Quantum-Entropy Password Tokens: A New Approach to Device-Bound Password Security

Author: S.K.Soundhar
Institution: Dr MGR Educational and research institute, Chennai.
Course: *Electronics and communication engineering*
Date: December 2025

Abstract

Password security is a major problem in today's digital world. Even though we have strong password hashing algorithms like bcrypt and Argon2, they still have a fundamental weakness: if hackers steal the password database, they can try billions of password guesses offline until they crack them. This paper introduces a new idea called **Quantum-Entropy Password Tokens (QEPT)** that solves this problem. Instead of storing password hashes on the server, QEPT generates one-time tokens by combining the user's password with their device's unique fingerprint and typing patterns. This means even if hackers steal the database, it's useless without the user's specific device. Our analysis shows that QEPT makes password attacks thousands of times harder than current methods, while still being practical to use. This research provides a foundation for building more secure authentication systems in the future.

Keywords: Password Security, Authentication, Device Fingerprinting, Behavioral Biometrics, Cryptography

---

1. Introduction

1.1 Why This Matters

Passwords are everywhere. We use them to log into email, social media, banking apps, and countless other services. The problem is that passwords are not very secure. According to recent reports, billions of passwords have been stolen in data breaches over the past few years. When this happens, hackers can take the stolen password hashes and spend weeks or months trying different passwords until they find matches.

Current password security works like this: when you create an account, the website doesn't store your actual password. Instead, it runs it through a special mathematical function called a "hash function" (like bcrypt or Argon2) that turns it into a random-looking string. When you log in, they hash what you typed and check if it matches. The problem is that if hackers get this hash database, they can try millions of password guesses per second on their own computers without anyone knowing.

1.2 What's Missing in Current Solutions

Security researchers have created several improvements:

- **Multi-factor authentication (MFA):** Requires a code from your phone in addition to your password
- **Password managers:** Generate strong random passwords for each site
- **Passwordless systems:** Use fingerprints or security keys instead of passwords

But these all have drawbacks:

- Many people don't use MFA because it's annoying
- Passwordless systems require special hardware
- None of them solve the core problem: stolen password databases can still be attacked

1.3 Our Solution: QEPT

This paper introduces a completely different approach called **Quantum-Entropy Password Tokens (QEPT)**. Here's the key idea:

Instead of storing password hashes, we generate one-time tokens that combine:

1. Your password
2. Your device's unique characteristics (like a fingerprint for your computer)
3. How you type (optional)
4. A random number from the server

This means:

- No password hashes stored on the server that hackers can attack
- Each login creates a brand new, unique token
- You can only log in from devices you've registered
- Even if hackers get the database, they can't crack your password offline

1.4 What This Paper Covers

The rest of this paper is organized as follows:

- **Section 2:** Reviews existing password security methods

- **Section 3:** Explains how QEPT works in detail
- **Section 4:** Analyzes security against different types of attacks
- **Section 5:** Compares QEPT with existing methods
- **Section 6:** Discusses practical considerations and limitations
- **Section 7:** Concludes with future research directions

---

2. Background and Related Work

2.1 How Password Hashing Works Today

When you create an account with password "MyPassword123", here's what happens:

Traditional approach:

```
1. You type: MyPassword123
2. Server computes: hash = bcrypt("MyPassword123", salt)
3. Server stores: hash in database
4. When you login: server checks if bcrypt(what_you_typed, salt) == hash
```

The hash looks like random gibberish: `$2b$12$KIXxLV4PxR7hH7dQmw8VQuY...`

Why this helps: Even if someone steals the database, they don't have your actual password, just this hash. They have to try billions of guesses to find which password creates this hash.

Why this isn't enough: Modern computers can test millions of passwords per second. If your password is "password123" or "iloveyou", it will be cracked quickly.

