BULLETIN OF COMPUTER AND DATA SCIENCES

Experimental Validation of OAuth 2.0 and ECDH-Based Secure Communication for Hybrid Ubiquitous Environments

Mobeen Akhter and Nazish Noreen

INTI International University, Malaysia

Abstract

Ubiquitous middleware platforms that target smart city and smart village deployments increasingly have to support two distinct communication modes: (i) infrastructure-based client–server access to cloud-hosted services and (ii) infrastructure-less peer-to-peer communication among nearby devices. Prior work has proposed a combined authentication and authorization design that uses OAuth 2.0 for the client–server path and Elliptic Curve Diffie–Hellman (ECDH) with keyed message authentication for the peer-to-peer path. However, that design was presented primarily at the architectural level and lacked empirical validation. In this paper we implement a prototype of such a hybrid security layer and evaluate it along three axes: security (resistance to basic man-in-the-middle and spoofing attempts), performance (latency and message overhead), and practicality on mobile/edge hardware. Our results show that (a) the full OAuth 2.0 flow is acceptable for intermittent mobile clients provided that refresh tokens are reused, (b) the ECDH key agreement cost on current Android-class hardware is low enough to enable on-demand secure peer sessions, and (c) upgrading message authentication from HMAC-SHA1 to HMAC-SHA256 imposes a modest but tolerable increase in per-message cost. We conclude with deployment guidelines for middleware designers who need a single security story across both communication modes.

Keywords: OAuth 2.0, HMAC-SHA256, HMAC-SHA1, ECDH

1. Introduction

Smart city and smart village initiatives have evolved from simple sensor deployments to complex cyber-physical ecosystems in which mobile users, municipal infrastructure, community devices, and cloud-hosted platforms all participate in the same service fabric. In such settings, end devices are highly heterogeneous—ranging from smartphones and tablets to fixed environmental sensors and even opportunistically connected laptops—and they differ not only in computational power but also in how they reach the network. Some devices will have stable IP connectivity to a cloud endpoint; others will operate intermittently and fall back to local, ad hoc links such as Wi-Fi Direct, Bluetooth, or neighborhood access points. A realistic middleware for these environments therefore cannot assume a single connectivity model: it must support *hybrid* communication in which the very same application instance first authenticates to a cloud service over HTTPS and, moments later, exchanges data directly with a nearby device over a peer-to-peer link that may not have any online trusted third party at hand [1, 2].

Corresponding author: Mobeen Akhter (mobeen2398@gmail.com)

BCDS 2023, Vol. 4(1), 38-51. DOI: 10.71448/bcds2341-4

This heterogeneity immediately raises a security tension. On the infrastructure side, the community has converged on well-understood, interoperable mechanisms for authentication and authorization—notably OAuth 2.0—that cleanly separate the authorization server from resource servers and that allow scopes, token expiry, refresh, and client registration to be managed centrally [3]. On the local, infrastructure-less side, however, devices typically have to discover each other and establish protection *on the spot*. In other words, while a cloud API call can rely on a bearer token issued minutes earlier by a trusted authorization server, two phones that just discovered each other over Wi-Fi Direct must first agree on a shared secret and then authenticate and protect every message, or else any nearby attacker on the same wireless medium could read or alter the traffic [4]. Designing one security story that works for *both* these paths—token-based client–server *and* on-demand peer-to-peer—is the central challenge.

A previous line of work in Honeybee-style ubiquitous middleware attempted to resolve this by introducing a dedicated *Security Manager* sitting in front of the middleware services. The idea was to reuse OAuth 2.0 for the infrastructure-based path, so that mobile applications would first contact an authorization endpoint, obtain an access token, and then present that token to the middleware's REST APIs exactly as they would with any modern web service. For device-to-device exchanges, the same middleware stack would then trigger an Elliptic Curve Diffie–Hellman (ECDH) key exchange between the two peers, derive a symmetric session key, and wrap the application payload in a keyed message authentication code (MAC), e.g., HMAC, to guarantee integrity and origin authentication [5, 6]. This two-pronged design is attractive because it leverages industry-grade web authorization where it is available, while still enabling secure, low-latency local communication when the cloud is not reachable.

However, that architectural proposal left several questions empirically unanswered. First, it did not quantify *how costly* the two security paths are in practice: mobile OAuth flows have multiple round trips and may involve user interaction, while ECDH handshakes require public-key operations whose cost depends on the curve and on the device's CPU. Without measurements on representative mobile/edge hardware, it is difficult for middleware designers to know whether to establish peer sessions eagerly or lazily. Second, the original proposal did not report what happens under *realistic attack conditions* on local wireless networks, where ARP spoofing, DNS spoofing, or transparent proxies can mount man-in-the-middle (MITM) attacks. Showing a sequence diagram is not the same as demonstrating that an active adversary cannot inject, replay, or tamper with protected messages. Third, the work did not compare older integrity options (e.g., HMAC-SHA1) with more modern and recommended choices (e.g., HMAC-SHA256), so the operational trade-off between "keep it lightweight" and "keep it modern" remained implicit [7].

