

编号: CACR20239SHEUZ



作品类别: ☒ 软件设计 ☐ 硬件制作 ☐ 工程实践

2023 年第八届全国密码技术竞赛作品设计报告

作品题目: 基于国密算法的轻量化车联网通信算法软件设计实现

2023 年 10 月 27 日

中国密码学会

基本信息表
编号：CACR20239SHEUZ
作品题目：基于国密算法的轻量化车联网通信算法软件设计实现
作品类别： <input checked="" type="checkbox"/> 软件设计 <input type="checkbox"/> 硬件制作 <input type="checkbox"/> 工程实践
<p>作品内容摘要：</p> <p>随着车联网产业的不断发展和飞速进步，现代车联网正在向智能化，多元化，高精度化的方向稳步发展。同时，车联网中的数据通信对高安全性，高可靠性，低时延等要求也越发急切。传统的加密方法和通信已经和很难满足越来越高的车联网通信要求了，于是，我们基于国家密码管理局认定和公布的密码算法标准及其应用规范以及轻量化物联网协议 MQTT 为基础设计出该款低时延，高安全性的轻量级加密通信系统。本文主要从设计方案和思路，系统测试以及应用前景三个方面讲解。</p> <p>设计思路和方案上，我们首先基于 GMSSL 实现的国密算法以及 MQTT 协议进行通信/加密方法的设计和改造，得到一套完善可靠的加密方案，并且同时实现了端对端加密和公开加密算法。同时，在 MQTT 协议的设计上，我们设计了一个特殊的数据包格式，保证了数据的传输；同时设计了多个 Topic 进行功能的分类以及特殊功能的实现。系统测试上，我们对系统的核心进行了多轮的功能测试和性能测试，同时对整个系统分别进行了负载测试，白盒测试等，检验了数据可导出，节点可追责，访问控制等多个功能的正确性和健壮性，同时也验证了系统的低延迟性可达到毫秒级通信。</p> <p>应用前景方面，我们分析了本系统在车联网中各方面的优势以及可应用方向，分别从复杂环境下的通信实时性，对设备的配置要求，车辆交通调度等方面分析了在实际生活中具有优势的场景区，具体又实际地给出了几个可能应用到本系统的车联网领域。</p> <p>作品特色：</p> <ol style="list-style-type: none"> 1) 特色：创新的加密工作流 2) 特色：独特的 Topic 设计 3) 特色：车辆数据实时分析报警 4) 特色：节点追责可视化 <p>关键词：</p> <p>车联网 国密 MQTT 协议 通信安全 身份认证</p>

目录

1	第一章 - 作品概述	5
1.1	引言	5
1.2	研究背景与意义	5
1.3	国内外研究现状	7
2	第二章 - 设计实现与方案	8
2.1	设计思路	8
2.1.1	国密算法介绍	8
2.1.2	模块简介	10
2.2	设计方案	11
2.2.1	加密模块设计	11
2.2.2	MQTT 模块设计	12
2.2.3	数据库模块	15
2.2.4	路径模块	17
3	第三章 - 系统测试与结果	19
3.1	测试方案	19
3.2	功能测试	20
3.2.1	访问控制测试	20
3.2.2	加解密模块功能测试	22
3.2.3	数据库模块测试	25
3.2.4	模拟通信测试	27

3.3	性能测试	32
3.3.1	解密模块测试	33
3.3.2	数据库模块测试	35
4	第四章 - 应用前景	38
5	第五章 - 结论	39

1 第一章 - 作品概述

1.1 引言

随着对监控道路状态、车辆状态、街道设施状态等需求持续增加，越来越多的智能汽车搭载了智能系统，包括多种多样的传感器、通信设备、监控装置等。此外，高性能的处理装置和存储装置也逐渐在现代车辆中普及。因此，现在的汽车不仅可以视为一辆代步工具，更可以被视为一个移动的网络节点，一个具有智能处理能力的计算和存储中心 [1]。由此，通过将现代智能汽车与先进的远程信息处理技术相结合，传统的车辆自组织网络（vehicular ad hoc network, VANET）逐渐演化成了车联网（Internet of vehicles, IoV）。

车联网产业是汽车、电子、信息通信、道路交通运输等行业深度融合的新型产业形态 [2]，能够实现车内、车际、车与路、车与人、车与服务平台的全方位网络连接，提升汽车智能化水平和自动驾驶能力，从而提高交通效率、改善驾乘感受 [3]。目前，车联网已经成为汽车产业不可逆转的发展方向，预计到 2025 年，中国市场 75.9% 的新车型将具备自动驾驶和联网功能。然而，车联网高速发展的同时，车联网中的安全隐患也愈发严重，各类车联网安全攻击事件频频发生，给用户隐私、财产、驾驶安全等多个方面带来了极大的威胁 [4, 5, 6]。因此，使用合适的方法对车联网通信进行加密势在必行。

对于轻量级的车联网系统，基于 MQTT 协议可以在提供高传输速度和稳定性的同时减少服务器成本，在加密过程使用国产密码算法，可以进一步减少使用常见密码破解方式攻击的可能，从而加强通信安全，防止信息泄露。

1.2 研究背景与意义

车联网是以车内网、车际网和车载移动互联网为基础，按照约定的通信协议和数据交互标准，在车与车（V2V）、车与路边设施（V2I）、车与行人（V2P）以及车与网络（V2N）之间进行无线通信和数据交换与共享的网络系统 [7]。车联网设备主要包括车联网终端和路侧设备。

从车联网终端角度，由于车联网终端集成了导航、移动办公、车辆控制、辅助驾驶等功能，导致车载终端更容易成为黑客攻击的目标，造成信息泄露，车辆失控等重大安全问题 [7]。因此车载终端面临着比传统终端更大的安全风险。车载终端存在的多个物理访问接口和无线连接访问接口使车载终端容易受到欺骗、入侵和控制的安全威胁，同时车载终端本身还存在访问控制风险、固件逆向风险、不安全升级风险、权限滥用风险、系统漏洞暴露风险、应用软件风险和数据篡改和泄露风险。从路侧设备的角度，由于路侧设备是车联网系统的核

心单元，它的安全关系到车辆、行人和道路交通的整体安全，主要面临非法接入、运行环境风险、设备漏洞、远程升级风险和部署维护风险 [8]。

从通信角度来说，轻量级车联网选择 MQTT 协议无疑可以为搭建系统提供便利。MQTT 是基于发布/订阅模式的物联网通信协议，具有简单易实现、支持 QoS、报文小等特点，在物联网协议中有举足轻重的地位。在车联网场景中，MQTT 依然能够胜任大量车辆和设备系统灵活、快速、安全接入的场景，同时还能保证复杂网络环境下消息实时性、可靠性，其主要应用优势如下 [9]：

1. 开放消息协议，简单易实现。市场上有大量成熟的软件库与硬件模组，可以有效降低车机接入难度和使用成本；
2. 提供灵活的发布订阅和主题设计，能够通过海量的 Topic 进行消息通信，应对各类车联网业务；
3. Payload 格式灵活，报文结构紧凑，可以灵活承载各类业务数据并有效减少车机网络流量；
4. 提供三个可选的 QoS 等级，能够适应车机设备不同的网络环境；
5. 提供在线状态感知与会话保持能力，方便管理车机在线状态并进行离线消息保留。

MQTT 目前有以下几个常见的安全隐患：认证和授权机制不充分、薄弱的密码、不安全的通信协议、拒绝服务（DoS）攻击 [9]。对于 MQTT 协议，安全尤为重要，因为通信过程发生信息泄露或被劫持，将会导致大量的终端失控，从而导致车辆信息、行程轨迹信息、用户隐私信息等关键信息流入不法分子手中，考虑到未来自动驾驶的普及，通信被劫持甚至有可能导致车辆驾驶的失控。从这个角度看，通信过程被攻击的威胁更甚于终端被攻击。

所以，车联网的安全是车联网从实验室走向实际运用必须保证的一环。而在我国，为了保障车联网安全，将车联网的通信过程与国密算法进行有机结合，无疑是一个很好的选择。国密，即国家密码局认定的国产密码算法；通过自主可控的国产密码算法保护重要数据的安全，是有效提升信息安全保障水平的重要举措。2022 年 2 月，工业和信息化部在现有国家车联网产业标准体系的基础上，组织编制了《车联网网络安全和数据安全标准体系建设指南》，其中已发布的 GB/T 37376-2019《交通运输数字证书格式》等国标文件中，凡涉及密码算法相关的内容，均考虑了国密的应用与实现。[10] 由此可见，基于国密算法的车联网通信不仅是为了保障车联网通信安全，更是受到国家重视、政策倾斜的技术方向。

1.3 国内外研究现状

为了防止数据在车联网内部或外部遭受攻击者非法窃听、篡改、伪造等威胁，车联网系统可采用密码技术对数据在传输、存储、使用的过程中进行加密保护，确保数据的机密性和完整性；建立完善的密钥、证书管理体系，保证密码资源的安全。同时，对车联网数据的访问进行权限控制，防止非授权用户访问。目前对于车联网安全的研究还比较少，但呈现逐年递增的趋势。图1展示了近 5 年网络安全顶级会议中车联网安全相关文献的发表情况，近年来车联网安全文献数量有不同程度的增长，可以看出车联网安全开始逐步受到企业和学术界的关注，对车联网安全展开系统化的研究势在必行 [11]。

