ChatGPT

三种物联网协议安全架构的全面对比分析：IoTivity、Thread 和 AllJoyn

引言

物联网（IoT）设备互联常用多种协议框架，其中 **IoTivity**、**Thread** 和 **AllJoyn** 是具有代表性的三种。它们分别由不同组织主导（IoTivity 来源于开放互联基金会 OCF，AllJoyn 曾由 AllSeen Alliance 推动，Thread 则由 Thread Group 推出），各自定位于协议栈的不同层级。本文将从完整协议栈视角，对这三种协议的 **安全架构和机制设计**进行系统性比较，包括：身份认证、数据加密、密钥管理、访问控制等安全机制；分析各协议在网络层、传输层、应用层可能存在的安全缺陷；列举实际的漏洞案例（CVE）和已披露的安全事件；总结实际开发与部署中常见的错误配置或误用；并引用现有学术研究和安全分析论文的结果作为参考。最后，通过对比表格梳理三者在安全设计上的异同，并辅以示意图展示其安全架构或攻击面结构。希望本报告为安全分析人员和研究者提供清晰详尽的参考。

IoTivity 的安全架构与机制设计

IoTivity 是 OCF（开放互联基金会）的开源参考实现，工作在 IP 网络之上，采用 RESTful 架构模型，基础协议为 **CoAP**（Constrained Application Protocol）[1]。IoTivity 的安全框架遵循 OCF 安全规范，强调从设备接入（onboarding）到通信的全流程强安全性[2]。主要安全机制包括：

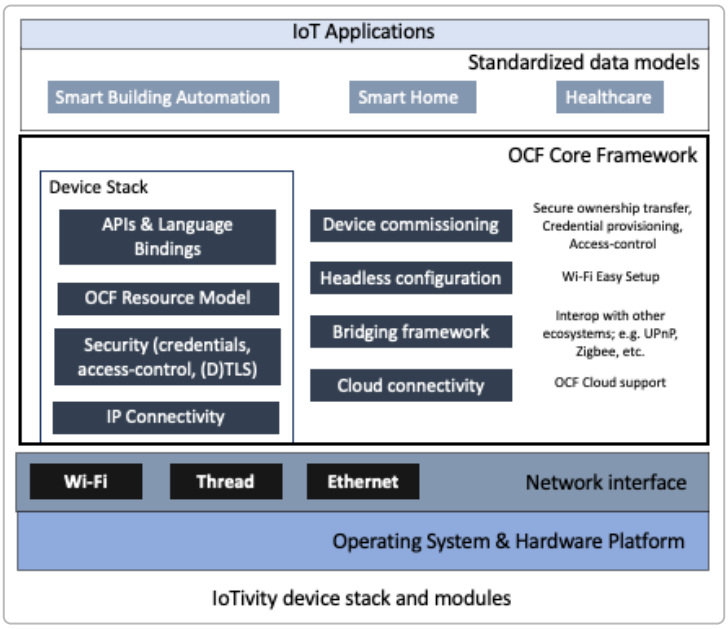
- **身份与认证**：每个 IoTivity 设备在接入网络时要经历所有权转移（Ownership Transfer）的过程。OCF规范定义了三种设备引入/认证模式：**"Just Works"**（即插即用，无需预先共享密钥，在首次握手时直接建立共享密钥，但缺乏验证，易受中间人攻击）[3]；**"Random PIN"**（使用用户提供的随机PIN码作为临时密码，利用密码验证的ECDH密钥交换ECSPEKE建立信任）；**"Public Key Infrastructure (PKI)"**（基于公钥证书，利用制造商提供的设备证书或自签名证书进行认证）[4][5]。在基于证书的模式下，设备会拥有由受信任CA签署的身份证书，用于后续认证。OCF保证所有设备在接入后都有唯一身份（通常以UUID表示），并通过认证确保设备身份名副其实[4]。

- **数据加密与传输安全**：IoTivity 在安全模式下使用 **DTLS (Datagram TLS)** 来保护设备间的消息通信[6]。具体而言，在设备成功完成所有权转移和认证后，客户端与服务器之间建立 DTLS 安全会话，对 CoAP 消息进行加密传输，防止窃听和篡改[7]。DTLS通常采用对称加密算法（如 AES-CCM）确保数据机密性和完整性，密钥则通过前述的密钥交换机制（ECDHE 等）协商生成[8]。对于组播通讯，OCF规范还定义了安全组播的方法（如基于预共享组密钥的保护）。需要注意的是，IoTivity 默认情况下并不会强制开启安全功能——必须在编译时启用特定标志（`SECURED=1`）才能使能上述安全机制，否则通信将以非加密方式进行[9]。这一点要求开发者特别注意，以免产品遗留"不设防"的后门。

- **密钥管理**：IoTivity 在设备引导过程中完成密钥的分发与建立。例如，使用"Just Works"时设备通过 ECDH 随机生成会话密钥但不验证身份；使用 PIN 模式时利用用户输入的 PIN 作为密码因子派生密钥；使用 PKI 模式时通过证书验证并使用设备的公私钥对完成 ECDHE_ECDSA 握手来生成共享密钥[10][11]。生成的共享会话密钥用于后续 DTLS 通信加密。此外，OCF还有设备凭证管理机制：设备可存储受信任的对等方公钥证书或预共享密钥，以支持后续直接建立安全会话。密钥和证书通常保存在设备的安全存储区域，由 IoTivity

的安全子系统（Security Resource Manager）管理。对于设备退网或更换所有者，OCF 提供吊销和重置机制，确保旧密钥失效并防止未经授权的后续访问。

