering a pre-TLS 1.3 session MUST generate a new 32-byte value. Otherwise, it MUST be set as a zero-length vector (i.e., a zero-valued single byte length field).]{**msg syntax**}
  - [5][client][legacy_compression_methods][For every TLS 1.3 ClientHello, this vector MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method in prior versions of TLS.]{**msg syntax**}
  - [6][client][Supported Versions][Implementations of this specification MUST send this extension in the ClientHello containing all versions of TLS which they are prepared to negotiate (for this specification, that means minimally 0x0304, but if previous versions of TLS are allowed to be negotiated, they MUST be present as well).]{**msg syntax**}
  - [7][client][Cookie][When sending the new ClientHello, the client MUST copy the contents of the extension received in the HelloRetryRequest into a "cookie" extension in the new ClientHello. Clients MUST NOT use cookies in their initial ClientHello in subsequent connections.]{**msg syntax && state syntax**}
  - [8][client][Signature Algorithms][Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension.]{**msg syntax**}
  - [9][client][Signature Algorithms][Clients offering these values <SHA-1 which is used in this context with either (1) RSA using RSASSA-PKCS1-v1_5 or (2) ECDSA.> MUST list them as the lowest priority (listed after all other algorithms in SignatureSchemeList).]{**msg syntax**}
  - [10][client][Key Share][Each KeyShareEntry value MUST correspond to a group offered in the "supported_groups" extension and MUST appear in the same order.]{**msg syntax**}
  - [11][client][Key Share][The key_exchange values for each KeyShareEntry MUST be generated independently.]{**msg syntax**}
  - [12][client][Key Share][Clients MUST NOT offer multiple KeyShareEntry values for the same group.]{**msg syntax**}
  - [13][client][Key Share][Clients MUST NOT offer any KeyShareEntry values for groups not listed in the client's "supported_groups" extension.]{**msg syntax**}
  - [14][client][Pre-Shared Key Exchange Modes][In order to use PSKs, clients MUST also send a "psk_key_exchange_modes" extension.]{**msg syntax**}
  - [15][client][Pre-Shared Key Exchange Modes][A client MUST provide a "psk_key_exchange_modes" extension if it offers a "pre_shared_key" extension.]{**msg syntax**}
  - [16][clientANDserver][Pre-Shared Key Exchange Modes][psk_dhe_ke: PSK with (EC)DHE key establishment. In this mode, the client and server MUST supply "key_share" values as described in Section 4.2.8.]{**msg syntax**}
  - [17][client][Pre-Shared Key Extension][The "pre_shared_key" extension MUST be the last extension in the ClientHello (this facilitates implementation as described below).]{**msg syntax**}{**VIOLATION**}

恶意报文构造-生成式

- 构建 windows 的 IIS 服务器环境
  - 拷贝基于 wireshark 捕获的 TLS 部分字节流，可以实现通信
  - 基于 LLM 生成 TLS 字节流，因为格式问题被拒绝
    - 没有样例（捕获的正常字节流）
    - 没有生成消息的约束，补充消息约束（client 端发送各个字段的约束）
    - 构建最小的 ClientHello 报文

```
// TLS Record Layer
16 03 03 00 72 // Content Type: Handshake (22), Version: TLS 1.2 (0x0303), Length: 114 (修改)

// Handshake Layer
01 00 00 6E // Handshake Type: Client Hello (1), Length: 110 (修改)

// ClientHello
03 03 // Legacy Version: TLS 1.2 (0x0303)
b4 9d 4c 5a 90 9a 4d 84 c8 f1 1a c5 4c 71 d4 70 // Random (32 bytes)
e2 bf 67 a6 b8 d0 de ca b0 e9 f2 27 e7 e7 3d 04
00 // Session ID Length: 0
00 02 // Cipher Suites Length: 2
13 01 // Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
01 00 // Compression Methods Length: 1, Compression Method: null (0)

// Extensions
00 43 // Extensions Length: 67 (修改)

// supported_versions Extension
00 2B 00 03 02 03 04 // Type: supported_versions (43), Length: 3, Version: TLS 1.3 (0x0304)

// supported_groups Extension
00 0A 00 04 00 02 00 1D // Type: supported_groups (10), Length: 4, Groups: x25519 (0x001D)

// signature_algorithms Extension
00 0D 00 06 00 04 04 03 08 04 // Type: signature_algorithms (13), Length: 6, Algorithms: ecdsa_secp256r1_sha256, rsa_pss_rsae_sha256

// key_share Extension
00 33 00 26 // Type: key_share (51), Length: 38
00 24 // Client Key Share Length: 36
00 1D // Group: x25519 (29)
00 20 // Key Exchange Length: 32
88 47 b5 30 a1 bc cb cc 7a 76 dc d0 9f 8f 9c e8 // Public Key (32 bytes)
c7 4a ea 7d 9c ff f8 dd 79 34 a1 be c2 64 6a 65
```