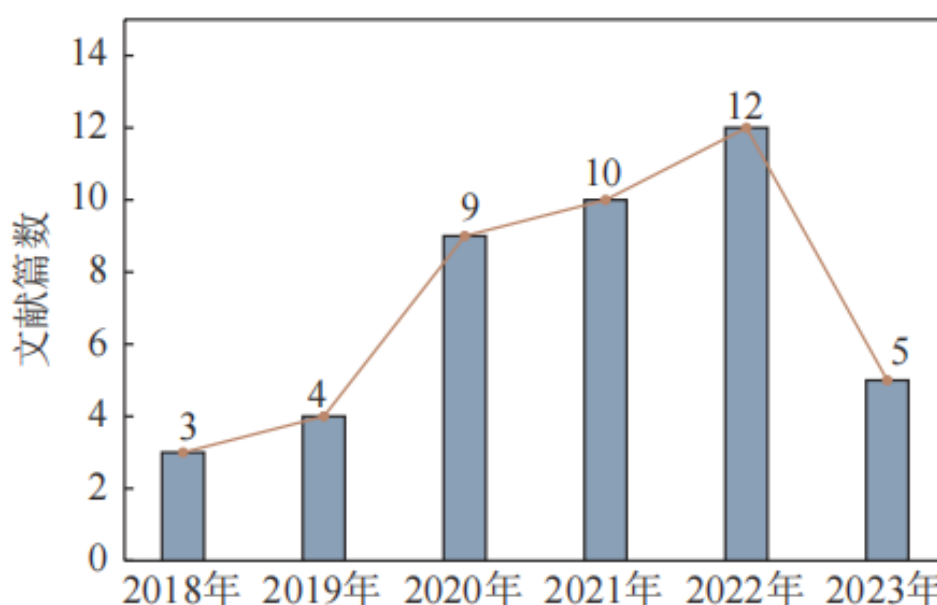


图 1: 近 5 年网络安全顶级会议车联网安全相关文献统计

目前车联网安全主要采用非对称密码算法，国外主要是 ECC NISTP256 和 brainpool，在我国主要使用 SM2、SM3 和 SM4 算法 [12]。国密算法应用在车联网体系中存在较多困难，由于车联网对时效性要求高，在保障时效性的情况下，密码算法强度便有所下降，这是国密算法应用在工业领域的一个最大的障碍，同时，对于车联网来说，通过 V2X 通信的不光是车辆之间，或者车与云端之间的通信，更大一部分是车端与路测设施之间的通信，因此路测设备与车体之间的密码算法接口需要统一，因此对每一家路侧设备制造商都有要求，目前对于此类标准还没有发布，对行业没有明确规范，因此推进难度较大 [13]。

2 第二章 - 设计实现与方案

2.1 设计思路

2.1.1 国密算法介绍

1. SM2 椭圆曲线算法

SM2 算法基于素域 F_p 和 F_{2^m} 上的椭圆曲线，以 ECC 椭圆曲线密码机制为基础进行设计，使用了较 ECDSA、ECDH 等国际标准更为安全的机制 [14]。SM2 的椭圆曲线方程为

$$y^2 \equiv x^3 + ax + b \pmod{p} \quad (1)$$

SM2 私钥生成方式如下：

- 生成一个椭圆曲线参数集 $\{p, a, b, G, n, h\}$ ， p 是椭圆曲线的有限域特征， a 和 b 数定义了椭圆曲线的方程， G 是椭圆曲线的基点， n 是基点 G 的阶， h 是曲线的余因子，通常应为 1。
- 选择一个随机数 d_A 作为私钥
- 计算公钥 $Q_A = d_A \cdot G$

SM2 算法加密过程如下 [15]：

- 选择一个随机数 k ，计算点 $C_1 = k \cdot G$
- 计算点 $S = M + C_1$ ， M 为要加密的信息
- 计算 $C_2 = S \oplus KDF(K, mLen)$
- 计算 $C_3 = Hash(C_1 || M || C_2)$

2. SM3 杂凑算法

SM3 密码杂凑算法适用于商用密码应用中的数字签名和验证，消息认证码的生成与验证以及随机数的生成，可满足多种密码应用的安全需求。SM3 在 SM2 和 SM9 中都有所使用 [16]。此算法接收输入长度小于 2 的 64 次方的比特消息，经过填充和迭代压缩，生成长度为 256 比特的杂凑值。

SM3 实现过程如下：

- 初始化 256 位寄存器 IV ，将其设置为初始值
- 将消息 M 按 512 位块进行分割为 M_1, M_2, \dots, M_n

- 消息扩展: $W_i = FF_j(W_{i-16} \oplus W_{i-9} \oplus (W_{i-3} \lll 15)) \oplus (W_{i-13} \lll 7) \oplus W_{i-6}$
- 压缩函数: $W'_i = W_i \oplus W_{i-3}$
- A, B, C, D, E, F, G, H 更新: $T_1 = (A \lll 12) + E + (FF_j(A, B, C) \oplus K_j) + W'_i$
- $T_2 = (A \lll 12) + E + (GG_j(E, F, G) \oplus K_j) + W'_i$
- $E' = D, D' = C \lll 9, C' = B \lll 19$
- $B' = FF_j(A, B, C) \oplus T_1$
- $A' = T_1, G' = F \lll 9, F' = E \lll 19$
- $E' = GG_j(E, F, G) \oplus T_2$
- $H' = H$
- 输出哈希值 $H = (A' \oplus B' \oplus C' \oplus D' \oplus E' \oplus F' \oplus G' \oplus H')$

3. SM4

SM4 算法是一种分组算法, 该算法的分组长度为 128 比特, 密钥长度为 128 比特。加密算法与密钥扩展算法都采用 32 轮非线性迭代结构。解密算法与加密算法的结构相同, 只是轮密钥的使用顺序相反, 解密轮密钥是加密轮密钥的逆序。每轮迭代的轮函数由一个非线性变换和线性变换复合而成, 非线性变换由 S 盒所给出 [17]。

SM4 加密流程如下:

- 初始化轮密钥 MK
- 将明文 M 分成 32 位块 X_0, X_1, \dots, X_n
- 令 i 从 1 到 n , 将 X_i 与轮密钥 MK_i 进行轮函数 F
- 输出加密后的密文 $C = X_n$

SM4 密钥扩展算法如下: 加密过程使用的轮密钥由加密密钥生成, 其中加密密钥 $MK = (MK_0, MK_1, MK_2, MK_3) \in (Z^{32})^4$, 加密过程的轮密钥生成方式如下:

$$(K_0, K_1, K_2, K_3) = (MK_0 \oplus FK_0, MK_1 \oplus FK_1, MK_2 \oplus FK_2, MK_3 \oplus FK_3) \quad (2)$$

$$rk_i = Ki + 4 = K_i \oplus T'(K_{i+1} \oplus K_{i+2} \oplus K_{i+3} \oplus CK_i), i = 0, 1, \dots, 31 \quad (3)$$

部分参数说明如下:

- T' 是将合成置换 T 的线性变换 L 替换为 L' , 见式 (3)

$$L'(B) = B \oplus (B \lll 13) \oplus (B \lll 23)$$

- 系统参数 FK 的取值为:

$$FK_0 = (A3B1BAC6), FK_1 = (56AA3350), FK_2 = (677D9197), FK_3 = (B27022DC)$$

- 固定参数 CK 取值方法为: 设 $ck_{i,j}$ 为 CK_i 的第 j 字节 ($i = 0, 1, \dots, 31; j = 0, 1, 2, 3$), 即 $CK_i = (ck_{i,0}, ck_{i,1}, ck_{i,2}, ck_{i,3}) \in (Z^{82})^4$, 则 $cki,j = (4i + j) \times 7 \pmod{256}$

2.1.2 模块简介

在车联网的车辆拓扑图中, 每个车可看作一个通信节点, 而由于车辆的高度可移动性以及位置的高度不确定性, 使得车辆拓扑图时刻都在变化着, 并且在车辆数目庞大, 道路情况复杂的情况下, 车辆拓扑图可能会变得十分的复杂。同时, 车机芯片的性能瓶颈也是我们需要关注的一大重点, 通信压力不能过大, 否则极有使车机崩溃, 甚至造成严重的交通事故。在此情况下, 各车机与服务器 (云) 之间的通信就需要满足如下的几个条件。

1. 通信的实时性, 节点与节点或者节点与云服务器之间的通信需要实时通信, 这样才能保证车辆运行的安全性和可靠性。
2. 通信的低开销, 对于车机来说需要处理的任务很多, 除了通信之外还需要实时处理传感器的信息, 运行用户安装的应用等功能, 低开销的通信可以保证车机的运行压力以及提高信息处理能力。
3. 通信的可靠性, 对于节点之间的通信, 数据信息的可靠是非常重要的, 通信数据的安全性决定了车辆运行可靠性和驾驶者的安全性。
4. 通信的复杂性, 对于节点之间的通信, 可能一个节点的信息需要被其他很多节点获取, 一个节点也需要获取其他很多节点的信息, 这就要求通信协议要支持复杂的通信拓扑。

针对上述要求, 我们采用了 MQTT 协议作为本作品的设计基础。基于 MQTT 协议的发布/订阅模型, 我们将每个通信节点看作客户端 (client), 将云服务器看作代理端 (broker)。MQTT 协议数据包开销相对于一般的 HTTP/HTTPS 要小很多, 同时通过发布/订阅模型也很好的满足了通信的实时性和复杂性。同时针对通信的安全可靠性要求, 我们将国密算法嵌入进 MQTT 协议中, 使得每个通信过程都是用国密有关算法进行加密的, 保证了数据的安全性。

2.2 设计方案

本软件中我们将软件分为如下几个模块

- 加密模块：用于加密用户发送的明文，具体可分为 *Public* 和 *P2P* 两种加密方式
- MQTT 模块：进行 Topic 分类，区分 *Public* 和 *P2P* 两种通信方式。
- 数据库模块：用于在 Broker 端记录用户发送的消息并做持久化存储，用于后续的追责和审计等。
- 路径模块：用于记录每个节点的行驶路径，支持导出等操作。

下面对于这些模块进行逐一分析。

2.2.1 加密模块设计

对于数据的加密，我们采用了设计了一个特殊的工作流，对于每一个要发送的数据包，都需要经过该加密流处理之后获得密文才会发送出去，对于客户端来说，正常的加密流程应该如下所示：