- **访问控制**：IoTivity 实现了细粒度的访问控制模型。每台设备都维护一个**访问控制列表（ACL）**，在 OCF规范中表示为资源路径 `/acl2`，其中包含若干条"访问控制项（ACE)"对象 [12]。每条ACE指定了允许访问该设备上某资源的主体（可以是特定对等设备的UUID，或特定角色角色证书），以及允许的操作集合（读、写、执行等）和资源URI范围 [12]。当启用了安全模式（资源被标记为 `OC_SECURE` 属性）时，IoTivity 设备会在处理每个传入请求前检查ACL，只有符合权限策略的请求才被执行 [13]。这种权限模型使得不同设备间可以按照最小权限原则进行互动。例如，一个智能门锁设备可以将开锁/上锁资源仅授予家庭管理员用户的设备访问，而拒绝普通传感器设备的此类请求。



IoTivity device stack and modules

*IoTivity*设备栈和模块架构示意图。该架构由底层操作系统和硬件平台支撑，通过多种网络接口（*Wi-Fi*、*Thread*、以太网等）接入*IP*网络。设备栈中包含网络通信、安全（凭证、访问控制、*DTLS* 等） [13] [6]、资源模型等模块，上层是*OCF*核心框架提供的设备配置、桥接、云接入等功能，以及应用层的标准数据模型。

安全缺陷与漏洞分析（IoTivity）：作为基于IP的开放框架，IoTivity 的安全主要取决于正确实施 OCF 规范。然而，在不同层面上仍可能出现安全问题：

- 网络层：IoTivity 本身运行于IP之上，通常使用现有链路层（Wi-Fi、Thread 等）的安全特性。但如果部署在开放网络下，攻击者可针对 IP 层进行传统网络攻击（如IP扫描、UDP泛洪）。此外，IoTivity早期版本的CoAP服务器存在可被利用进行DDoS放大的漏洞：研究发现 IoTivity 1.3.1 及之前版本中，攻击者可以向其CoAP 接口发送伪造源地址的小请求包，诱使设备发送较大的响应到受害者IP，从而放大流量 [14]。该漏洞（CVE-2019-9750）允许**分布式拒绝服务**攻击，被动员的IoTivity设备充当放大器，将造成目标服务器过载 [14]。修复措施是在设备端引入请求率限制和源地址验证策略。

- 传输层：IoTivity 依赖 DTLS 保证传输安全，但DTLS本身可能遭受如握手泛洪、重放攻击等。如果设备在握手阶段不采用Cookie机制，攻击者可伪造大量初始握手请求耗尽设备资源。此外，如果使用不安全的密码套件（例如使用静态PSK且弱密码），可能被暴力破解密钥。OCF规范强制使用强加密（如 ECC 和 AES-

CCM），但实际实现中开发者误用也会引入漏洞。例如，**未开启安全模式**就是一大风险：如前所述，若开发者忘记在编译时启用 `SECURED=1` 或将资源标记为安全，通信将以明文进行 [9]，攻击者可直接嗅探篡改数据。**"Just Works"模式滥用**亦是常见错误——它方便调试但没有身份校验，若在产品中启用则可能被中间人攻击拦截会话 [3]。因此生产环境应避免使用无认证的握手方式。

- 应用层：IoTivity 提供了丰富的资源访问接口，应用层安全问题多与访问控制配置不当有关。若设备的ACL配置过于宽松（例如为简化开发而允许了 `*` 通配符访问所有资源），将导致权限滥用风险。攻击者一旦在网络内取得合法身份（例如攻陷另一台设备获取其证书/密钥），可能利用过度权限来操纵敏感设备功能。OCF 规范允许角色证书和分组权限管理，但开发者若未合理划分权限域，**特权滥用**就很难避免。另外，IoTivity设备还可能受普通应用层漏洞影响（如缓冲区溢出、整数溢出等编码漏洞）。虽然截至目前未公开报告很多此类漏洞，但安全审计不可掉以轻心。

- 已披露漏洞案例：除了前述的 CVE-2019-9750 DDoS 放大漏洞 [14] 外，IoTivity 项目本身相对较新，公开的安全漏洞有限。一些研究提及了 IoTivity 在默认情况下未启用安全、需要显式编译开启的设计缺陷 [9]——这在安全领域被认为是危险的"默认不安全"配置。未来随着 IoTivity 部署增加，可能会出现更多安全通报，开发者应紧跟OCF安全规范的更新并及时升级版本。

实际误用与错误配置（IoTivity）：在实践中，IoTivity 的误用主要集中在以下方面：（1）**忽略安全模式**：开发者为图方便，在开发调试时关闭了安全功能，却忘记在生产固件中开启，导致设备通信全程明文，无任何认证，加密形同虚设。[9] 指出只有启用特定编译选项和资源标志后安全才生效，因而这个步骤的疏忽相当常见。（2）**滥用默认凭证/测试证书**：OCF规范建议每台设备出厂烧录唯一证书或密钥，但一些厂商为省事可能在所有设备上使用相同的测试证书或默认PSK。这将导致一台设备密钥泄露，整个系列产品陷入被动。（3）**权限配置不当**：ACL若配置不严谨，可能让不该有权的设备获权。例如某些开发者为确保互操作，干脆允许网络内任何设备读写其资源。这完全破坏了访问控制的意义。（4）**随机数种子不足**：虽未见公开报告，但物联网常见问题是在资源受限设备上随机数源不足，导致密钥可预测。如果IoTivity设备没有可靠的随机数，密钥交换将容易被破解。总之，IoTivity 的安全依赖正确的实现和配置。OCF也提供了安全认证计划来避免上述错误，但设备制造商和开发者自身的安全意识仍是关键。