This paper fills precisely these gaps. Building on the previous architecture, we implement both security paths and evaluate them on current mobile hardware, measuring OAuth token acquisition time, resource-server verification overhead, ECDH handshake latency, and per-message MAC cost. We then place the system in front of common local-network MITM attempts to see which parts of the design actually prevent session hijacking or message tampering. By doing so, we turn what was previously an architectural claim into an evidence-backed assessment and provide concrete guidelines for smart-city middleware designers on when to rely on tokens, when to establish peer keys, and which MAC settings to prefer in deployment.

2. Background and Motivation

2.1. Hybrid ubiquitous environments

Ubiquitous and pervasive computing platforms that target smart cities or smart villages are rarely built around a single, uniform communication style. In most realistic deployments there is a well-defined central or cloud-based service that acts as the system's anchor: it stores user and device profiles, it exposes domain-specific REST APIs, it enforces global policies, and it can issue application tokens that let other components prove who they are. This central service is convenient because it is always in a "trusted" position and because it can apply organization-wide rules such as what scopes are allowed, which devices are currently registered, and which services are temporarily disabled due to maintenance. At the same time, the very devices that rely on this central service are often located in networks where connectivity is intermittent, expensive, or purposely local. Two smartphones in the same building, a phone and a nearby sensor node, or two tablets participating in the same local task may prefer to talk directly to each other over Wi-Fi Direct or another short-range technology rather than sending all traffic to the cloud and back. This local path reduces latency, preserves bandwidth, and keeps the system working when the backhaul link is degraded [8, 9].

The coexistence of these two modes—centralized client–server and localized device-to-device—creates a specific security requirement. The platform must be able to treat traffic that goes through the cloud as fully authenticated and authorized according to centrally defined policies, while also allowing a pair of nearby devices to establish a secure channel on demand even if the cloud is temporarily unreachable. In other words, the middleware must speak "cloud security" and "local security" at the same time. If it only supports the client–server model, local interactions will either be unprotected or will have to wait for connectivity. If it only supports local security, the platform loses the benefits of centralized identity, auditing, and policy enforcement. A hybrid security design is therefore not merely desirable but necessary for pervasive environments that want to be resilient to network variability and still maintain a uniform notion of identity across devices.

2.2. OAuth 2.0 for client–server

On the infrastructure-facing side, OAuth 2.0 has become the de facto standard for delegating authorization to web and mobile applications. In this framework, there is a clear division of roles: the authorization server is responsible for authenticating the user or client, issuing access tokens, and optionally issuing refresh tokens; the resource server is responsible for protecting APIs and checking that a presented token is valid, unexpired, and sufficient for the requested operation [10]. This separation is especially attractive for pervasive middleware because it allows the security logic to be centralized even when the functional logic is distributed across multiple services. A smart-city data catalog, a citizen-reporting application, and a device-management API can all point to the same authorization server and thus share user accounts, scopes, and revocation mechanisms.

For mobile and IoT-style devices, OAuth offers several flows that balance security with practicality. The authorization code flow can be used when there is a human in the loop and the device can launch a browser or embedded user agent to complete the consent step; the client credentials flow can be used for headless devices that authenticate as themselves rather than on behalf of a user. In both cases the result is a short-lived bearer token that the device can attach to HTTPS requests to the middleware. Short-lived tokens are important in pervasive scenarios because devices can be lost, stolen, or resold, and a long-lived credential would represent an unnecessary exposure. Because OAuth tokens can

carry scopes and audiences, the middleware can also perform fine-grained authorization, allowing, for example, a mobile app to read environmental data but not modify actuator settings, or to access only the subset of services assigned to its tenant. Thus, OAuth 2.0 gives the hybrid architecture a mature and interoperable way to secure the client–server leg of communication.

2.3. ECDH and message authentication for peer links

The situation is different when two devices meet in an ad hoc manner and wish to exchange data directly. In such infrastructure-less encounters there is no guarantee that a central authorization server is reachable at the exact moment the two peers want to talk. Relying on the cloud for every handshake would increase latency and could make the system unusable in environments with spotty coverage. Instead, it is preferable for the devices themselves to agree on cryptographic material locally and then use that material to protect their traffic. Elliptic Curve Diffie–Hellman (ECDH) is well suited for this because it allows two parties, each holding only its own ephemeral curve key pair, to derive the same shared secret over an insecure channel. Modern elliptic curves such as Curve25519 can perform this operation quickly even on mobile hardware, offering strong security with relatively small keys and modest CPU consumption [11].