1. 设消息发送者为 A ， A 有一个 $SM4$ 密钥，一个 $SM3$ 的 $HMAC$ 密钥，以及一对 $SM2$ 公私钥对。
2. 设明文为 X ， A 首先对 X 进行 $SM3-HMAC$ 的认证符计算得到认证符 $HMAC(X)$ ，然后利用 A 的 $SM2$ 私钥对认证符进行消息签名，得到签名结果 $sig_{SM2-rk_A}(SM3(X))$ ，简称为 sig 。
3. 将 sig 与 X 进行拼接，得到新的字节流，然后使用 A 的 $SM4$ 密钥对该字节流进行 $SM4$ 加密，得到加密后的结果 $SM4_{sk}(sig + X)$ ，称为 Y_2 。
4. 接下来需要对消息类型做判断，分为两种情况。一种是 *Public* 类型的消息，也就是说对于车联网中的所有被 A 信任的通信节点都可以接收到这个消息；另一种是 *P2P* 类型的消息，是指发送者 A 指定接受者（称为 B ），该消息只有 B 可以解密得到明文，其他的通信节点即使接收到信息也无法解密出明文。

- (a) 对于 *Public* 类型的消息，我们将 A 的 $SM4$ 密钥 sk 进行封装，利用 A 的 $SM2$ 公钥对其进行加密，得到加密后的密钥 $SM2_{pk_A}(sk)$ ，称为 Y_1 。

(b) 对于 *P2P* 类型的消息，我们将将 *A* 的 *SM4* 密钥 *sk* 进行封装，利用 *B* 的 *SM2* 公钥对其进行加密，得到加密后的密钥 $SM2_{pk_B}(sk)$ ，称为 Y_1 。

5. 将 Y_1 和 Y_2 拼接，即为最终需要发送的密文 Y 。

根据上述步骤，我们可以得到密文的公式如下所示：

$$Y(X) = SM2_{pk}(sk) + SM4_{sk}(sig_{SM2-rk_A}(SM3(X)) + X)$$

工作流的两种模式示意图如图2和图3所示。

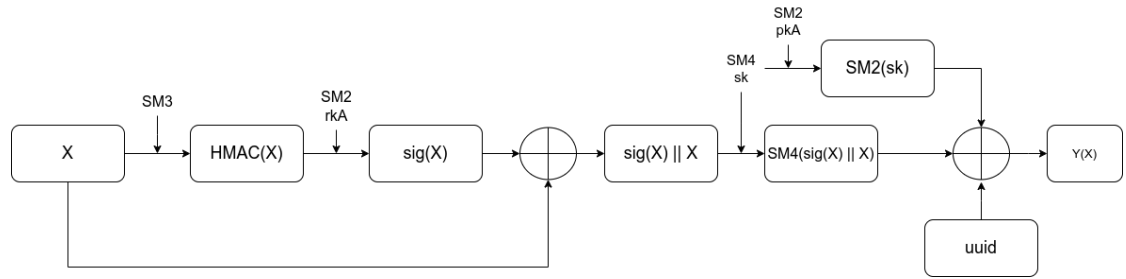


图 2: Public 模式加密流程

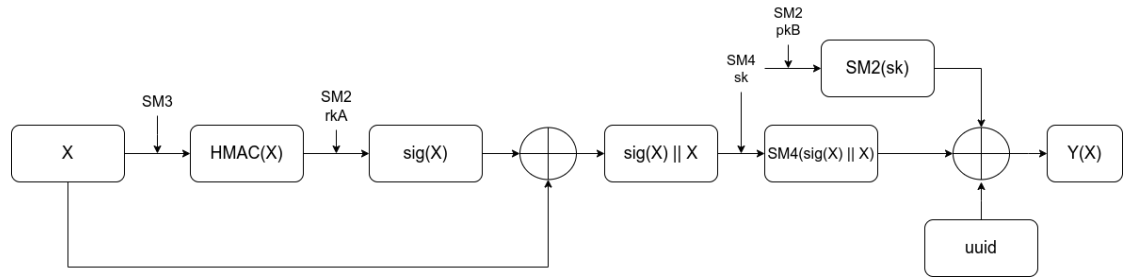


图 3: P2P 模式加密流程

2.2.2 MQTT 模块设计

该模块主要包含两个子模块，数据包模块和 Topic 模块。下面是对这两个模块的分析。

a. MQTT 的数据包格式 基于身份认证的需要以及信息安全的需要，我们将数据包载荷分为两个部分，如下所示

- 用户的唯一标识符 $uuid$ ，该标识符由节点第一次加入车联网，代理端自动为其生成的标识符作为该节点的唯一标识。
- 用户需要发送的密文 Y ，也就是2.2.1中得到的公式。

将这两部分进行拼接，即得到了完整的数据包载荷的公式

$$payload = uuid_{sender} + Y(X)$$

b. MQTT 的 Topic 设计 对于 MQTT 协议的 topic, 我们主要设计了四个 Topic 用于转发消息，如下所示。

- 公共频道 $/public$ ，用于发布 Public 类型的信息。
- P2P 频道 $/p2p/\{uuid_{receiver}\}$ ，用于发布 P2P 类型的信息。
- 路径记录频道 $/checkpoint$ ，用于记录车辆的行驶路径。
- 报警追责频道 $/accident$ ，用于记录车辆事故以及节点追责。

对于**公共频道**，任意一个通信节点都可以向该 topic 下发布消息，同时任意通信节点也可以订阅该 topic 用于获取公共信息，以此来实现消息的公共性，也就是通信的一对多性。具体通信过程如下所示

1. 设发送者为 A ，当 A 需要发送消息时，首先利用 PGP 工作流中的 Public 加密模式对明文 X 加密，得到密文 $Y(X)$ ，然后发布到 $/public$ 话题下。
2. 对于任意接受者 B ， B 首先从数据包的有效载荷 $payload$ 中获取发布该消息的用户的 $uuid$ ，然后在本地存储的文件中查找该 $uuid$ 对应的密钥信息，包括 $SM2$ 公私钥， $SM3-HMAC$ 密钥，以及 $SM4$ 的密钥以及初始化向量 iv 。如果没有查找到则代表该接受者不被 A 信任，此时需要接受者与 A 进行密钥交换与协商，而该过程在此略去。
3. 接受者利用读取到的密钥信息对消息进行解密，解密过程如图4所示。最终得到明文 X ，再针对该明文做出一些相关操作。

对于**P2P 频道**，若某个通信节点想和另一个节点通信，需要将消息发布到接受者的 Topic 下，也就是 $/p2p/\{uuid_{receiver}\}$ 。而对于每个节点的 $uuid$ 都是唯一确定的，所以不会出现 Topic 重复的情况，一定程度上确定了消息的单射一致性。P2P 模式实现了消息的私密性，也就是通信的多对一特性，具体通信过程如下。

1. 设发送者为 A ，接受者为 B ，当 A 需要发送消息时，首先利用 PGP workflow 中的 P2P 加密模式对明文 X 加密，得到密文 $Y(X)$ ，然后发布到 $/p2p/\$ \{uuid_B\}$ 话题下。
2. 接受者 B 订阅自己的私人 Topic，也就是 $/p2p/\$ \{uuid_B\}$ 。当 A 发布消息时， B 会立刻接收到，然后根据 *payload* 里面的 *uuid* 在本地文件中查找相关的密钥消息。如果没有查找到则代表该接受者不被 A 信任，此时需要 B 与 A 进行密钥交换与协商，而该过程在此略去。
3. B 利用读取到的密钥信息对消息进行解密，解密过程如图5所示，最终得到明文 X ，再针对明文做出一些相关操作。

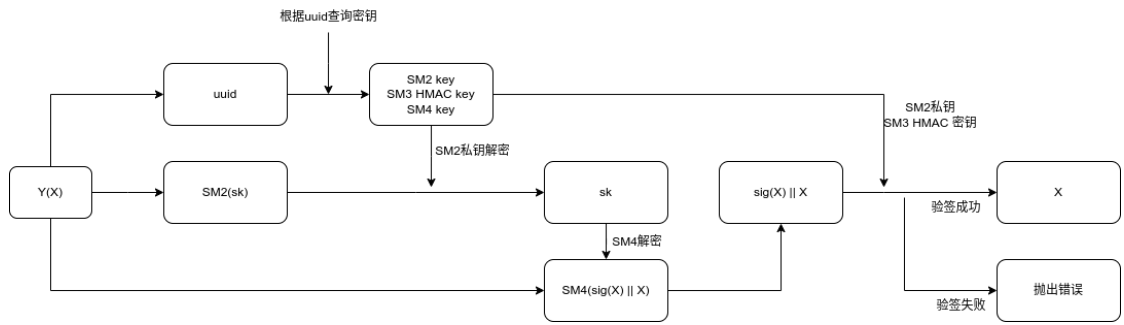


图 4: Public 模式 PGP 解密流程

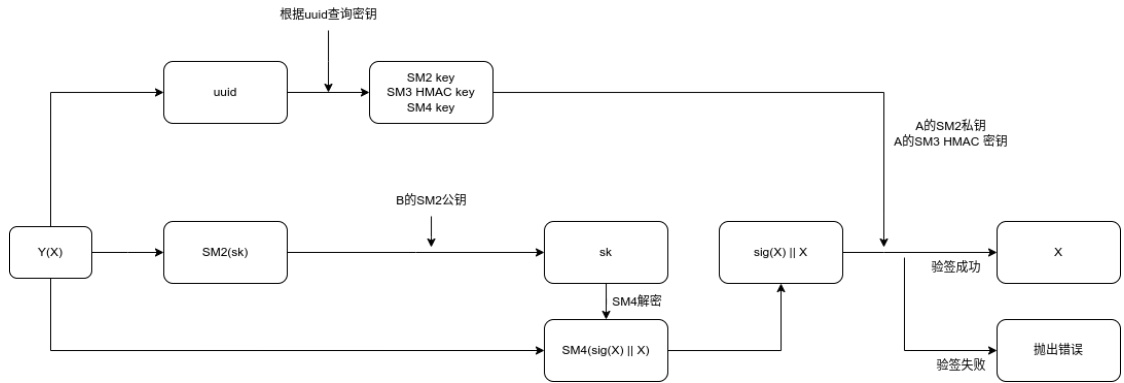


图 5: P2P 模式 PGP 解密流程

对于**路径记录频道**，车辆每到一个特殊的地点，则自动的向该 Topic 下发布一条消息，表示车辆已到达该地点，此时服务器会记录该节点并添加进用户的行驶路径中，并同步更新数据库中的相关数据。