AllJoyn 的安全架构与机制设计

AllJoyn 是一个开源的物联网应用框架，提供**分布式消息总线**让不同设备和应用互联互通。AllJoyn 不同于IoTivity基于CoAP的REST模型，它采用类似于 D-Bus 的总线协议，支持在局域网内发现服务、远程过程调用等 [1]。AllJoyn侧重于**应用层**的互操作，具有去中心化架构（无单一集线器），在安全设计上相当成熟完备 [15] [16]。其主要安全机制包括：

- **身份与信任域**：AllJoyn 使用**公钥基础设施 (PKI)** 来建立设备身份和信任关系。每个接入 AllJoyn 网络的应用/设备（AllJoyn 将物理设备或软件应用统称为"应用 (app)"）都被视为独立的安全主体，必须拥有由**安全管理器 (Security Manager)**签署的**身份证书** [16]。安全管理器是AllJoyn架构中的特殊组件，由管理员（如家庭户主）操作，用于发行和管理设备证书 [16]。在 AllJoyn 的 **Security 2.0**模型中，引入了**"信任域 (Trust Domain)"**的概念，例如家庭环境可以被视作一个信任域，域内的设备由同一个安全管理器发行证书，从而彼此信任通信 [17]。初次部署时，管理员通过安全管理器将受信任的根证书植入每台设备，并给设备颁发身份证书。该证书包含设备的公钥以及一个权限清单摘要（manifest digest）[18]，证明该设备在网络中的身份和权限。所有 AllJoyn 设备开机加入网络时会相互认证证书链，只有共享同一信任域根证书的设备才能互相信任通信。

- **认证与密钥交换**：AllJoyn 建立会话时使用**椭圆曲线 Diffie-Hellman 临时密钥交换 (ECDHE)** 来生成对称会话密钥，并辅以多种认证方式来验证密钥交换的对方身份 [19]。在 AllJoyn 的安全机制1.0中，应用可以自行选择支持的认证方式，包括：`ECDHE_NULL`（不认证，仅做DH握手，存在主动中间人攻击风险） [3]；`ECDHE_PSK`（预共享秘钥认证，PSK需足够随机保证安全）；`ECDHE_ECDSA`（使用数字证书进行ECDH握手认证，即基于双方身份证书的签名验证）；以及 `SRP` / `ECSPEKE`（基于密码/短PIN码的方案，用于用户输入场景） [3]。自 AllJoyn **Security 2.0** 起，框架统一改进为主要使用 **ECDHE_ECDSA** 方式，即在完成初始设备信任配置后，后续会话一律使用证书认证的ECDH密钥交换，提升安全一致性 [10]。成功完成密钥交换后，双方使用 NIST P-256 椭圆曲线参数计算出共享密钥，随后通过 TLS PRF 算法派生会话密钥 [11]。AllJoyn 使用的对称加密为 **AES-CCM**（提供加密和消息认证） [8]，确保通信内容机密性和完整性。

- **数据加密与通信安全**：AllJoyn 的消息总线在架构上位于应用层，但其 **端到端数据**同样经过加密保护。建立起安全会话（Session）后，AllJoyn 所有在该会话上的信号、方法调用、返回值等消息，都使用前述协商的对称密钥加密 [8]。换句话说，与IoTivity通过DTLS保护CoAP包类似，AllJoyn通过自己的总线协议对应用数据逐消息进行机密性和完整性保护。由于AllJoyn支持多种底层传输（TCP、UDP、Bluetooth等），其安全机制独立于传输层，在应用框架内完成加密，因此即使总线消息走TCP，仍是不依赖TLS而由AllJoyn层加密。在设备之间通常存在一个AllJoyn **路由节点 (Router)** 负责转发总线消息，AllJoyn采用**分布式信任**——各节点自行验证消息签名和来源授权，并不需要中心化网关。AllJoyn Security 2.0 还支持组播信号的加密和认证，利用共享会话密钥实现组通信的安全。

- **访问控制**：AllJoyn 提供 **粒度极细**的访问控制机制，可以精确到接口成员级别（即具体某个方法或信号） [20]。不过，一般推荐在**接口级**进行控制，以管理复杂度 [20]。AllJoyn 中每个应用在安装/加入网络时，都要提交一个**权限清单 (Manifest)**，列出该应用提供的接口及其包含的成员，以及该应用可能调用/消费的接口列表 [20]。这个清单类似于手机APP声明所需权限。管理员在安全管理器上审核后，会为应用生成最终的Manifest，并将其摘要（哈希）写入应用的身份证书扩展域 [18]。因此证书不仅证明设备身份，也绑定了它宣称的权限范围。当运行时一个应用尝试调用另一应用的接口时，目标设备会检查调用方身份证书中的manifest摘要，与其提交的权限清单比对，验证调用方是否被授予了访问该接口的权限 [21]。只有权限匹配的请求才会被执行。这种模型实现了**集中授权、分布执行**：授权决定在安装时由安全管理者赋予，每台设备在运行时独立检查对方的证书和权限，不需要每次请求都询问中央服务器。这有效防止了超出授权范围的操作，提高了智能家庭环境中应用交互的安全性 [16] [20]。举例来说，一个智能门锁AllJoyn应用的证书manifest只允许它消费本家庭安防管理器接口，但不允许控制其他设备；如果恶意应用试图发锁定/解锁命令，由于其证书中不含门锁接口权限，门锁设备将拒绝执行。