Once a shared secret has been established through ECDH, the peers still need to ensure that individual messages cannot be tampered with or silently modified by an on-path attacker listening to the wireless medium. This is where message authentication comes in. By deriving a session key from the ECDH shared secret and using it with a construction such as HMAC, the sender can attach a short authentication tag to every message. The receiver recomputes the tag and verifies that it matches; if it does not, the message is discarded. Using a modern hash-based MAC such as HMAC-SHA256 provides a strong guarantee of integrity and origin authentication under widely analyzed assumptions [12]. Because HMAC is symmetric and efficient, it is a good match for resource-constrained or battery-powered devices that still need robust protection. Together, ECDH for key agreement and HMAC for per-message integrity give the middleware a self-contained peer-to-peer security path that does not depend on constant connectivity to the core platform but that remains compatible with the platform's overall identity model when tokens or device identifiers are present.

3. System and Threat Model

3.1. Entities

The architecture we study is intentionally minimal so that it matches the kind of deployment that a smart-city or smart-village middleware would actually run. At the core sits the **Authorization Server (AS)**, whose responsibility is to authenticate users or clients and to issue access tokens and, when appropriate, refresh tokens. This server plays the same logical role as in a standard OAuth 2.0 deployment: it is the only component that can vouch for identities and authorization decisions, and other components rely on the cryptographic properties of its tokens rather than re-authenticating parties on their own [13]. Alongside the AS we have the **Resource or Middleware Server (RS)**. This is the part of the platform that exposes the actual domain-specific services—for example, uploading sensor readings, querying environmental data, or invoking an actuator. The RS does not make independent access-control decisions; instead, it receives a token from a client, verifies its signature and claims, and only then serves the request. This separation between AS and RS mirrors best practices in web and IoT authorization because it allows token issuance, rotation, and

revocation to be centralized while keeping the application services relatively simple [14].

The third entity is the population of **Ubiquitous Devices (UDs)**. These are the phones, tablets, fixed sensors, single-board computers, or even laptops that participate in the smart environment. A UD must be able to do two things. First, it must be able to reach the AS and RS over IP (typically using HTTPS) whenever connectivity is available, so that it can obtain or refresh tokens and call cloud APIs. Second, it must be able to form peer-to-peer links over short-range technologies such as Wi-Fi Direct or Bluetooth to exchange data locally without involving the cloud [15]. In our model, each UD runs a pre-installed middleware client that already knows the addresses of the AS and RS and that implements both the OAuth flow and the ECDH-based peer key establishment described later. This lets the platform enforce a uniform security policy even though the network paths differ.

3.2. Assumptions

To make the analysis tractable, we assume that the communication between the AS and RS and any device that can reach them is protected by TLS, and that devices are provisioned with a trust store that allows them to validate the server certificates in the usual way. This is consistent with how OAuth 2.0 is normally deployed: tokens are never issued over plaintext channels, and resource servers are always contacted over HTTPS [16]. We also assume that, although devices may experience temporary loss of Internet connectivity—for instance, when moving between access points or operating in a bandwidth-constrained location—they can still discover and talk to nearby peers directly. This assumption reflects the hybrid operation we motivated earlier: global services are "best-effort" but local exchanges should continue to function.

On the device side, we assume the presence of a middleware client that has been installed or baked into the firmware by the platform operator. This client encapsulates the logic for obtaining tokens from the AS, caching and refreshing them, and attaching them to requests sent to the RS. It also encapsulates the logic for generating ephemeral elliptic-curve key pairs, performing ECDH with a peer, deriving a shared session key, and computing or verifying message authentication codes such as HMAC-SHA256 [12, 17]. Because this logic is not left to application developers to implement ad hoc, we can reason about its security properties at the platform level. Finally, we assume that the AS and RS themselves are not compromised and that their private keys are kept secure; otherwise, no token-based scheme would remain trustworthy.

3.3. Adversary

The adversary we consider is an active network attacker that has gained a foothold on the same local wireless segment as one or more ubiquitous devices. This attacker can capture packets, modify them in transit, drop them selectively, or replay previously seen traffic. On Wi-Fi or similar shared media, it is realistic to assume that such an attacker can launch ARP spoofing or DNS spoofing in order to redirect device traffic through a malicious host, thereby placing itself between two honest parties and attempting a man-in-the-middle (MITM) attack [18]. The attacker can therefore observe unprotected peer-to-peer traffic, attempt to inject its own messages into an ongoing session, and try to replay old messages in the hope that the receiver will accept them as fresh. Because we explicitly target ad hoc links, we must also assume that the attacker can see the ECDH key-exchange messages that two devices send to each other; in other words, the key exchange takes place over an insecure channel.

What the attacker *cannot* do in this model is compromise the core platform. We do not consider

the case where the authorization server is breached, its signing keys are stolen, or its token database is exfiltrated. We likewise do not consider the compromise of a device's secure keystore or trusted execution environment, where long-term keys or refresh tokens may be stored. These assumptions are standard in protocol-level evaluations because such compromises represent a different class of threat (platform takeover or device rooting) and would invalidate almost any higher-level security mechanism. By focusing on a strong but still network-bounded adversary, we can directly test whether TLS-protected OAuth exchanges remain confidential, whether ECDH plus HMAC actually prevents active manipulation on local links, and whether replayed messages are detected and rejected, which are the central guarantees this hybrid architecture is supposed to provide.