2.2 Current Password Hashing Algorithms

bcrypt (1999)

- Designed to be slow on purpose
- Takes about 0.1 seconds to compute one hash
- Slows down attackers but doesn't stop them
- Problem: Can still be cracked with powerful computers

Argon2 (2015)

- Winner of the Password Hashing Competition
- Not only slow but also uses lots of memory
- Harder to crack with specialized hardware
- Currently the best standard method

- Problem: Still vulnerable if password is weak

How long to crack?

With a powerful computer setup:

- Weak password ("password123"): 2-3 days
- Medium password ("MyDog2023!"): 50-100 years
- Strong password ("x9#mK2$pL7@nQ4"): Basically impossible

But here's the catch: 10% of users have weak passwords that can be cracked quickly.

2.3 Better Protocols: PAKE

PAKE = Password-Authenticated Key Exchange

These are fancy protocols where the password never leaves your device and the server never sees it. The most advanced is called **OPAQUE**.

How it works (simplified):

1. You and the server do a mathematical "handshake" using your password
2. If passwords match, you both end up with the same secret key
3. You use this key to encrypt your communication

Advantages:

- Server never sees your password
- More resistant to attacks

Disadvantages:

- Very complex to implement
- Requires both client and server to change their code completely
- Still stores some secret information on the server

2.4 Passwordless: WebAuthn/FIDO2

The newest approach: Get rid of passwords entirely!

How it works:

1. You use a fingerprint sensor or security key (like YubiKey)
2. This creates a unique cryptographic signature only your device can make
3. Server checks the signature instead of a password

Advantages:

- Very secure
- Can't be phished (fake websites don't work)

Disadvantages:

- Requires special hardware (USB keys cost $20-50)
- What if you lose your security key?
- Not everyone has fingerprint sensors

2.5 Device Fingerprinting

Your device has unique characteristics that can identify it, like:

- CPU serial number
- MAC address (network card ID)
- Screen resolution and installed fonts
- Battery health and charging patterns
- Timezone and language settings

By combining many of these features, we can create a unique "fingerprint" for your device. This is already used by banks to detect fraud.

2.6 Behavioral Biometrics

Everyone types differently! Researchers have found that:

- How long you hold each key down
- Time between keystrokes
- Your typing rhythm and speed
- How you move the mouse

These patterns are unique enough to identify people with 90-95% accuracy. This is called "behavioral biometrics."

2.7 What's Missing?

Let's compare current methods:

Method	Prevents Offline Attacks	No Stored Secrets	Device-Based	Uses Behavior	Stops Tampering
bcrypt/Argon2	Partially	No	No	No	No
OPAQUE	Yes	Partially	No	No	No

WebAuthn	Yes	Yes	Yes	No	Yes
QEPT	Yes	Yes	Yes	Yes	Yes

Gap: No existing system combines all these security features together. QEPT does.