安全缺陷与漏洞分析（AllJoyn）：AllJoyn 的安全设计较为完善，但在不同层面上也有一些值得关注的风险：

- 网络/传输层：AllJoyn 消息可以通过多种传输媒介。典型情况下，在本地网络内通过 UDP 进行发现（广播简介），再建立 TCP 连接进行数据总线通信（也可选用UDP传输模式）。由于AllJoyn应用层已加密，所以传输层的安全风险主要是拒绝服务类。比如攻击者若向AllJoyn路由服务发送畸形或恶意序列的总线消息，可能导致AllJoyn守护进程过载或崩溃。例如，微软 Windows 内置的 AllJoyn 路由服务曾曝出**输入验证错误**导致的 DoS 漏洞：远程攻击者可发送特制数据触发服务崩溃 [22]（CVE-2024-21438）。此外，由于AllJoyn需要在局域网内进行**发现**，攻击者可以利用频繁的假冒设备广播来扰乱AllJoyn设备的发现流程，耗尽其处理能力。虽然AllJoyn对发现有节流策略，但大规模局域攻击仍会造成通信延迟。

- 应用层协议：AllJoyn 本身协议复杂度较高，实现中可能存在传统软件漏洞。特别是 AllJoyn **路由节点** 需要处理来自多个对等方的消息，若消息解析存在溢出或内存错误，将成为被攻击的入口。2017年前后曾有安全研究者分析AllJoyn代码并提交漏洞报告。最近的案例是 2024 年披露的 Windows AllJoyn Router Service 信息泄露漏洞（**CVE-2024-38257**）：AllJoyn 路由服务在建立 ARDP（可靠数据流协议，AllJoyn自有的一个传输

层协议）会话时，可能发送包含未初始化内存的重置数据包，导致泄露路由进程内存信息 [23] [24]。该漏洞允许远程攻击者在无需身份验证的情况下获取服务内存中的敏感信息，已被微软修补 [25]。这表明即使框架提供了加密和认证，底层实现的不安全仍可能导致信息泄露等问题。

- 密钥管理与信任：AllJoyn 依赖 CA 签署证书建立信任。如果 **安全管理器**自身或根CA证书泄露，整个信任域的安全将崩溃——攻击者可伪造合法证书加入网络。虽然这种场景不属于协议漏洞，但在实际部署中必须严格保护CA私钥。另一个潜在问题是**证书吊销**：AllJoyn 网络通常离线运行，没有在线证书状态检查机制。如果某设备证书需要吊销，需要管理员主动在设备上更新信任列表。在这期间，被盗证书可能继续被接受。为此AllJoyn提供了Trust management接口用于更新信任anchor，但依赖管理员操作及时性。AllJoyn Security 2.0 通过trust zone隔离一定缓解了管理问题，但仍需注意。

- 权限策略错误：尽管AllJoyn提供了精细的权限控制，但**错误配置或过度授权**依然可能发生。例如开发者可能在manifest模板中列出过多接口，授予应用不必要的权限。如果恶意应用冒充合法应用拿到签名manifest，可能滥用其权限集合。研究者曾比较各IoT框架的权限模型，指出AllJoyn由于具备细粒度控制，反而存在被错误配置的隐患——不如简单模型直观 [26]。因此安全管理工具和UI需引导管理员谨慎赋权。另外，在 Security 1.0时代，一些厂商可能没有部署安全管理器、未使用身份证书（即运行在无安全模式下），这使AllJoyn通信缺乏任何加密和认证，完全暴露于本地网络的恶意设备。这实际上等同于关掉了安全功能，是严重误用。所幸随着Security 2.0普及，此类配置在新设备中已不常见。

实际误用与错误配置（AllJoyn）：AllJoyn 曾经因为其可选的安全模式而出现一些常见误用：（1）**未启用安全**：在早期AllJoyn设备中，开发者若没有集成安全框架，设备通信就处于明文且无认证状态。这种情形下攻击者在同一网络即可随意加入AllJoyn总线与设备交互。安全建议是始终启用Security 2.0，但仍需检查产品是否默认打开。（2）**不当的权限Manifest**：Manifest 的编制比较复杂，开发者可能遗漏必要权限导致功能异常，或者反之开放过多权限留下安全隐患。一些案例中，厂商为省事对所有应用发放了“通用”manifest，几乎等同于全开放，这背离了原则。（3）**忽视设备证书管理**：安全管理器需要在设备添加/移除时更新证书和权限。如果更换手机或管理员，忘记重新签发证书会导致旧设备仍持有权限，又或者新设备迟迟拿不到权限无法正常工作。（4）**密码握手使用弱PIN**：AllJoyn 支持基于PIN的ECDHE认证，如果用户设置的PIN码太简单（比如“123456”），攻击者可以暴力穷举握手，绕过认证直接建立会话。应使用足够长度和随机性的PIN并限制重试。此外，像 *ECDHE_NULL* 这种无认证模式应绝对禁止在生产中使用 [3]。总之，AllJoyn的安全能力很强，但只有在正确使用的前提下才能真正保障IoT环境安全。