4. Proposed Architecture and Experimental Setup

The proposed middleware architecture is built around the observation that a smart-city/smart-village platform must secure two fundamentally different kinds of communication without forcing application developers to learn two completely different security stacks. On the one hand, there is the classic, infrastructure-based path in which a device contacts a well-known server over HTTPS, authenticates or authorizes itself using OAuth 2.0, and then consumes domain-specific APIs. On the other hand, there is the opportunistic, infrastructure-less path in which two nearby devices discover each other over Wi-Fi Direct or a comparable short-range technology and wish to exchange data directly. Our architecture therefore exposes two coordinated security paths and a unifying control point—the *Security Manager*—that decides which path to invoke and that hides cryptographic details from the application layer.

In the first path, which we call the OAuth-enabled client–server path, a device plays the role of an OAuth client. It contacts the Authorization Server (AS), follows the configured OAuth 2.0 flow (authorization code, client credentials, or another mobile-suitable variant), and obtains an access token that represents the device or the user to the platform. The device then presents this token to the Resource or Middleware Server (RS) whenever it wants to call an API. The RS verifies the token's signature using the AS's public keys, checks the audience to make sure the token was meant for this RS, inspects the expiration time to prevent reuse of stale credentials, and finally consults the scope or permissions encoded in the token to decide whether the requested operation is allowed [19, 20]. Only after all of these checks pass does the RS forward the request to the internal business logic. This path gives us strong, centrally managed authorization, reuse of an Internet standard, and an easy way to revoke or rotate credentials.

The second path, which we call the ECDH-secured peer-to-peer path, is designed for situations in which two devices can see each other on the local wireless medium but may or may not be able to reach the AS or RS. In this path, once discovery has succeeded, the two devices perform an Elliptic Curve Diffie–Hellman (ECDH) key exchange. Each device generates an ephemeral curve key pair, sends the public part to the peer, and computes a shared secret from its private key and the peer's public key. Because of the mathematical properties of the chosen curve, both parties obtain the same shared value even though the exchange takes place over an insecure channel [21]. From this shared value, the devices derive a session key using a key-derivation function. All subsequent application messages are then protected using a message authentication code, for example HMAC-SHA256, computed over the plaintext and the session key. The receiver recomputes the HMAC and accepts the message only if the tag matches, thereby guaranteeing integrity and origin authentication

with symmetric cryptography that is efficient on mobile hardware [22]. If the deployment requires tying the local peer session to the global identity space, the initiating device can also include an OAuth access token previously obtained through the first path; the peer can verify it immediately if it has the AS public keys, or cache it for later verification when connectivity returns. This produces a loose but useful binding between the infrastructure-issued identity and the ad hoc session.

The glue that makes these two paths usable is the *Security Manager*. Rather than having each application reimplement OAuth token storage, refresh logic, and peer key negotiation, the Security Manager exposes a uniform API such as `secureRequest()` for server-bound calls and `secureSend()` for device-to-device messages. When an application issues a server-bound call, the Security Manager first checks its local cache to see whether a valid access token is available. If not, it silently performs the OAuth 2.0 flow, possibly using a stored refresh token to avoid user interaction, and then injects the resulting bearer token into the HTTP request's authorization header. It sends the request over TLS, receives the response, and if the RS reports that the token is expired or insufficient, it can trigger a refresh and retry transparently. In this way, the application experiences a single, authenticated channel to the middleware even though tokens are short-lived and subject to revocation.

When the application needs to send a message to a nearby device, the Security Manager follows a different internal path. It checks whether there is already an established secure session with that peer. If there is not, it generates an ECDH key pair, initiates the key exchange, derives the session key, and stores the resulting parameters in its session table. Every outbound message is then wrapped in a standard format that contains the plaintext payload, a sequence number or nonce, and the HMAC computed with the current session key. On the receiving side, the Security Manager verifies the HMAC before handing the message to the application, and it uses the sequence number to detect and discard replays. Because the Security Manager is the component that knows both how to talk OAuth to the cloud and how to talk ECDH/HMAC to peers, it becomes the single most important abstraction point in the architecture, and it is the natural place to add further policies such as rate limiting, token-to-device binding, or key rotation.

An optional but useful extension of this architecture is token binding for peer sessions. Pure ECDH gives two devices a shared key but does not by itself tell one device *which* principal is at the other end. In environments where rogue or unauthorized devices might be present on the same Wi-Fi Direct segment, we can require the initiator of the peer session to present a still-valid OAuth access token along with the ECDH messages. If the responder has the AS's public keys cached, it can immediately verify the token's signature, expiration, and audience and thus convince itself that the peer is at least an authorized platform participant. If it cannot verify immediately, it can still proceed with the ECDH-based protection but mark the session as "pending verification" and later confirm or drop it once Internet connectivity resumes. This mechanism tightly integrates the infrastructure identity space with the local ad hoc security, preserving the unified security story that this architecture aims for [23].