对于**报警追责频道**，如果两个车辆发生相撞等事故，车机会自动检测传感器相关数据并判断是否发生事故，如果判断发生了事故则向该 Topic 下发布一条报警消息，内容为另一辆车的 *uuid*，此时服务器接收到该报警消息后，会进行发出报警信息等一系列动作。

2.2.3 数据库模块

为了实现软件中的节点追责，数据导出等功能，我们添加了数据库模块用于将数据作持久化处理。在将数据库部署在 Broker 端时，我们可以方便地存储和分析来自各种设备的数据。这也使我们能够实时监控和分析传感器、设备和客户端生成的数据，从而更好地理解 and 优化系统性能。同时在一些节点下线时，可以将有关数据进行存储从而避免重要数据丢失。

在本软件中，我们采用了 MongoDB 数据库，该数据库在车联网中有着广泛的应用，同时也具有着数据结构灵活，查询效率高，数据处理实时性等独特的优势。我们为本软件主要设计了四个 collection，分别用于存储不同的信息和数据结构。下面是对这些数据结构的解释与示例。

- *pems*: 该集合用于存储所有节点的密钥信息和用户信息，包括 *uuid* 和该用户的密钥信息，如表1所示。
- *graph*: 该集合用于存储用户的行驶路径，以有向图的形式存储，主要包括 *uuid* 和一个边数组，如表2所示。
- *public_message*: 该集合用于存储所有用户发送的公共消息，包括发送者的 *uuid*，时间戳 *timestamp* 以及消息主体 *message*，如表3所示。
- *p2p_message*: 该集合用于存储所有 p2p 类型的信息，包括发送者和接受者的 *uuid*，时间戳 *timestamp* 以及消息主体 *message*，如表4所示。

表 1: *pems collection*

字段名	类型	说明	值示例
uuid	String	用户唯一标识	2d4f674cdb934ad0bca257949960a45e
pass	Int32	加密 SM2 的密码	33236
sm2_key	String	SM2 算法的私钥	—BEGIN ENCRYPTED PRIVATE KEY— —END ENCRYPTED PRIVATE KEY—
sm4_key	String	SM4 算法的密钥	e69989cab34180545cd333a9cb06811d
sm4_iv	String	SM4 算法的初始 化向量 (CBC 模 式)	4e18af0da87ed9d2191c6b9297b88581
sm3_hmac_key	String	SM3 算 法 的 HMAC 密钥	0936681ca6c50303b0f08ef341bc5215

表 2: *graph collection*

字段名	类型	说明	值示例
owner	String	用户标识, 以 uuid 标识	2d4f674cdb934ad0bca257949960a45e
timestamp	ISODate	节点完成该路径的 时间戳	1999-12-31T23:59:59Z
edges	Array	路径边的集合	[{"source_vertex_name": "...", "tar- get_vertex_name": "..."}]

表 3: *public message*

字段名	类型	说明	值示例
uuid	String	用户标识, 以 uuid 标识	2d4f674cdb934ad0bca257949960a45e
timestamp	ISODate	消息发送时的时间 戳	1999-12-31T23:59:59Z
message	String	用户发送的明文消 息	test message

表 4: *p2p message*

字段名	类型	说明	值示例
sender	String	发送者的 uuid	2d4f674cdb934ad0bca257949960a45e
receiver	String	接收者的 uuid	a87fd0eec32a4a60a07b0103247b6a45
timestamp	ISODate	消息发送时的时间戳	1999-12-31T23:59:59Z
message	String	用户发送的明文消息	test message

2.2.4 路径模块

车辆行驶路径模块是软件的一个重要模块，专门设计用于存储、管理和分析车辆的行驶路径。它利用有向图数据结构，为车队管理和车辆监控提供了高度可视化和灵活的解决方案。本模块可记录重要事件，如违规行为、事故或停留时间过长，以及生成报告供管理层审查。这个模块的使用将有助于优化车队管理、提高车辆的效率 and 安全性，并降低运营成本。它为车辆行驶路径的可视化、实时监控和数据分析提供了强大的工具，为软件增加了强大的位置管理和车队调度功能。下面将介绍这个模块的具体实现。

a. 路径的数据结构 为了满足特定的车辆路径记录需求以及数据可导出等功能需求，我们在该模块中引入了有向图的数据结构用于表示车辆行驶的路径。我们对原本基础的有向图进行了封装和拓展，使得在基础的有向图基础上，添加了**路径可视化**，**路径查询**，**实时拓扑分析**等功能，数据结构示意图如图6所示。同时，为了实现车辆行驶路径的持久化存储，我们在 Broker 端不停的监听名 */checkpoint* 的 Topic，每当有节点在该 Topic 下发布消息，即更新该节点的行驶路径，并存储进数据库

b. 节点追责 所谓节点追责就是当发生事故时，可以找出发生事故的地点，节点以及时间，以此获得事故的相关信息，也达到了节点追责的效果。根据 MQTT Broker 的 Topic 设计，当发生事故时，相关节点会向 */accident* 发布一条消息，其中包含了另一个事故节点的信息，然后 Broker 会发出报警信息，告诉管理者发生了事故，并且会在本地查找并生成事故节点的最近一张行驶路径图，以此来帮助管理者更快更方便的找到事故的点和参与者。

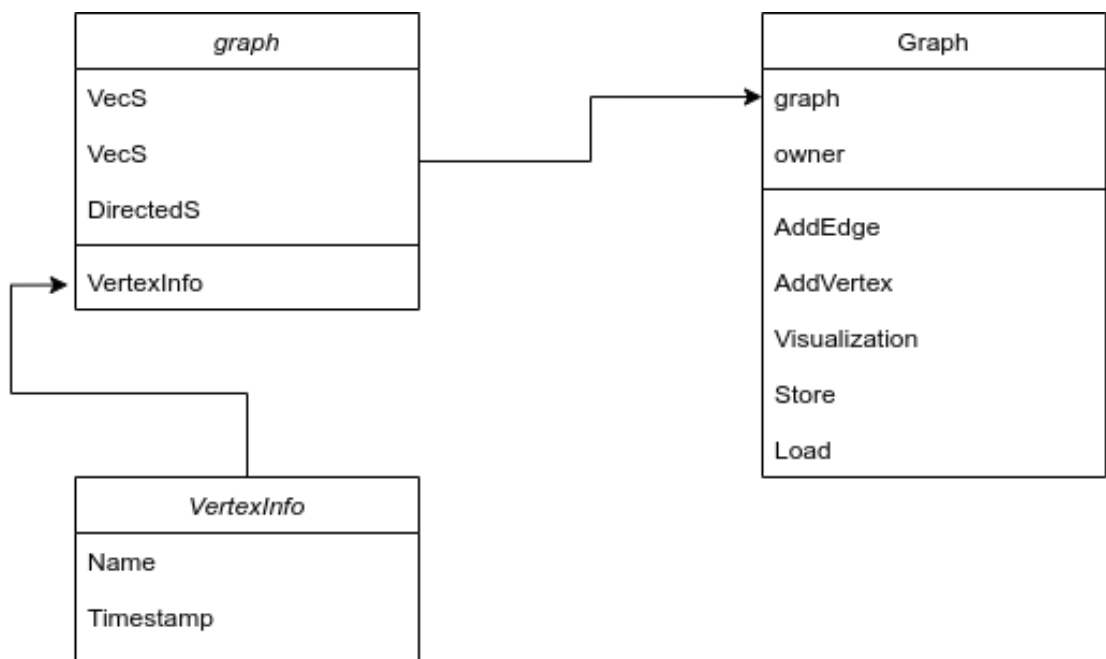


图 6: 有向图的封装

3 第三章 - 系统测试与结果

为了保证本软件运行时的可靠性和安全性，我们为本软件设计了一套测试方案，其中对于本软件的各个方面进行了测试，包括软件功能的正确性，核心函数的健壮性，功能函数的效率以及错误处理能力。

3.1 测试方案

由于缺少真实的运行环境，我们只能在本机上进行测试，测试结果等与真实的生产环境相比一定会有差别。测试环境如表5。

表 5: 测试环境

操作系统	Linux x86_64
CPU	16 AMD Ryzen 7 5800H with Radeon Graphics
Memory	16GB
软件及协议信息	MongoDB-7.0.2 MQTT 5.0

我们将各个模块的核心函数提取出来，分别进行了正确性测试 (FunctionalTest) 和效率测试 (PerformanceTest)，以此查看各个模块的功能和效率。同时对于整个软件的功能，我们采用自动化脚本的方式进行模拟客户端连接并随机发送消息，以此来达到模拟现实生活中车辆随机分布和随机发布消息的情况，同时测试软件的功能正确性和对错误的处理能力。测试方案如表6所示。

表 6: 测试方案

测试内容	测试方法	测试工具
加解密模块 FunctionalTest	黑盒测试，随机测试，Smoke Test	Google Test Framework
数据库模块 FunctionalTest	黑盒测试，随机测试	Google Test Framework
加解密模块 PerformanceTest	白盒测试，Smoke Test	Google Benchmark Framework
数据库模块 PerformanceTest	随机测试	Google Benchmark Framework
模拟通信 System Test	自动化测试，负载测试，白盒测试	Python3,paho-mqtt

需要说明的是，由于大部分测试是功能测试，故我们将核心函数独立出来封装为一个单

独的.hpp 文件，并放在测试文件夹下，测试文件夹内容如下所示

```
/test
├── Benchmark_Decrypt.cpp ..... 加解密模块性能测试文件
├── Benchmark_Database.cpp ..... 数据库模块性能测试文件
├── Test_Decrypt.cpp ..... 加解密模块功能测试文件
├── Test_Database.cpp ..... 数据库模块功能测试文件
├── Decrypt.hpp ..... 加解密模块核心函数
├── util.hpp ..... 工具类文件
├── accident.py ..... 模拟事故脚本
└── mqtt-emulator.py ..... 模拟高并发脚本
```