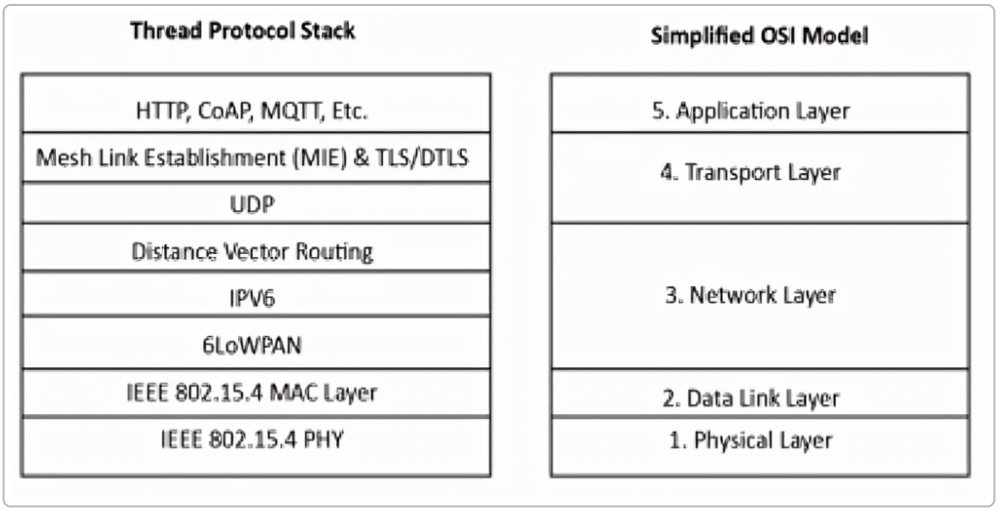
Thread 的安全架构与机制设计

Thread 是针对低功耗 mesh 网状网络设计的协议，工作在 IEEE 802.15.4 无线PHY/MAC层之上，为 IoT 设备提供了 IPv6 网络通信能力 [27]。与IoTivity和AllJoyn偏重应用层不同，Thread主要涵盖网络和传输层功能，但其规范也将安全作为核心要素，强调**网络层入网认证**和**链路加密** [28] [29]。Thread 协议栈由多个标准组成：物理和MAC层使用802.15.4（2.4GHz）；网络层使用6LoWPAN 压缩的 IPv6 和网状路由（基于IEEE 802.15.4的Mesh Link Establishment协议和Distance Vector路由）；传输层通常采用UDP（也可支持TCP）；应用层可以运行如CoAP、MQTT或更高层协议 [30] [31]。Thread 的安全机制体现在：

- **网络加入与设备认证**：Thread 网络通过一种**Mesh Commissioning Protocol (MeshCoP)**进行设备的安全接入（commissioning）。当新设备（Joiner）要加入Thread网络时，必须通过一个**加入凭证**（通常是6~8位的数字代码或QR码中的字符串）与网络管理员（Commissioner）进行握手认证 [32]。Commissioner可以是网络内的边界路由器或通过旁路通信（如BLE）与Joiner对接的临时设备。Thread 使用 **DTLS** 在加入过程中建立一个安全信道：Joiner和Commissioner基于用户提供的加入代码进行PSK身份验证，DTLS握手成功后，Commissioner会将网络的主加密密钥、安全策略等发送给Joiner [33]。只有通过正确凭证认证的新

设备才能获取网络密钥进入网络 [34] 。Thread 将这种新设备加入流程设计为 **用户主动触发**且短时间窗口内有效，以避免陌生设备未经授权潜入。 [35] 指出Thread网络中的所有加入必须由用户发起，加入完成后还需与Commissioner在应用层完成额外的安全认证步骤（例如交换Frame Counter防止重放） [36] 。这一系列机制确保了**网络边缘安全**：未经认证的设备无法加入网络，即使窃听到网络通信也因无密钥而无法解密。

- **链路层加密**：Thread 网络的所有**链路层帧**都使用网络范围共享的 **AES-128 密钥** 进行加密和认证 [37] 。这个密钥称为 **Network Key**，在设备入网时由Commissioner下发。基于IEEE 802.15.4 MAC 安全框架，Thread 帧采用 AES-CCM 模式加密，有效防御无线窃听和注入攻击 [37] 。同时，每台设备同邻居之间维护帧计数器，通过*MLE*协议互相交换并更新计数器，用于防御重放攻击 [38] 。也就是说，即便攻击者录制了*Thread*网络中的加密帧，重放也会因计数器失配而被拒绝。此外，*Thread* 设备在加入网络后会 随机化自己的*MAC*地址和*IPv6*地址 [39] （通过临时标识符）以提高隐私，避免敌手通过设备标识跟踪网络拓扑。需要注意的是，*Thread* 所有节点共享同一个*Network Key*，这意味着 一处失陷，全网受威胁*：如果攻击者攻破了一台设备并读取出Network Key，就能解密全网通信并以合法节点身份发送帧 [40] 。因此 Thread 建议配合应用层的端到端加密以提供第二道防线 [39] 。换言之，Thread 的链路加密主要防外部攻击，对内部恶意节点缺乏隔离机制。



Thread 协议栈与*OSI*模型对比示意图。左侧为*Thread*网络栈各层，包括物理层和*MAC*层（*IEEE 802.15.4*，无线链路加密）、*6LoWPAN*适配、*IPv6*网络层、网状路由（*MLE*/路由协议）、传输层*UDP*，以及可选的*Mesh Link Establishment*协议和*DTLS/TLS*安全层 [37] ；右侧为对应的*OSI*五层模型。*Thread*将安全功能内嵌于*MAC*层（全网共享密钥）和加入流程（*DTLS*认证），同时允许在传输层以上采用应用层安全协议实现端到端加密。

- **密钥管理与更新**：Thread 网络密钥可定期滚动更新（Key Rotation）以减少密钥泄露风险。Thread 定义了网络密钥的更换机制：新密钥可下发到所有节点，期间老旧密钥和新密钥会并存一段时间，确保网络平滑过渡，随后旧密钥作废。这要求所有设备正确实现密钥更新协议，否则不同密钥设备间将无法通信。此外，每对相邻节点之间Thread还有临时会话密钥（Link Key）用于单播时防止密钥重用（这在Thread 1.2版本中有所增强）。然而，Thread早期版本主要依赖全局Network Key，带来的隐患是**单点突破**：一旦攻击者获知Network Key，就能模拟加入网络。幸运的是，实务中可以通过硬件安全元件保护密钥存储，并使用调试锁定等手段防止密钥提取。但仍需考虑物理攻击场景，例如废弃设备若未重置清除，其闪存中的Network Key可能被恢复。Silicon Labs的OpenThread实现就曾曝出**密钥存储未加密**漏洞：CVE-2023-41095指出某版本的OpenThread SDK未对设备闪存中的安全参数（如Network Key）加密，攻击者可能修改或提取这些网络凭据 [41] [42] 。这个例子说明，协议虽允许安全实现，但实际装置的密钥管理细节（如是否加密存储）同样影响整体安全。