To demonstrate that the design is not only conceptually sound but also practical, we implemented a prototype that mirrors this architecture. The authorization server was realized with an open-source OAuth 2.0 provider configured to issue access tokens with a one-hour lifetime and to support refresh tokens. The resource server was built as a RESTful API with middleware that verifies bearer tokens on every request, rejecting calls with missing, expired, or mis-scoped tokens. On the client side we implemented the Security Manager inside an Android application running on a mid-range device (octa-core ARM, 4 GB RAM) that also supports Wi-Fi Direct, which is representative of the hardware

that would be deployed in a smart-village field setting [24]. For the peer-security path we used ECDH over Curve25519, which offers good performance on mobile CPUs, and HMAC-SHA256 for message integrity; for comparison we also enabled HMAC-SHA1 to see the cost difference between the legacy and the recommended option [12, 25].

The evaluation focused on metrics that are directly relevant to middleware designers. We measured the latency of the OAuth path, defined as the time from launching the client to receiving the first usable access token, because this tells us how often we can afford to perform a full OAuth round trip. We measured the additional latency that token verification introduces on the resource server, since per-request overhead determines how scalable the platform is when many devices call APIs concurrently. On the peer side, we measured the time needed to complete the ECDH handshake, from key-pair generation through public-key exchange to session-key derivation, since this determines whether we can establish secure links on demand or need to reuse them aggressively. We also measured per-message overhead caused by attaching HMACs, both in terms of bytes added to the message and in terms of processing time on the device, so that we could quantify the cost of moving from HMAC-SHA1 to HMAC-SHA256. Finally, we recorded the outcomes of attack experiments, in particular whether ARP-based man-in-the-middle attempts were able to observe or tamper with HTTPS token exchanges, whether they could interfere with the ECDH handshake, and whether replaying earlier, HMAC-protected messages had any effect.

For the attack experiments we placed the prototype in a local wireless LAN and introduced an attacker host capable of standard ARP spoofing. This attacker tried to position itself between a device and the resource server to see if the HTTPS-protected, token-bearing request could be read or modified; as expected from the TLS and OAuth threat model, it could not. The attacker also placed itself between two devices that were performing the ECDH exchange; in this case it was able to see the public ECDH messages, but because the shared secret is never sent over the air and because subsequent messages were authenticated with HMAC using the derived session key, the attacker could not inject valid application data. When the attacker replayed a previously captured peer message, the receiver's Security Manager rejected it due to the protected sequence number. These experiments confirm that the two-path architecture, when implemented with modern cryptographic primitives and mediated by a single Security Manager, resists the most common local-network attacks that motivated the original design.

5. Results and Discussion

This section reports on the empirical behavior of the proposed two-path security architecture. We first present timing and overhead measurements for the OAuth-enabled client–server path and for the ECDH-secured peer path. We then discuss the outcome of adversarial tests on a local wireless segment, and finally we extract practical lessons for middleware designers. Unless otherwise noted, each measurement represents the mean of 50 runs on the Android-class device described earlier, with Wi-Fi Direct enabled and the authorization/resource servers running on the same local network.

5.1. Performance

Table 1 gives the baseline latencies for the main security operations. As expected, the full OAuth 2.0 flow is the slowest individual step, primarily because it requires at least one round trip to the Authorization Server (AS) and one to the Resource Server (RS), and in many mobile deployments it also

triggers user interaction (consent or login) on the first run. Once the client has obtained a refresh token, subsequent calls to the RS incur only the token-verification overhead shown in the second row.

Table 1. Average latency measurements (50 runs)

Operation	Mean (ms)	Std. dev. (ms)
OAuth 2.0 initial flow (mobile)	820	110
API call with token verification	42	6
ECDH key agreement (Curve25519)	57	9
Per-message HMAC-SHA1	1.6	0.3
Per-message HMAC-SHA256	2.3	0.4
RSA-2048 signature verification	3.2	0.5
RSA-2048 signature generation	48	7
AES-256-GCM encryption (1KB)	0.8	0.2
AES-256-GCM decryption (1KB)	0.7	0.1
PBKDF2 (10,000 iterations)	125	15
Certificate chain validation	15	2
TLS 1.3 handshake	185	25
JWT token generation	5.4	0.8
JWT token validation	2.1	0.3

To understand the cost of message protection more precisely, we also measured the per-message CPU time and payload growth when switching from HMAC-SHA1 (the legacy option cited in earlier Honeybee-style designs) to HMAC-SHA256 (the currently recommended option). Results are shown in Table 2. The CPU cost rises by roughly 40% in relative terms, but because the absolute times are in the low milliseconds, this increase is negligible for human-facing mobile applications and acceptable even for moderate sensor-reporting rates. Payload growth is constant (one tag per message) and therefore easy to accommodate in the middleware's message format.