3.2 功能测试

根据测试方案，我们将对软件中的一些核心模块进行对应的功能测试，主要模块为加解密模块，数据库模块以及路径模块。下面是测试过程以及结果。

3.2.1 访问控制测试

我们将用户的数据存放在 MongoDB 数据库中，包括用户的 id、密码、订阅的主题和是否是超级权限用户等四条信息。其中，用户的 id 在数据库中称为“username”，后面存有一个 16 字符长度的随机字符串，该字符串不仅会存放在数据库，还会保存在客户端所在主机。用户密码在数据库中为“password”，后面存有经过了 PBKDF2 算法加密的密码。PBKDF2 算法是一种常见的密码加密算法，其原理简而言之就是利用 salted hash 进行指定轮数的重复计算，经过这样加密后可以有效地防御“彩虹表”攻击。订阅的主题和超级权限用户规定了访问权限。数据库数据截图如图7所示。

通过我们编写的数据库写入程序，用户数据会被添加到数据库中，或数据库对应用户的内容会被修改。在调整配置文件后，本系统可以实现访问控制。输入正确用户名和密码可以进行订阅发布，否则无法订阅发布。成功订阅发布截图如图8所示，失败订阅发布截图如图9所示。

由测试结果可以看出，本系统成功将用户信息存放在 MongoDB 数据库，并实现了登陆操作，还具有权限管理功能。

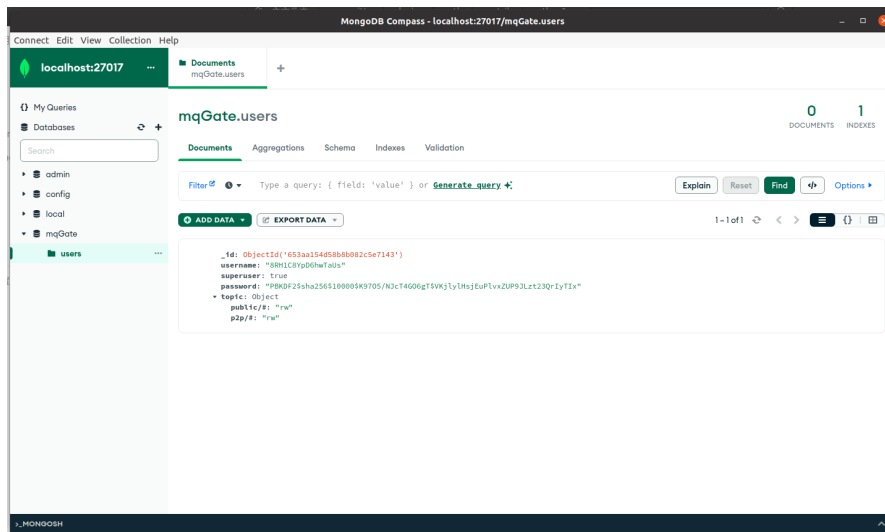


图 7: 数据库内数据结构示意图

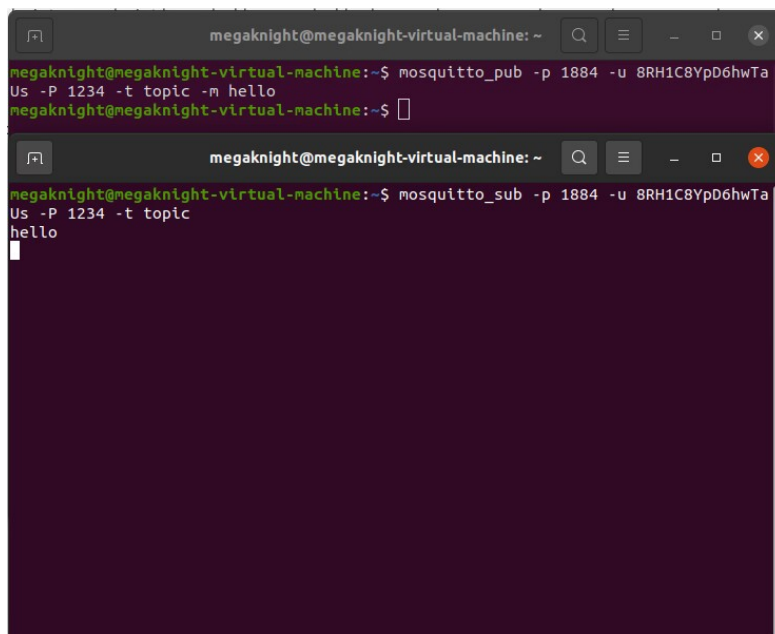
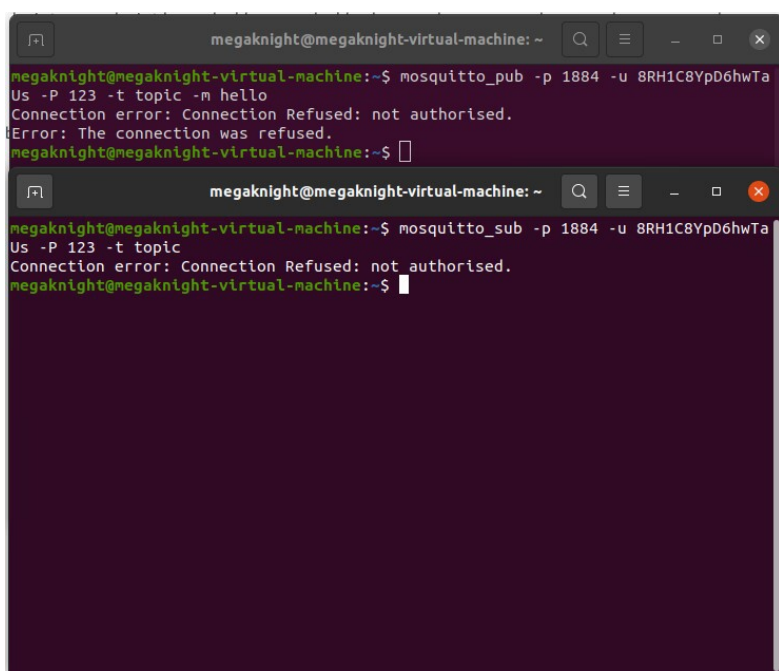


图 8: 订阅发布成功截图



The image shows two terminal windows from a virtual machine named 'megaknight'. The top window shows the command 'mosquitto_pub -p 1884 -u 8RH1C8YpD6hwTa' being executed, which results in a 'Connection error: Connection Refused: not authorised.' and 'Error: The connection was refused.' The bottom window shows the command 'mosquitto_sub -p 1884 -u 8RH1C8YpD6hwTa' being executed, which also results in a 'Connection error: Connection Refused: not authorised.' Both windows show the user 'megaknight' at the prompt 'megaknight@megaknight-virtual-machine: ~'.

```
megaknight@megaknight-virtual-machine: ~  
megaknight@megaknight-virtual-machine:~$ mosquitto_pub -p 1884 -u 8RH1C8YpD6hwTa  
Us -P 123 -t topic -m hello  
Connection error: Connection Refused: not authorised.  
Error: The connection was refused.  
megaknight@megaknight-virtual-machine:~$  
  
megaknight@megaknight-virtual-machine: ~  
megaknight@megaknight-virtual-machine:~$ mosquitto_sub -p 1884 -u 8RH1C8YpD6hwTa  
Us -P 123 -t topic  
Connection error: Connection Refused: not authorised.  
megaknight@megaknight-virtual-machine:~$
```

图 9: 订阅发布失败截图

3.2.2 加解密模块功能测试

对于加解密模块，主要测试的功能就是对于明文的加密以及对密文的解密，我们将这两个过程整合，成为一个加解密黑盒，我们只需要比较测试函数的输入输出即可，黑盒的输入是一个随机生成的字符串 *input*，长度在 1-1000 之间，黑盒的输出是一个字符串 *output*，如果 $input == output$ ，则该模块功能正常。示意图如图10所示。

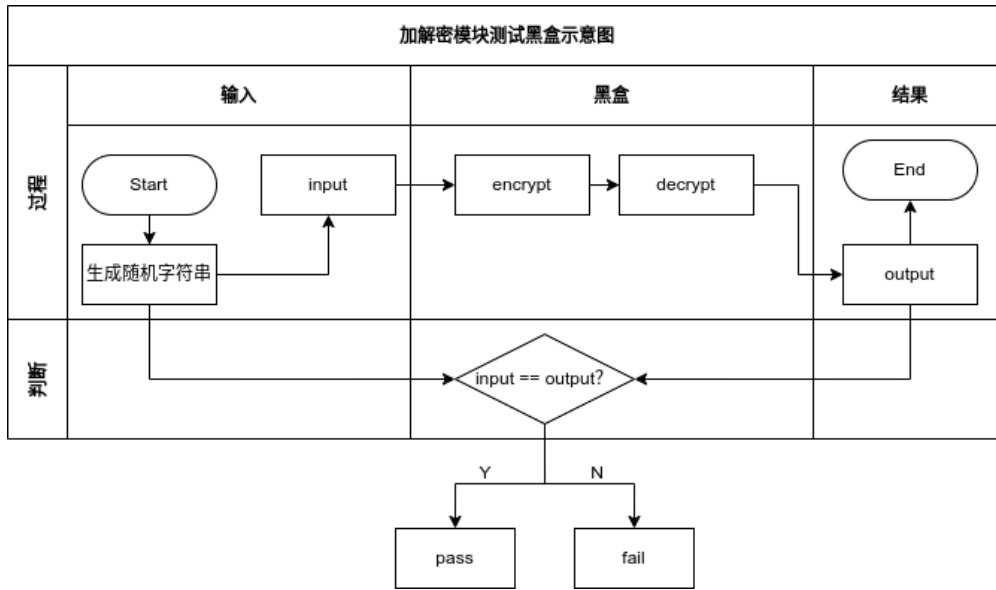


图 10: 加解密模块黑盒示意图

在该测试中，我们设计了不同长度的输入字符串，每个长度都循环测试 10 轮，并且同时测试 Public 和 P2P 两个模式，分别统计通过的案例和失败案例并计算通过率，最终获得结果。具体代码实现如下。

```

/*
 * @file Test_Decrypt.cpp
 * @brief 测试解密函数
 * @compile g++ -Wall -o Test_Decrypt Test_Decrypt.cpp
 *          -I/usr/local/include/mongocxx/v_noabi -I/usr/local/include/bsoncxx/v_noabi
 *          -lmongocxx -lbsoncxx -lgmssl -lgtest
 */

#include "Decrypt.hpp"
#include "util.hpp"
#include <gtest/gtest.h>

TEST(P2PDecryptTestCase, P2PDecryptTest){
    /*
     * 生成随机字符串，长度从1到1000，每个长度生成10个随机字符串
     * 然后检测解密后的字符串是否与原字符串相等
     */
    for (int i = 1; i < 1000; ++i) {