- **访问控制与路由安全**：Thread 协议本身不区分网络内不同设备的权限——加入网络的设备默认可以相互通信，路由器会为所有设备转发IPv6数据包。它**没有应用层访问控制**机制；权限控制需要由运行在Thread上的应用协议（例如OCF/IoTivity、Matter等）自行实现。因此，从Thread层面看，其安全边界停留在"谁能加入网络"和"网络外的人无法读取/注入数据"这两点。一旦设备在网内，Thread假设它是受信任的节点，可参与路由并访问网络中其他节点的IP接口。这意味着若有设备被攻陷成为"内鬼"，Thread无法在网络层加以阻止内部恶意流量。这是Mesh网络常见的信任模型局限。为此，一些学术分析建议结合**软件定义外围**或监测机制检测异常内部通信。Thread 1.1标准本身未包含入侵检测，但Thread 1.2引入了对设备属性的某些策略控制。除此之外，Thread网络的路由安全主要依赖于正确实现 RPL（Routing Protocol for Low power and Lossy Networks）的安全版本。不过，目前Thread采用较简化的距离向量路由，不同于Zigbee那样有信任中心，故路由欺骗的可能性不大，除非攻击者已获取网络密钥。

安全缺陷与漏洞分析（Thread）：Thread 相对新颖且规范严格，但仍存在一些安全隐患和已披露漏洞：

- 链路层和协议层攻击：由于Thread运行在2.4GHz IEEE 802.15.4，无线物理层固有的**干扰与阻断（干扰攻击）**在所难免。攻击者可以发射无线干扰信号导致Thread网络通信不稳（属于物理层DoS）。这种攻击无法通过协议解决，只能通过频跳或信道避让缓解。另外，Thread 虽用AES-CCM加密帧，但 **802.15.4规范的某些模式**可能被利用。例如上述 CVE-2023-2626 就揭示了 Thread 实现中存在**认证绕过**漏洞：攻击者可以构造使用"Key ID Mode 2"的特殊帧来绕过OpenThread边界路由器的安全检查 [43]。Key ID Mode 2 是 802.15.4 的一种密钥标识模式，使用静态全局加密密钥。由于OpenThread对这种模式处理不当，未认证节点也能通过此方式发送伪装成合法的数据帧进入Thread网络 [43]。利用该漏洞，攻击者可以在未加入网络的情况下与Thread设备双向通信，绕过原本的入网认证防护 [44] [45]。这一问题已在新版实现中修复，通过禁用不安全的密钥模式来堵住后门。

- 传输层及*IP*层：Thread 网络上的IP通信可能受到传统IP攻击影响，例如IPv6分片攻击（Thread采用6LoWPAN分片，若实现不当可能引发缓冲区溢出或重组拒绝服务）。幸而未有公开报告指明Thread栈存在此类漏洞。边界路由器（Border Router）作为Thread网络通向外部的网关，其实现安全性也至关重要。CVE-2024-3017 披露了Silicon Labs某多协议网关中OpenThread边界路由器的缓冲区漏洞，可导致内存破坏 [46]。这提醒我们需要重视边界路由器软件的安全审计，因为它既面对外网又连接Thread内网，是攻击面的交汇点。此外，如果Thread边界路由与云端通信未加TLS或验证证书，也会带来风险——不过这是超出Thread协议范围的实现问题。

- 应用层：Thread 本身不涉及应用层协议，但其主要目的是作为IP承载，为上层如Matter、OCF等提供通信。在应用组合中可能出现安全错配。例如，Thread保证了链路安全，但开发者误以为这足够安全，在应用层传输敏感数据却**未加密**，那一旦攻击者进入网络就可截获这些数据。因此最佳实践是在Thread之上再加一层端到端的安全（比如Matter应用层有自己的加密，会话也有ACL；OCF应用层也有ACL和认证）。Thread规范本身允许任何应用流量穿行，对于不安全的应用协议（如未认证的UDP命令），Thread并不阻拦。

- 实际漏洞事件：除了前述OpenThread参考实现的漏洞（CVE-2023-2626、CVE-2023-41095等） [43] [41] 外，Thread协议本身尚未出现严重设计漏洞。但学术界已指出Thread继承了IEEE 802.15.4的一些弱点，如：MAC层采用全网单密钥，设备被俘后密钥易泄露 [40]；又如若设备未妥善保存帧计数器，重放攻击有机可乘等。一些论文通过对比Zigbee和Thread，认为Thread虽改进了加入认证，但仍需警惕**内部人攻击**以及**侧信道**（如功耗分析提取密钥）等非常规手段。总的来看，Thread在安全性上比前代802.15.4网状协议有显著提升，但也不是万无一失。

实际误用与错误配置（Thread）：Thread网络常见的错误使用主要涉及网络凭证管理和设备生命周期：（1）**弱加入凭证**：Thread加入码若设得过于简单（例如常见的短PIN且未限制尝试次数），攻击者可以尝试穷举正确的凭