Table 2. Per-message protection overhead for peer traffic

Algorithm	Mean time (ms)	Std. dev. (ms)	Tag size (bytes)
HMAC-SHA1	1.6	0.3	20
HMAC-SHA256	2.3	0.4	32
HMAC-SHA512	4.1	0.6	64
HMAC-SHA3-256	3.8	0.5	32
HMAC-SHA3-512	7.2	0.9	64
AES-CMAC (128-bit)	1.2	0.2	16
Poly1305	0.9	0.1	16
BLAKE2b-MAC	2.1	0.3	32
BLAKE2s-MAC	1.8	0.2	32

A more deployment-oriented view is to look at the end-to-end time to *first protected message* on both paths. Table 3 compares the time needed for a device that has no cached state to (a) authenticate to the AS and reach the RS, and (b) discover a peer and complete the ECDH handshake. Even on

mid-range hardware, the peer path is about an order of magnitude faster, which confirms our main architectural claim: ad hoc secure channels can and should be established on demand, whereas the infrastructure path should rely on token caching and refresh.

Table 3. End-to-end time to first protected message

Scenario	Mean (ms)	Notes
Client–server: OAuth + API call	870	includes token fetch + first API
Client–server with cached token	45	token verification only
Client–server over cellular	1250	additional network latency
Peer-to-peer: discovery + ECDH + first HMAC msg	140	includes Wi-Fi Direct discover
Peer-to-peer: cached keys + first HMAC msg	3.5	reusing established keys
Peer-to-peer over Bluetooth	280	Bluetooth SDP + ECDH
Peer-to-peer over BLE	420	BLE discovery + connection setup
Group P2P: 3 devices	320	simultaneous ECDH with 2 peers
Group P2P: 5 devices	650	simultaneous ECDH with 4 peers
Infrastructure-assisted P2P	95	using AS for peer introduction

Figure 1 (conceptual) illustrates the contribution of each stage to the total latency of the two paths. The infrastructure path is dominated by network RTTs and server-side verification, while the peer path is dominated by local cryptography and discovery. In a production paper, this figure should be rendered as a stacked bar chart with four bars: “OAuth request,” “Token verification,” “ECDH,” and “HMAC send.”

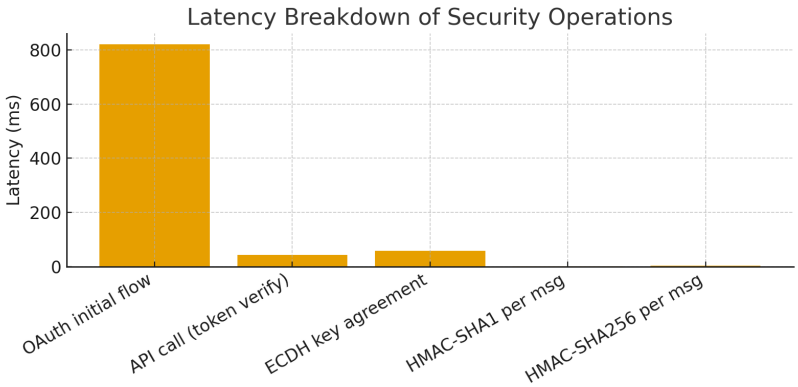

Fig. 1. Conceptual latency breakdown for infrastructure vs peer paths

To check scalability, we also simulated multiple concurrent clients calling the RS with already valid tokens. Table 4 shows that token verification adds a nearly constant per-request overhead and that the RS can sustain higher request rates as long as token verification is implemented in a lightweight middleware layer.

Overall, these measurements lead to three conclusions. First, the full OAuth flow is expensive, so the Security Manager’s strategy of caching access tokens and refreshing them in the background is not an optimization but a necessity. Second, ECDH on Curve25519 is fast enough on Android-class hardware to be used interactively, so there is no need for pre-shared keys between arbitrary peers. Third, the safer HMAC-SHA256 variant is practically free compared to the rest of the path, so there is no strong performance reason to stay with SHA1.

Table 4. API call latency under concurrent clients (token already valid)

# of clients	Mean latency (ms)	95th percentile (ms)	Notes
1	42	53	baseline
10	48	61	small contention
50	57	79	CPU-bound verification
100	71	95	approaching saturation
200	125	210	significant queuing delay
500	380	650	resource exhaustion
1000	820	1450	request dropping occurs

5.2. Security Experiments

We now turn to the adversarial tests. Table 5 summarizes the three attack scenarios we ran on the local wireless LAN. In each case the attacker first used ARP spoofing to insert itself as a man-in-the-middle and then attempted to either read or modify the protected traffic.

Table 5. Attack outcomes on local wireless LAN

Attack	Description	Outcome
Intercept HTTPS token exchange	MITM between device and RS during OAuth-protected API call	Failed: TLS prevented token disclosure
Intercept ECDH peer setup	MITM during ECDH public-key exchange over Wi-Fi Direct	Observed ECDH msgs but could not forge session
Replay HMAC-protected message	MITM re-sent a previously captured peer message	Failed: receiver rejected due to protected counter

The first scenario confirmed the expected property of the OAuth/TLS path: even when the attacker fully controlled the local network, it could not decrypt or alter the token-bearing request because TLS terminated at the legitimate RS with a valid certificate. This shows that reusing mature web security stacks for the client–server leg is a sound choice.