```

```

        for (int j = 0; j < 10; ++j) {
            auto msg = generateRandomString(i);
            auto receiver = get_uuid();
            EXPECT_EQ(msg,p2p_handler(generate_p2p(msg.c_str(),get_uuid().c_str(),receiver.c_str()),
                                     receiver));
        }
    }
}

TEST(PublicDecryptTestCase,PublicDecryptTest) {
    /*
    * 生成随机字符串，长度从1到1000，每个长度生成10个随机字符串
    * 然后检测解密后的字符串是否与原字符串相等
    */
    for (int i = 1; i < 1000; ++i) {
        for (int j = 0; j < 10; ++j) {
            auto msg = generateRandomString(i);
            EXPECT_EQ(msg,public_handler(generate_public(msg.c_str(),get_uuid().c_str())));
        }
    }
}

int main(int argc, char** argv) {
    mongocxx::instance ins{};
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

测试程序的运行截图如图11所示，测试结果如表7所示。

表 7: Test_Decrypt 测试结果

测试单元	测试案例数 (个)	通过数 (个)	失败数 (个)	通过率
PublicHandler	10000	10000	0	100%
P2PHandler	10000	10000	0	100%


```

→ ./TEST_Decrypt
[=====] Running 2 tests from 2 test suites.
[=====] Global test environment set-up.
[=====] 1 test from P2PDecryptTestCase
[ RUN      ] P2PDecryptTestCase.P2PDecryptTest
[          OK ] P2PDecryptTestCase.P2PDecryptTest (531801 ms)
[=====] 1 test from P2PDecryptTestCase (531801 ms total)

[=====] 1 test from PublicDecryptTestCase
[ RUN      ] PublicDecryptTestCase.PublicDecryptTest
[          OK ] PublicDecryptTestCase.PublicDecryptTest (274796 ms)
[=====] 1 test from PublicDecryptTestCase (274796 ms total)

[=====] Global test environment tear-down
[=====] 2 tests from 2 test suites ran. (806598 ms total)
[ PASSED   ] 2 tests.

```

图 11: Test_Decrypt 运行截图

由测试结果可以看出，所有的测试案例均**通过**。

3.2.3 数据库模块测试

对于数据库模块，主要测试的功能是 CRUD（增删查改）等操作，但是鉴于通信操作的不可恢复性，我们并未提供更新和删除操作，也就是说数据库模块只有新增和查找功能，我们分别对数据库进行了 10000 次增查操作，运行完毕后查看操作和数据库中的数据以检查操作是否成功。

在该测试中，我们随机生成了 10000 个不同的存储数据，然后分别对这 10000 条数据进行增查操作，分别统计通过的案例和失败案例并计算通过率，最终获得结果。具体代码实现如下。

```

/*
 * @file Test_Database.cpp
 * @brief 测试数据库函数
 * @compile g++ -Wall -o Test_Database Test_Database.cpp

```

```

-I/usr/local/include/mongocxx/v_noabi -I/usr/local/include/bsoncxx/v_noabi
-lmongocxx -lbsoncxx -lgmssl -lgtest
*/

#include <gtest/gtest.h>
#include "Decrypt.hpp"

std::vector<std::tuple<std::string, std::string, std::string>> testcases;

TEST(InsertTestCase, InsertTest){
    for (int i = 0; i < 10000; ++i) {
        auto sender = get_uuid();
        auto receiver = get_uuid();
        auto message = generateRandomString(i);
        testcases.emplace_back(sender, receiver, message);
        EXPECT_TRUE(util::insert_p2p_message(sender, receiver, message));
        EXPECT_TRUE(util::insert_public_message(sender, message));
    }
}

TEST(SelectTestCase, SelectTest){
    std::cout << std::get<2>(testcases[1]) << std::endl;
    std::cout << util::select_public_message(std::get<0>(testcases[1])) << std::endl;
    for (int i = 0; i < 10000; ++i) {
        EXPECT_EQ(std::get<2>(testcases[i]),
            util::select_public_message(std::get<0>(testcases[i])));
        EXPECT_EQ(std::get<2>(testcases[i]),
            util::select_p2p_message(std::get<0>(testcases[i]), std::get<1>(testcases[i])));
    }
}

int main(int argc, char** argv) {
    mongocxx::instance inst{};
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

测试程序的运行截图如图12所示，测试结果如表8。

由测试结果可以看出，所有测试案例均**通过**。

```
→ ./Test_Database
[=====] Running 2 tests from 2 test suites.
[=====] Global test environment set-up.
[=====] 1 test from InsertTestCase
[ RUN      ] InsertTestCase.InsertTest
[          OK ] InsertTestCase.InsertTest (101042 ms)
[=====] 1 test from InsertTestCase (101042 ms total)

[=====] 1 test from SelectTestCase
[ RUN      ] SelectTestCase.SelectTest
[          OK ] SelectTestCase.SelectTest (53505 ms)
[=====] 1 test from SelectTestCase (53505 ms total)

[=====] Global test environment tear-down
[=====] 2 tests from 2 test suites ran. (154548 ms total)
[ PASSED   ] 2 tests.
Passed test cases: 2
```

图 12: Test_Database 运行截图

表 8: Test_Database 测试结果

测试单元	测试案例数 (个)	通过数 (个)	失败数 (个)	通过率
P2P_insert	10000	10000	0	100%
Public_insert	10000	10000	0	100%
P2P_select	10000	10000	0	100%
Public_select	10000	10000	0	100%

3.2.4 模拟通信测试

对于车机的服务器，在现实生活中需要承受高额的流量压力和计算要求，然而我们在本地无法完全还原出现实生活中的复杂车辆情况，但是我们利用 Python 脚本和 mqtt 的 python 库对运行在本地服务器上模拟，从而达到模拟现实生活中的车辆情况。

在该测试中，首先编写 Python 模拟 MQTT 客户端连接的情况，然后随机选择模拟客户端进行消息的发布，作为期望的结果，理应有一个或者多个客户端可以接收到该消息并解密，同时 Broker 代理端也会将该条数据解密存入数据库，从而实现一个完整的通信过程，此外，我们还对节点的追责，数据的导出均做了模拟和测试，测试过程如下。

a. MQTT 客户端模拟 在 MQTT 客户端模拟这方面，我们采用 Python 脚本的形式实现自动化测试。其中模拟 MQTT 客户端连接的代码如下所示。同时，还需要模拟用户发送信息的过程，我们也采用 MQTT 的 Python 库实现，如此，我们便成功的模拟出了现实生活中的车辆连接情况，同时也可以对其进行节点追责，数据导出等操作。

```
import paho.mqtt.client as mqtt
import threading

# MQTT 代理 (broker) 信息
broker_address = "localhost"
port = 1883

# 定义回调函数来处理连接事件
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print(f"Connected to MQTT Broker with client ID: {client._client_id}")
        client.subscribe("test/topic")
    else:
        print(f"Connection failed for client ID {client._client_id} with code {rc}")

# 定义回调函数来处理接收消息
def on_message(client, userdata, message):
    print(f"Received message '{message.payload.decode()}' on topic '{message.topic}'
          for client ID {client._client_id}")

# 创建多个 MQTT 客户端
num_clients = 10000
clients = []

for i in range(num_clients):
    client = mqtt.Client(f"MyClient_{i}")
    client.on_connect = on_connect
    client.on_message = on_message
    client.connect(broker_address, port, keepalive=60)
    clients.append(client)

# 启动多个线程，每个线程处理一个 MQTT 客户端
def run_client(client):
```

```

        client.loop_start()
    while True:
        pass

threads = [threading.Thread(target=run_client, args=(client,)) for client in clients]

for thread in threads:
    thread.start()

# 等待所有线程完成
for thread in threads:
    thread.join()

```

b. 节点追责测试 模拟节点追责的过程如下，在上一步模拟的客户端中任选两个客户端模拟事故节点，分别记作 A 和 B ，具体步骤如下。

1. 首先让二者在指定路线上行驶一段时间，假设在某个 *checkpoint* 时，二者发生事故。
2. A 向服务器发送消息，说明 A 发生事故，消息中包含了 B 的信息。
3. Broker 端收到消息后，在后台调出 A 和 B 的行驶路径，同时发出报警给管理者和其他车机，警示当前事故道路。

进行节点追责模拟测试的脚本代码如下所示。运行截图如图13所示，数据库结果如图14所示。

```

import time
import threading
import paho.mqtt.client as mqtt

# MQTT 代理 (broker) 信息
broker_address = "localhost"
port = 1883

A = mqtt.Client("A")
B = mqtt.Client("B")
C = mqtt.Client("C")

def on_A_connect(client, userdata, flags, rc):