证。应使用足够长度的随机加入码并在每次尝试失败后增加延迟，以防暴力破解。（2）**默认网络密钥**：Thread规范要求每个网络有唯一的Network Key。但如果厂商在多个设备上出厂预置了相同的默认密钥（用于简化首次部署），那么不同家庭的网络可能使用同一密钥，非常危险。一旦密钥曝光，攻击面将不限于单个网络。正确做法是随机生成初始网络密钥，并在部署完成后通过Commissioner要求用户更改。（3）**未及时移除设备**：当某设备退网或遗失时，管理员应通过边界路由器将其从网络中移除并触发Network Key更新。懒于管理可能导致丢失设备仍保有密钥，从而成为攻击入口。（4）**忽视硬件安全**：正如前述CVE-2023-41095所揭示的，如果芯片厂商未对Thread密钥做安全存储，加上IoT设备经常物理暴露在外，攻击者可以通过调试接口、flash读取等方式提取密钥。因此生产设备应启用安全存储、调试锁定等措施。（5）**缺少应用级验证**：一些开发者以为有了Thread的MAC加密就无需额外认证，结果让设备直接信任所有来自Thread网络内的消息。这种配置忽视了内部攻击的可能性。在安全要求高的场景，应在Thread基础上再加应用层身份验证机制（例如消息签名或挑战响应）来验证发送方的合法性。

三种协议安全设计的对比总结

为了直观比较 IoTivity、AllJoyn 和 Thread 在安全架构上的差异，下面表格总结了它们在协议层次、加密机制、身份认证、访问控制等方面的主要特征：

比较维度	IoTivity (OCF)	AllJoyn	Thread
定位层次	应用层框架 + 传输协议（基于IP/CoAP）[1]； 提供设备资源模型和设备间交互标准。	应用层消息总线框架（类D-Bus）； 提供设备发现、RPC通信的统一机制。	网络/传输层协议（IPv6网状网络）； 提供底层联网和路由，应用层由上层协议实现。
基础通信协议	CoAP/UDP (默认) + 可扩展到MQTT、HTTP 等； 传输加密采用 DTLS [6]。	AllJoyn路由协议（采用自定义的Reliable UDP或直接TCP通信）； 自带加密认证，不依赖底层TLS。	IEEE 802.15.4 无线 + 6LoWPAN + IPv6 + UDP； 可跑任意IP上层协议（常用CoAP/MQTT等）。
身份认证机制	所有权转移(Ownership Transfer)：Just Works（无验证ECDH）[3]、PIN码（对密码ECDH）、X.509证书（三方CA签名）[4]; 设备有OCF分配的UUID作为身份，PKI模式下设备持有制造商证书。	基于PKI的信任域；每设备由安全管理器颁发身份证书[16]; 支持多种ECDHE认证：无认证（测试用）、PSK、密码/Pin、证书（主用）[3]; Security 2.0统一为证书认证（ECDHE_ECDSA）[10]。	加入网络需提供网络凭证（加入码）； 通过DTLS PSK握手验证加入者身份[35]; 成功加入后无单独身份标识，信任由网络密钥共享体现（网络即身份）。

比较维度	IoTivity (OCF)	AllJoyn	Thread
加密算法与密钥	ECDHE P-256 密钥交换; 对称密钥用于 DTLS AES-CCM 通信[8]; 支持对称PSK和非对称证书两种密钥体系; 每对设备建立独立会话密钥，组播有组密钥。	ECDHE P-256 + AES-CCM [8]; 长期密钥为设备的ECC私钥 + 可信根CA; 会话密钥每次对话临时生成; 证书中嵌入权限清单哈希用于授权[18]。	所有节点共享网络密钥（AES-128）[37]用于MAC层加密; 入网DTLS握手基于加入码派生PSK; 支持网络密钥轮换; 邻居间维护帧计数防重放[38]。
数据加密传输	基于DTLS的端到端加密（CoAP报文加密）[6]; 支持设备-云TLS连接; 未secure标记的资源走明文（需避免）。	应用层总线消息加密（无论底层TCP/UDP）; 总线所有消息用会话密钥AES加密签名传输; 防中间人篡改，保障隐私。	链路层AES加密（802.15.4帧）[37]; 网络外无法嗅探明文; 端到端加密需由应用实现（如Matter、OCF over Thread等自行加密）。
访问控制模型	访问控制列表(ACL)每设备存储，基于设备ID授权资源操作[12]; 权限粗粒度（读/写），需在代码中为资源标注安全并配置ACL[13]; 支持角色证书。	Manifest权限清单 + 安全管理器集中签发; 细粒度到接口/方法级[20]; 每请求由目标验证调用方证书和manifest[21]; 可实现复杂策略，如只允许特定设备调用特定功能。	无应用层访问控制; 网络层只控制加入资格; 一旦入网，设备可自由发送IP数据包; 可通过边界路由器隔离Thread子网对外通信权限，但网内不控流。
已知安全事件	*DDoS*放大攻击: CVE-2019-9750 CoAP接口可被滥用进行DDoS[14]; 默认不安全配置: 未启用SECURED编译标志导致通信明文[9]; 目前无已知远程攻破或执行漏洞报告。	*Windows AllJoyn*服务拒绝: CVE-2024-21438 输入验证错误导致DoS[22]; 信息泄露: CVE-2024-38257 AllJoyn路由发送未初始化内存导致泄密[23][24]; 早期设备若未用Security 2.0，则通信明文易被嗅探。	认证绕过: CVE-2023-2626 非法帧可绕过OpenThread边界路由认证[43]; 密钥保护不足: CVE-2023-41095 Silabs实现未加密存储网络密钥[41]; 常规物理层干扰、内网渗透等威胁也需考虑。
常见误配置	未开启安全模式/资源未标记安全导致明文传输; 误用Just Works于生产环境（无认证易MITM）[3]; 所有设备使用同一出厂PSK或证书; ACL配置过于宽泛（权限滥用）。	未部署安全管理器（跳过证书认证直接信任所有通信）; manifest权限过度（应用拿到不必要高权限）; 弱PIN用于设备Claim（易被猜测）; 证书不更新或吊销不及时（离职设备仍有权限）。	默认加入码简单（易被暴力猜测）; 多个网络使用相同初始密钥; 遗忘移除密钥泄露的设备; 未使用硬件安全存储密钥; 认为MAC加密即可，不再使用应用层加密认证。