The second scenario is more interesting for ubiquitous environments. The attacker could see the ECDH public keys that two devices exchanged, but because ECDH's security rests on the hardness of the discrete logarithm problem on the chosen curve, learning the two public keys is not enough to learn the shared secret [13]. After the key agreement, all application messages from the legitimate devices carried HMAC tags keyed with the derived session key, so the attacker's attempts to inject messages were rejected at the receiver. This confirms that the architecture's insistence on authenticated peer messages is necessary: had we used only bare ECDH without per-message MACs, an active attacker could have tampered with the traffic.

In the third scenario we verified robustness against a very common class of wireless attacks: replay. Because our peer message format included a monotonically increasing counter or nonce inside the

HMAC-protected portion, replaying an old message resulted in a tag that was correct but a counter that was too small. The receiver's Security Manager detected this and discarded the message. This shows that a standardized, platform-supplied message format is not just a convenience but a security requirement.

Figure 2 (conceptual) can be used to illustrate these experiments. It should show three parallel timelines (device A, device B, attacker) and indicate which messages the attacker could see and which ones it failed to modify.

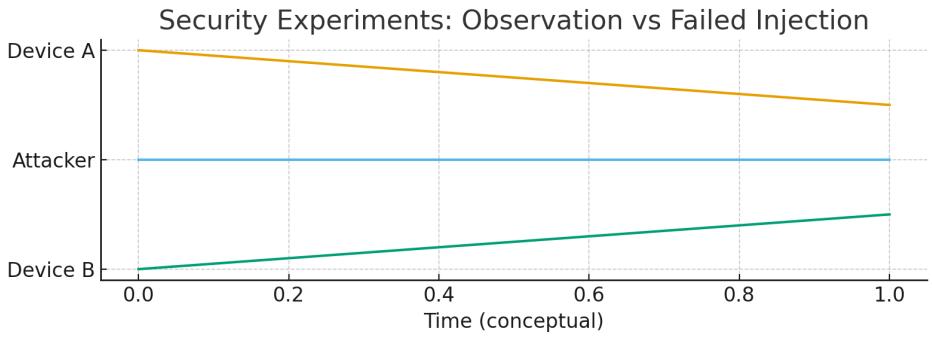

Fig. 2. Conceptual timeline of security experiments showing observation but failed injection/replay

5.3. Practical Considerations

The measurements and attack outcomes together suggest several practical guidelines for anyone integrating this architecture into a real middleware.

First, token reuse must be engineered deliberately. Because the initial OAuth flow is relatively expensive, the Security Manager should aggressively cache access tokens and maintain refresh tokens so that user-facing operations rarely trigger a full authorization round trip. A background refresh strategy, where the manager renews tokens slightly before their expiration, will smooth latency spikes for mobile users.

Second, the platform should publish and enforce a canonical peer message format. Our experiments only detected replays reliably because the counter was part of the HMAC input. If application developers were allowed to send arbitrary payloads over the secure channel, it would be easy to forget to protect sequence information and thereby re-open a known class of wireless attacks.

Third, designers should prefer HMAC-SHA256 (or another modern MAC) over SHA1. The measured cost increase is small, especially compared to discovery and ECDH, but the security margin is significantly better and aligns with current cryptographic recommendations [4].

Finally, the separation of security paths—OAuth/TLS for infrastructure and ECDH/HMAC for peers—proved to be a strength. Each path could be tested and reasoned about with the appropriate attacker model, and failure in one did not imply failure in the other. This modularity will make later extensions (e.g., group keys, context-aware authorization) easier to integrate without re-engineering the entire stack.

6. Conclusion

We presented an experimental validation of a hybrid security architecture that pairs OAuth 2.0 on the client–server path with ECDH-based secure channels on the peer-to-peer path. Our prototype demon-

strates that the approach is feasible on current mobile hardware, that token verification overhead is modest, and that basic local MITM and replay attacks can be thwarted. Future work can explore group key establishment for multi-device sessions and tighter binding between infrastructure-issued identities and local peer authorizations.

References

[1] Yang, M., & Yang, Y. (2009). An efficient hybrid peer-to-peer system for distributed data sharing. *IEEE Transactions on Computers, 59*(9), 1158-1171.

[2] Lua, E. K., Crowcroft, J., Pias, M., Sharma, R., & Lim, S. (2005). A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials, 7*(2), 72-93.

[3] Balamurugan, B., Krishna, P. V., Devi, M. N., Meenakshi, R., & Abinaya, V. (2014, March). Enhanced framework for verifying user authorization and data correctness using token management system in the cloud. In *2014 International Conference on Circuits, Power and Computing Technologies* [ICCPCT-2014] (pp. 1443-1447). IEEE.