```

```

if rc == 0:
    print(f"Connected to MQTT Broker with client ID: {client._client_id}")
    client.subscribe("/accident")
    client.subscribe("/checkpoint")

    client.publish("/checkpoint", "paylaod")
    client.publish("/checkpoint", "payload")
    time.sleep(3)
    client.publish("/accident", "payload")
else:
    print(f"Connection failed for client ID {client._client_id} with code {rc}")

def on_B_connect(client, userdata, flags, rc):
    if rc == 0:
        print(f"Connected to MQTT Broker with client ID: {client._client_id}")
        client.publish("/checkpoint", "payload")
        client.publish("/checkpoint", "payload")
    else:
        print(f"Connection failed for client ID {client._client_id} with code {rc}")

def on_C_connect(client, userdata, flags, rc):
    if rc == 0:
        print(f"Connected to MQTT Broker with client ID: {client._client_id}")
        client.subscribe("/accident")
    else:
        print(f"Connection failed for client ID {client._client_id} with code {rc}")

def on_message(client, userdata, message):
    print(f"Received message '{message.payload.decode()}' on topic '{message.topic}'
          for client ID {client._client_id}")

A.on_connect = on_A_connect
B.on_connect = on_B_connect
C.on_connect = on_C_connect

A.connect(broker_address, port, keepalive=0)
B.connect(broker_address, port, keepalive=0)
C.connect(broker_address, port, keepalive=0)

def run_client(client):
    client.loop_start()

```

```

while True:
    pass

thread1 = threading.Thread(target=run_client, args=(A,))
thread2 = threading.Thread(target=run_client, args=(B,))
thread3 = threading.Thread(target=run_client, args=(C,))

# 启动线程
thread1.start()
thread2.start()
thread3.start()

# 等待所有线程完成
thread1.join()
thread2.join()
thread3.join()

```

运行完脚本后查看服务器端，可以发现多出了一张图片（如图15）并且发出了报警信息，在其他的模拟节点上也都接受到了警示信息，由此可看出节点追责功能正常。

```

B Home → Accident Point
插入新的文档成功。
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
1698337056: [2023-10-27 00:17:36:758]:success decrypt sm4 key and iv
/home/russ/GmSSL/src/sm2_lib.c:189:sm2_signature_from_der():
/home/russ/GmSSL/src/sm2_lib.c:286:sm2_verify():
1698337056: [2023-10-27 00:17:36:761]:sm2_verify success
1698337056: [2023-10-27 00:17:36:761]:after decrypt,get msg: A Home
1698337056: [2023-10-27 00:17:36:761]:a87fd0eec32a4a60a07b0103247b6a45 checkpoint A Home
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
1698337056: [2023-10-27 00:17:36:875]:success decrypt sm4 key and iv
/home/russ/GmSSL/src/sm2_lib.c:189:sm2_signature_from_der():
/home/russ/GmSSL/src/sm2_lib.c:286:sm2_verify():
1698337056: [2023-10-27 00:17:36:878]:sm2_verify success
1698337056: [2023-10-27 00:17:36:878]:after decrypt,get msg: Accident Point
数组字段更新成功。
1698337056: [2023-10-27 00:17:36:879]:a87fd0eec32a4a60a07b0103247b6a45 checkpoint Accident Point
A Home → Accident Point
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
/home/russ/GmSSL/src/sm2_lib.c:728:sm2_ciphertext_from_der():
/home/russ/GmSSL/src/sm2_lib.c:848:sm2_decrypt():
1698337056: [2023-10-27 00:17:36:995]:success decrypt sm4 key and iv
/home/russ/GmSSL/src/sm2_lib.c:189:sm2_signature_from_der():
/home/russ/GmSSL/src/sm2_lib.c:286:sm2_verify():
1698337056: [2023-10-27 00:17:36:998]:sm2_verify success
1698337056: [2023-10-27 00:17:36:998]:after decrypt,get msg: fc04472f91ff4df58567e7f205f268fa
1698337056: [2023-10-27 00:17:37:020]:a87fd0eec32a4a60a07b0103247b6a45 accident fc04472f91ff4df58567e7f205f268fa

```

图 13: 节点追责 Broker 端运行截图

<pre> _id: ObjectId('653a90706822229057039be1') owner: "fc04472f91ff4df58567e7f205f268fa" timestamp: 2023-10-26T16:14:40.000+00:00 edges: Array (1) 0: Object source_vertex_name: "B Home" source_vertex_time: 2023-10-26T16:14:40.000+00:00 target_vertex_name: "Accident Point" target_vertex_time: 2023-10-26T16:14:40.000+00:00 </pre>
<pre> _id: ObjectId('653a90726822229057039be2') owner: "a87fd0eec32a4a60a07b0103247b6a45" timestamp: 2023-10-26T16:14:42.000+00:00 edges: Array (1) 0: Object source_vertex_name: "A Home" source_vertex_time: 2023-10-26T16:14:43.000+00:00 target_vertex_name: "Accident Point" target_vertex_time: 2023-10-26T16:14:43.000+00:00 </pre>

图 14: 节点追责数据库结果截图

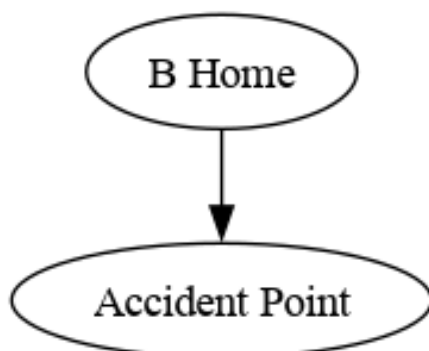


图 15: 系统生成的 B 的行驶节点图

3.3 性能测试

验证程序正确性的同时，软件对于解密和数据库插入查找等操作对效率有着较高的要求，故还需要对这两个模块进行性能测试 (PerformanceTest)，也就是 Benchmark。下面是测试过程以及结果。

3.3.1 解密模块测试

对于解密模块,需要进行性能测试的是两个解密函数: *p2p_handler* 和 *public_handler*。我们利用 GoogleBenchmark 框架对这两个函数进行了多轮的 Benchmark 测试,测试代码如下所示,测试程序的运行截图如图16和17所示,测试结果如表9所示。

```
/*
 * @file Benchmark_Decrypt.cpp
 * @brief 测试解密函数
 * @compile g++ -Wall -o Benchmark_Public_Decrypt Benchmark_Decrypt.cpp
 *          -I/usr/local/include/mongocxx/v_noabi -I/usr/local/include/bsoncxx/v_noabi
 *          -lmongocxx -lbsoncxx -lgmssl -lbenchmark
 * @compile g++ -Wall -o Benchmark_P2P_Decrypt Benchmark_Decrypt.cpp
 *          -I/usr/local/include/mongocxx/v_noabi -I/usr/local/include/bsoncxx/v_noabi
 *          -lmongocxx -lbsoncxx -lgmssl -lbenchmark
 */
#include <benchmark/benchmark.h>
#include <vector>
#include <string>
#include "Decrypt.hpp"

mongocxx::instance inst{};

std::vector<std::tuple<std::string, std::string, size_t, size_t>> p2p_cases = {
    ...
};

std::vector<std::tuple<std::string, std::string, size_t, size_t>> public_cases = {
    ...
};

void p2p_handler_benchmark(benchmark::State& state) {
    for (auto _ : state) {
        for (auto & p2p_case : p2p_cases) {
            benchmark::DoNotOptimize(p2p_handler(std::get<0>(p2p_case), std::get<1>(p2p_case)));
        }
    }
}

void public_handler_benchmark(benchmark::State& state) {
    for (auto _ : state) {
```

```


    for (auto & public_case : public_cases) {
        benchmark::DoNotOptimize(public_handler(std::get<0>(public_case)));
    }
}

BENCHMARK(p2p_handler_benchmark)->Range(1,1<<10);

BENCHMARK(public_handler_benchmark)->Range(1,1<<10);

BENCHMARK_MAIN();

```



```

→ ./Benchmark_P2P_Decrypt
2023-10-26T13:50:17+08:00
Running ./Benchmark_P2P_Decrypt
Run on (16 X 4391.88 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 512 KiB (x8)
  L3 Unified 16384 KiB (x1)
Load Average: 1.31, 0.81, 0.94

```

Benchmark	Time	CPU	Iterations
p2p_handler_benchmark/1	234264170 ns	230891789 ns	3
p2p_handler_benchmark/8	230536545 ns	228809632 ns	3
p2p_handler_benchmark/64	230435873 ns	228539240 ns	3
p2p_handler_benchmark/512	230493209 ns	228472580 ns	3
p2p_handler_benchmark/1024	232024491 ns	229362490 ns	3

图 16: P2P 解密模式 Benchmark 结果

```

→ ./Benchmark_Public_Decrypt
2023-10-26T13:53:54+08:00
Running ./Benchmark_Public_Decrypt
Run on (16 X 3951.24 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 512 KiB (x8)
  L3 Unified 16384 KiB (x1)
Load Average: 1.55, 1.10, 1.02

```

Benchmark	Time	CPU	Iterations
public_handler_benchmark/1	116678041 ns	115851530 ns	6
public_handler_benchmark/8	116601305 ns	115796402 ns	6
public_handler_benchmark/64	116669202 ns	115855200 ns	6
public_handler_benchmark/512	117398732 ns	116572335 ns	6
public_handler_benchmark/1024	116751335 ns	115982642 ns	6

图 17: Public 解密模式 Benchmark 结果

表 9: 解密模块 Benchmark 结果

		轮数					平均耗时 (ns)
		1	8	64	512	1024	
P2P	Time(ns)	228654826	229060583	229589261	229803419	228627331	229147084
	CPU Time(ns)	227004941	227250874	227800169	227937196	227009399	227400516
Public	Time(ns)	119906854	118866702	118600151	118238475	118145970	118751630
	CPU Time(ns)	118403798	117452280	117690270	117322563	117291372	117632057

根据表格以及输出结果可知, P2P 解密模式的 Benchmark 在229 147 084 ns 左右, Public 解密模式的 Benchmark 在118 751 630 ns 左右, 均达到了毫米级。

3.3.2 数据库模块测试

对于数据库主要有两种操作, 增加数据和查找数据, 主要是两个函数: *insert_message* 和 *select_message*。我们利用 GoogleBenchmark 框架对这两类函数进行多轮的 Benchmark 测试, 测试代码如下所示, 测试程序的运行截图如图18所示, 测试结果如表10所示。

```

/*
 * @file Benchmark_Database.cpp
 * @brief 测试数据库函数