上述对比可以看出：**IoTivity** 和 **AllJoyn** 都提供了端到端的安全架构（认证、加密、访问控制）并偏重应用层互操作，但实现方式不同——IoTivity基于DTLS和ACL，AllJoyn基于PKI证书和manifest。**Thread** 则专注于网络层安全，通过链路加密和加入控制保障网络通信基础，但不涉及具体应用指令级的授权。IoTivity与AllJoyn在目标上类似，都希望实现异构设备间**安全互联互操作**，但IoTivity依赖标准化的OCF模型（REST风格，资源-方法），AllJoyn采用分布式总线（事件/方法调用）。安全上，IoTivity更接近传统Web/DTLS模型，而AllJoyn引入了**集中授权+去中**

心校验的新颖机制。Thread的出现其实可以与应用框架互补：正如有厂商将OCF (IoTivity)跑在Thread网络上，以Thread提供低层安全传输，OCF负责高层认证授权 [47] [48]。

结论

物联网安全架构需要从**全栈视角**来审视：底层网络链路需要加密和设备准入控制，上层应用协议需要严格的身份认证和权限管理。IoTivity、AllJoyn 和 Thread 三种协议各自覆盖了这一纵深防御中的不同层面。IoTivity (OCF) 提供了开放标准的设备互操作框架，借助TLS级加密和ACL访问控制保障设备间交互安全，但依赖正确配置来生效，其"默认关闭安全"策略在部署时必须警惕 [9]。AllJoyn 则通过成熟的PKI体系和细粒度权限模型实现了强大的端到端安全，适合本地自治的物联网环境，但对管理流程和正确配置要求较高；它的集中授权概念为智能家居安全提供了借鉴模板 [16] [20]。Thread专注于低功耗设备的网络接入和传输安全，它为上层协议打下了一个"封闭且加密"的网络环境，阻挡了大部分外部攻击 [37]，但不解决内部授权问题，需要与诸如IoTivity/AllJoyn/Matter等配合以实现应用级安全。

综合来看，没有一种单一协议能包揽IoT系统从底至顶的所有安全需求：**Thread** 可作为联网和密钥分发的基础设施，**IoTivity/AllJoyn** 等提供应用层互通和权限管理，两者可以集成（OCF 已支持在Thread网络上运行其框架 [47]）。对于物联网安全研究者，IoTivity和AllJoyn丰富的安全特性值得深入分析比较，而Thread的设计则体现了对低层安全的重视及对Zigbee教训的改进。实际部署中，应充分利用各协议的安全机制，例如启用IoTivity的安全模式并正确配置ACL、使用AllJoyn Security 2.0的证书机制、设置复杂的Thread加入凭证并定期轮换网络密钥等。同时也需注意各协议已曝光的漏洞，及时打补丁升级版本（如更新OpenThread堆栈以修复认证绕过 [43]，打上操作系统针对AllJoyn服务的补丁 [23] 等）。总之，保障物联网安全需要**分层防御、优势互补**：网络层阻止非法接入和窃听，传输层确保数据机密完整，应用层严格控制操作权限。IoTivity、AllJoyn、Thread 三种协议的协同运用，加上开发者安全意识和正确实践，才能织就物联网设备与云端交互的安全防护网。

---

[1] IoTivity vs AllJoyn - what is the difference? - Stack Overflow
https://stackoverflow.com/questions/27947856/iotivity-vs-alljoyn-what-is-the-difference

[2] [4] Tools - IoTivity
http://iotivity.org/tools/

[3] [8] [10] [11] [17] [19] rwc.iacr.org
https://rwc.iacr.org/2016/Slides/AllJoyn-RWC2016.pdf

[5] [6] [7] [9] [13] [15] [16] [18] [20] [21] [26] Security Implications of Permission Models in Smart-Home Application Frameworks - InfoQ
https://www.infoq.com/articles/smart-home-permission-models/

[12] [PDF] OCF Security Primer for Device Vendors
https://openconnectivity.org/wp-content/uploads/2018/06/4.-Security-Introduction-Architecture.pdf

[14] CVE-2019-9750 Detail - NVD
https://nvd.nist.gov/vuln/detail/CVE-2019-9750

[22] Denial of service in Microsoft AllJoyn API
https://www.cybersecurity-help.cz/vdb/SB20240312324

[23] [24] [25] CVE-2024-38257 Impact, Exploitability, and Mitigation Steps | Wiz
https://www.wiz.io/vulnerability-database/cve/cve-2024-38257

27 28 ripe88.ripe.net
https://ripe88.ripe.net/presentations/80-2024-05-22-Systron-Lab_presentation-at-RIPE-IoTWG_V2.pdf

29 33 35 36 37 38 39 40 UG103.11: Thread Fundamentals
https://www.silabs.com/documents/public/user-guides/ug103-11-fundamentals-thread.pdf

30 31 32 The Thread Protocol: Redefining IoT Connectivity for a Smarter World
https://thinkpalm.com/blogs/the_thread_protocol_redefining_iot_connectivity_for_a_smarter_world/

34 Thread protocol - Zephyr Project Documentation
https://docs.zephyrproject.org/latest/connectivity/networking/api/thread.html

41 42 CVE-2023-41095: Missing Encryption of Security Keys in Silicon Labs OpenThread SDK
https://www.clouddefense.ai/cve/2023/CVE-2023-41095

43 44 45 NVD - CVE-2023-2626
https://nvd.nist.gov/vuln/detail/CVE-2023-2626

46 CVE-2024-3017 Detail - NVD
https://nvd.nist.gov/vuln/detail/cve-2024-3017

47 48 cascoda.com
https://www.cascoda.com/wp-content/uploads/2021/12/Cascoda-over-Thread-whitepaper.pdf