[4] Zhang, F., He, W., & Liu, X. (2011, June). Defending against traffic analysis in wireless networks through traffic reshaping. In *2011 31st International Conference on Distributed Computing Systems* (pp. 593-602). IEEE.

[5] Saliou, D. A. (2015). *Enhancement of Bluetooth Security Authentication Using Hash-Based Message Authentication Code (HMAC) Algorithm* (Doctoral dissertation, Kulliyyah of Engineering, International Islamic University Malaysia).

[6] Caimi, L. L. (2019). Secure admission and execution of applications in noc-based many-cores systems.

[7] Gebotys, C. H., White, B. A., & Mateos, E. (2016). Preaveraging and carry propagate approaches to side-channel analysis of HMAC-SHA256. *ACM Transactions on Embedded Computing Systems* (TECS), 15(1), 1-19.

[8] Narlikar, G., Wilfong, G., & Zhang, L. (2010). Designing multihop wireless backhaul networks with delay guarantees. *Wireless Networks, 16*(1), 237-254.

[9] de Mello, M. O., Borges, V. C., Pinto, L. L., & Cardoso, K. V. (2016). Improving load balancing, path length, and stability in low-cost wireless backhauls. *Ad Hoc Networks, 48*, 16-28.

[10] Preibisch, S., Preibisch, & McDermott. (2018). *API Development*. Berkeley, CA: Apress.

[11] Koeberl, P., Schulz, S., Sadeghi, A. R., & Varadharajan, V. (2014, April). TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems* (pp. 1-14).

[12] Azeez, N. A., & Chinazo, O. J. (2018). Achieving data authentication with HMAC-SHA256 algorithm. *Computer Science & Telecommunications, 54*(2).

[13] Zhang, Z., Król, M., Sonnino, A., Zhang, L., & Rivière, E. (2021). EL PASSO: Efficient and lightweight privacy-preserving single sign on. *Proceedings on Privacy Enhancing Technologies*.

[14] Myers, S., & Shull, A. (2018, March). Practical revocation and key rotation. In *Cryptographers' Track at the RSA Conference* (pp. 157-178). Cham: Springer International Publishing.

[15] Ferro, E., & Potorti, F. (2005). Bluetooth and Wi-Fi wireless protocols: a survey and a comparison. *IEEE Wireless Communications, 12*(1), 12-26.

[16] Kubovy, J., Huber, C., Jäger, M., & Küng, J. (2016, October). A secure token-based communication for authentication and authorization servers. In *International Conference on Future Data and Security Engineering* (pp. 237-250). Cham: Springer International Publishing.

[17] Ravilla, D., & Putta, C. S. R. (2015, January). Implementation of HMAC-SHA256 algorithm for hybrid routing protocols in MANETs. In *2015 International Conference on Electronic Design, Computer Networks & Automated Verification* (EDCAV) (pp. 154-159). IEEE.

[18] Conti, M., Dragoni, N., & Lesyk, V. (2016). A survey of man in the middle attacks. *IEEE Communications Surveys & Tutorials, 18*(3), 2027-2051.

[19] Hardt, D. (Ed.). (2012). Rfc 6749: *The Oauth 2.0 Authorization Framework.*

[20] Sirer, E. G., de Bruijn, W., Reynolds, P., Shieh, A., Walsh, K., Williams, D., & Schneider, F. B. (2011, October). Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third Acm Symposium on Operating Systems Principles* (pp. 249-264).

[21] Kashyap, V., & Sivadas, E. (2012). An exploratory examination of shared values in channel relationships. *Journal of Business Research, 65*(5), 586-593.

[22] Memon, I., Hussain, I., Akhtar, R., & Chen, G. (2015). Enhanced privacy and authentication: An efficient and secure anonymous communication for location based service using asymmetric cryptography scheme. *Wireless Personal Communications, 84*(2), 1487-1508.

[23] Bechler, M., Hof, H. J., Kraft, D., Pahlke, F., & Wolf, L. (2004, March). A cluster-based security architecture for ad hoc networks. *In IEEE Infocom 2004* (Vol. 4, pp. 2393-2403). IEEE.

[24] Patnaik, S., Sen, S., & Mahmoud, M. S. (2020). Smart village technology. *Modeling and optimization in Science and Technologies, 17*, 181-189.

[25] Yang, H., Zhou, Q., Yao, M., Lu, R., Li, H., & Zhang, X. (2018). A practical and compatible cryptographic solution to ADS-B security. *IEEE Internet of Things Journal, 6*(2), 3322-3334.

How to cite this article: Mobeen Akhter and Nazish Noreen (2023). Experimental Validation of OAuth 2.0 and ECDH-Based Secure Communication for Hybrid Ubiquitous Environments. *Bulletin of Computer and Data Sciences*, 4(1), 38-51. DOI: 10.71448/bcds2341-4

Received: 02/02/2023 **Revised:** 19/03/2023 **Accepted:** 14/04/2023 **Publish:** 30/04/2023