```

```

* @compile g++ -Wall -o Benchmark_Database Benchmark_Database.cpp
    -I/usr/local/include/mongocxx/v_noabi -I/usr/local/include/bsoncxx/v_noabi
    -lmongocxx -lbsoncxx -lgmssl -lbenchmark
*/
#include "Decrypt.hpp"
#include <benchmark/benchmark.h>

mongocxx::instance ins{};

static void insert_benchmark(benchmark::State& state) {
    for (auto _ : state) {
        auto sender = get_uuid();
        auto receiver = get_uuid();
        auto message = generateRandomString(100);
        util::insert_p2p_message(sender, receiver, message);
        util::insert_public_message(sender, message);
    }
}

static void select_benchmark(benchmark::State& state) {
    for (auto _ : state) {
        auto sender = get_uuid();
        auto receiver = get_uuid();
        auto message = generateRandomString(100);
        util::select_p2p_message(sender, receiver);
        util::select_public_message(sender);
    }
}

BENCHMARK(insert_benchmark)->Range(1,1<<10);
BENCHMARK(select_benchmark)->Range(1,1<<10);

BENCHMARK_MAIN();

```

```

→ ./Benchmark_Database
2023-10-27T01:45:57+08:00
Running ./Benchmark_Database
Run on (16 X 3524.13 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 512 KiB (x8)
  L3 Unified 16384 KiB (x1)
Load Average: 0.95, 0.89, 0.80

```

Benchmark	Time	CPU	Iterations
insert_benchmark/1	9496452 ns	1704316 ns	418
insert_benchmark/8	9499119 ns	1688749 ns	410
insert_benchmark/64	9579579 ns	1695564 ns	417
insert_benchmark/512	9652189 ns	1741511 ns	399
insert_benchmark/1024	9527695 ns	1701232 ns	410
select_benchmark/1	17476002 ns	1844111 ns	380
select_benchmark/8	17465762 ns	1855855 ns	377
select_benchmark/64	17398093 ns	1852255 ns	361
select_benchmark/512	17471449 ns	1859451 ns	369
select_benchmark/1024	17489709 ns	1890986 ns	356

图 18: 插入查找 Benchmark 运行截图

表 10: 数据库模块 Benchmark 结果

		轮数					平均耗时 (ns)
		1	8	64	512	1024	
insert	Time(ns)	9496452	9499119	9579579	9652189	9527695	9551006.8
	CPU Time(ns)	1704316	1688749	1695564	1741511	1701232	1703161
select	Time(ns)	17476002	17465762	17398093	17471449	17489709	17460203
	CPU Time(ns)	1844111	1855855	1852255	1859451	1890986	1860531.6

由测试运行结果和表可知，插入单条数据的 Benchmark 约为9 551 006 ns，查询单条数据的 Benchmark 约为17 460 203 ns，均达到了**毫秒级**。

由上述测试的结果可知，本软件的稳定性，健壮性和效率都有一定的保证，这也给本软件应用到实际生产中提供了强有力的保障。

4 第四章 - 应用前景

国密算法目前作为我国官方认定的密码算法，在车联网安全领域进行实际应用却存在诸多困难。正如前文所言，车联网通信要求时效性，因此，这让密码算法在保证通信短时间内可以稳定传输的条件下，强度不得不随之降低。此外，标准的不统一、各厂商接口的不一致也让推广基于国密算法的车联网系统步履艰难。

但推进基于国密算法的车联网系统在今后已经势在必行，这不仅是因为随着车联网的发展和普及，标准体系正在逐渐走向成熟，更是因为攻击方式逐渐多元化，而现在的安全防护措施明显无法抵抗日益增强的攻击。运用自主研发的国密无论是对于安全保护意义还是战略意义都十分关键。

本系统作为基于国密算法的车联网通信系统，将数据隐私和数据安全视为首要任务，通过对 MQTT 协议通信过程使用国密算法加密，在实现百万级并发的同时，保证传输速度并提高系统安全性。本系统不仅具有 MQTT 协议订阅发布灵活、搭建简单、Payload 格式多样的特点，同时还对车机计算能力要求不高，可以适应复杂网络环境下的通信实时性，有助于大范围推广使用。此外，使用国密算法让对通信过程进行加密，可以对通信过程中车辆地理位置数据，如行驶轨迹、停车位置等，用户驾驶行为数据，如速度、急刹车和转弯习惯等进行保护，且相比传统加密安全性更高。

在车辆交通调度方面，基于车联网的车辆调度系统会依赖车联网协议对车辆地理位置信息和驾驶信息进行分析。本系统通过对通信过程中传输的车辆地理位置信息和驾驶信息进行保护，防止在通信过程中消息被劫持篡改导致的车辆调度系统出现系统级的错误。同时本系统的低延时特性，可以实现对车辆信息的实时获取与监控，对区域内交通情况进行分析与判断，例收集区域内车辆速度行驶速度及区域内车辆密集信息，从而判断拥堵情况进行合理的车辆调度，有效提升交通效率。

当然，仅仅利用密码算法解决车联网安全问题显然并不实际。与其他技术进行结合，同时更新车联网固件，减少固件漏洞，才能为车联网健康发展保驾护航。

5 第五章 - 结论

本文陈述了一种基于国密的轻量级车联网通信系统，该系统具备以下功能：

- 用户可以自由地加入和退出，在数据库中消除数据不会留下缓存；
- 用户的 id 自动生成，减少用户交互；
- 对于不同用户进行了 topic 限制，所有用户都能在 public 下发布，但在 p2p 主题下发布的内容只有具有对应 id 的用户才能收到；
- 数据通信采用了国密进行加密，保障通信安全；
- 实现了节点追责可视化功能，对于经过的节点画出流程图；
- 支持 TB 级数据存储，可以满足十万个读写并发操作，同时具有 mqtt 的传输短延迟的特征。

该系统在通信过程中创新地引入了 topic 访问控制，在保留传统 mosquitto 插件的权限管理的同时，引入了在通信过程中管理 topic 的发布和订阅，在保证隐私的同时，对于不正当信息进行丢弃也为系统的通信增加了稳定性。

此外，本系统对节点追责进行路径可视化处理。本系统生成的节点路径图展示了触发某个事件的所有节点和触发顺序，清晰地描绘了各个节点的功能，让查找、追责节点变得简单而一目了然，从而简化了用户的操作。

总的来说，本系统实现了对轻量化车联网通信的国密加密处理，在拥有较强的信息储存能力和较短的传输延迟的同时，具备权限可控制、出错可追责的功能。

参考文献

- [1] Wang Xiufeng, Cui Gang, Wang Chunmeng. Dynamic prediction of V2V link delay in urban Internet of vehicles[J]. Journal of Computer Research and Development, 2017, 54 (12) : 2721–2730 (in Chinese) (王秀峰, 崔刚, 王春萌. 城市车联网 V2V 链路时延动态预测 [J]. 计算机研究与发展, 2017, 54 (12): 2721–2730)
- [2] Zhang Ning, Yang Peng, Ren Ju, et al. Synergy of big data and 5G wireless networks: Opportunities, approaches, and challenges[J]. IEEE Wireless Communications, 2018, 25 (1) : 12–18
- [3] Tang Xiaolan, Xu Yao, Chen Wenlong. Bus data-driven city Internet of vehicles forwarding mechanism[J]. Journal of Computer Research and Development, 2020, 57 (4) : 723–735 (in Chinese) (唐晓岚, 琰尧, 陈文龙. 公交数据驱动的城市车联网转发机制 [J]. 计算机研究与发展, 2020, 57 (4) : 723–735)
- [4] Liu Yuanni, Li Yi, Chen Shanzhi. Blockchain-based Internet of vehicles security: A review[J]. SCIENTIA SINICA Informationis, 2023, 53 (5) : 841–877 (in Chinese) (刘媛妮, 李奕, 陈山枝. 基于区块链的车联网安全综述 [J]. 中国科学: 信息科学, 2023, 53 (5): 841–877)
- [5] Li Xinghua, Zhong Cheng, Chen Ying, et al. Overview of Internet of vehicles security[J]. Journal of Cyber Security, 2019, 4(3): 17–33 (in Chinese) (李兴华, 钟成, 陈颖, 等. 车联网安全综述 [J]. 信息安全学报, 2019, 4 (3): 17–33)
- [6] Kuang Boyu, Anmin Fu, Gao Yansong, et al. FeSA: Automatic federated swarm attestation on dynamic large-scale IoT devices[J]. IEEE Transactions on Dependable and Secure Computing, 2023, 20(4): 2954-2969
- [7] 车联网安全技术与标准发展态势前沿报告 (2019 年) . (中国通信学会, 2019, 12: 3)
- [8] 车联网安全技术与标准发展态势前沿报告 (2019 年) . (中国通信学会, 2019, 12: 4)
- [9] EMQ, 车联网场景中的 MQTT 协议, 2022
- [10] EMQ, 国密在车联网安全认证场景中的应用, 2022
- [11] Kuang Boyu, Li Yuze, Gu Fangming, Su Mang, and Fu Anmin. Review of Internet of Vehicle Security Research: Threats, Countermeasures, and Future Prospects. Journal of Computer Research and Development, 2023, 60 (10) : 2305 (in Chinese) (况

博裕, 李雨泽, 顾芳铭, 苏铨, 付安民. 车联网安全研究综述: 威胁、对策与未来展望. 计算机研究与发展, 2023, 60 (10): 2305)

[12] 车联网安全技术与标准发展态势前沿报告 (2019 年). (中国通信学会, 2019, 12: 29)

[13] 车联网中密码算法应用现状分析 (中国信息安全, 2019, 9)

[14] 国密算法概述 (SM1、SM2、SM3、SM4、SM7、SM9、ZUC)

[15] GM/T 0003-2012, SM2 椭圆曲线公钥密码算法 [S]

[16] SM3 密码杂凑算法, 国家密码管理局, 2010 年 12 月

[17] 信息安全技术, SM4 分组密码算法, 中华人民共和国国家标准 GB/T 32907-2016