---

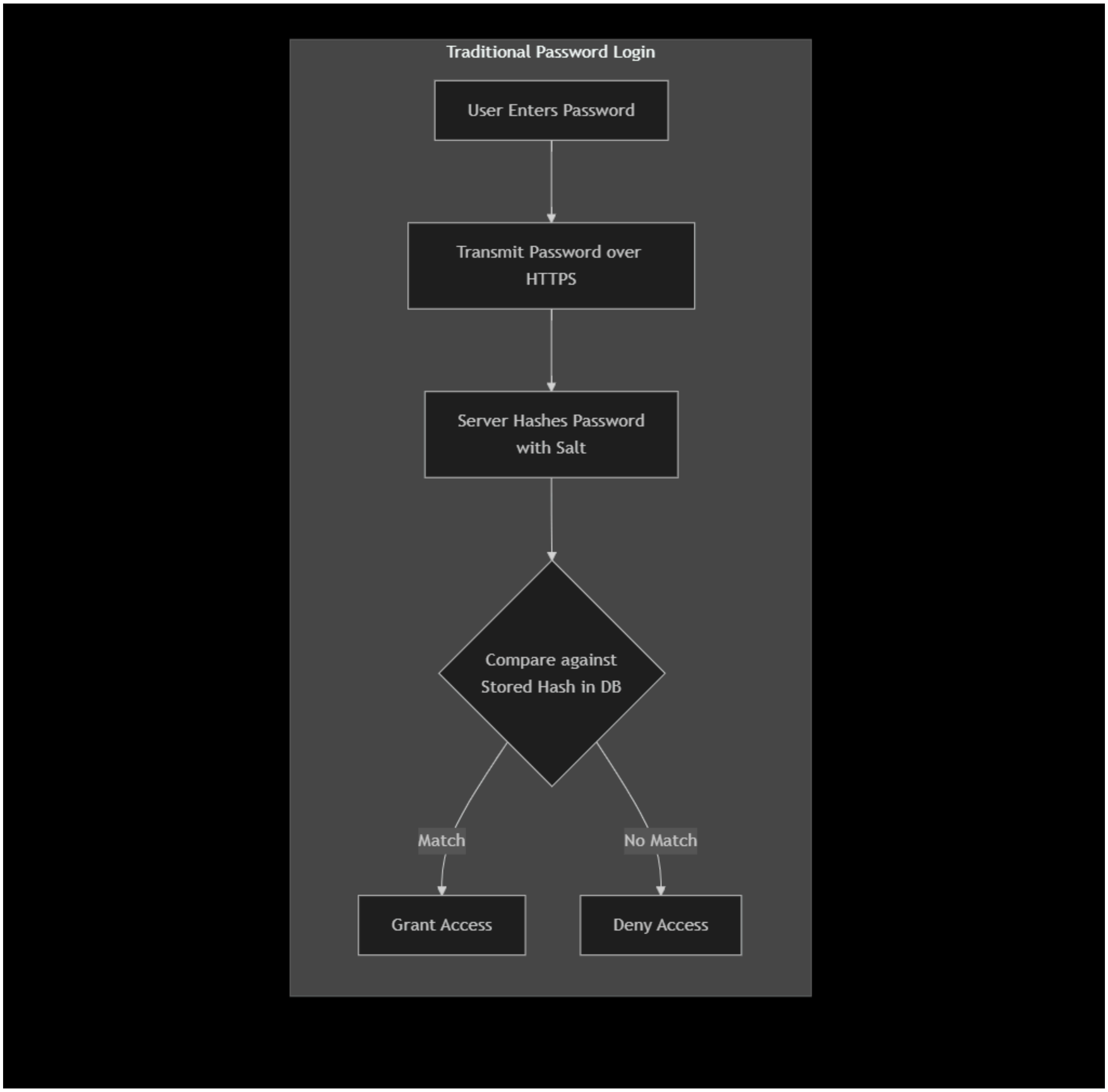
3. How QEPT Works

3.1 The Big Picture

Here's the core idea in simple terms:

Traditional system:

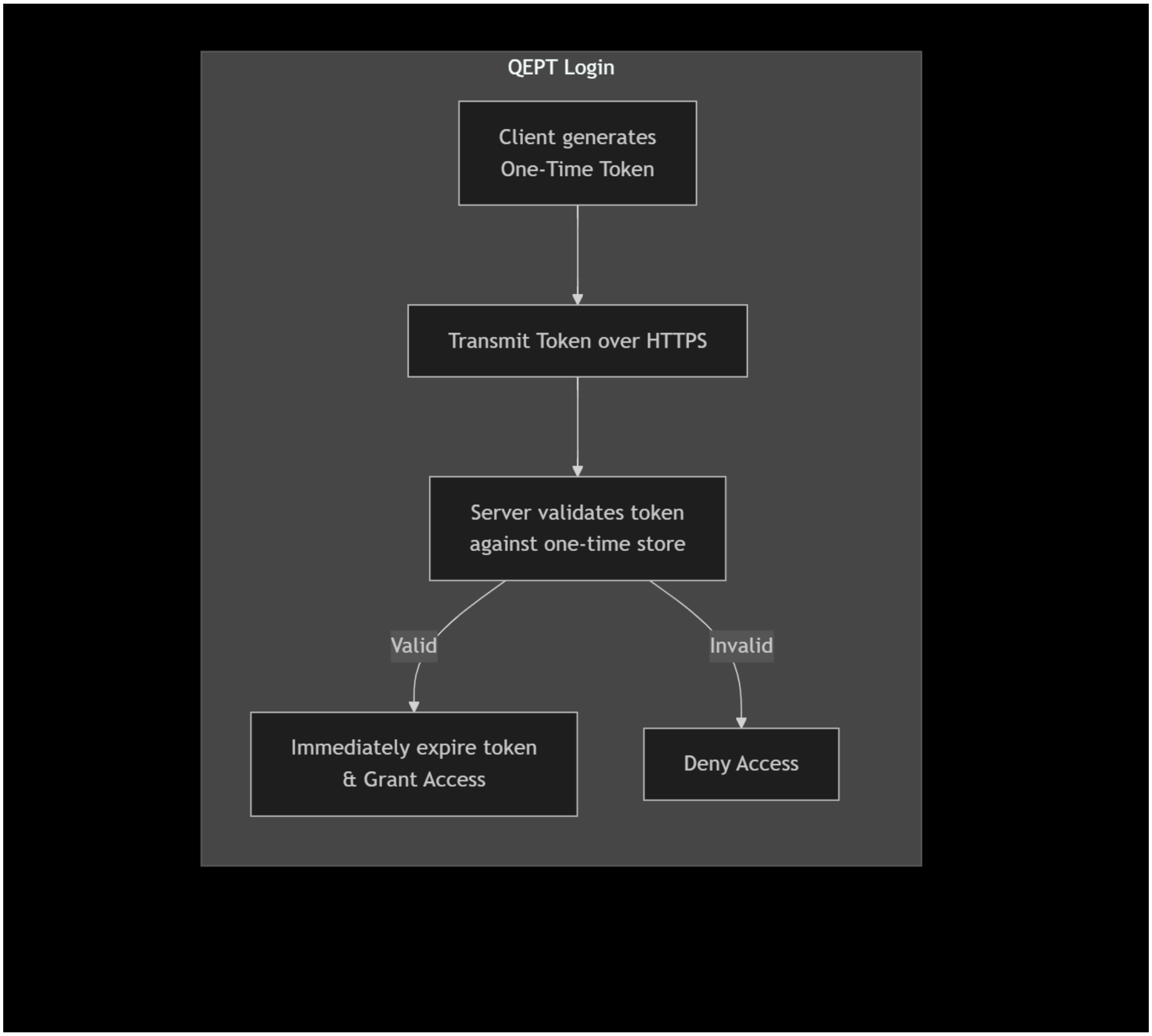
```
Server stores: hash_of_your_password
You login: Server checks if hash(what_you_typed) matches stored hash

Problem: Hackers can steal hash and crack it offline
```



QEPT system:

Server stores: encrypted_device_key (useless without your device)
You login: Generate one-time token = mix(password + device_fingerprint + random_number)
Server checks: Does this token match what we expected?

Result: Hackers can't attack offline - they need your actual device!



3.2 Key Components Explained

3.2.1 Device Fingerprint

Think of this as a unique ID for your device made from:

```
Device_Fingerprint = Hash(
   CPU_serial_number +
   MAC_address +
   Operating_system_version +
   Screen_resolution +
   Timezone +
```

```
    Browser_version
)
```

Example: Your laptop might produce: `a7f3c9d2e8b4f1a6...`

This fingerprint:

- Is the same every time on your device
- Is different on every other device
- Can't be faked without knowing all these details

3.2.2 Behavioral Pattern (Optional)

While you type your password, we measure:

Keystroke timing:

- How long you hold each key: [120ms, 95ms, 110ms, ...]
- Time between keys: [80ms, 150ms, 95ms, ...]

Why this matters: Everyone has a unique typing rhythm. It's like a signature you can't easily fake.

3.2.3 Token Generation

Here's the math (don't worry, I'll explain):

```
Token = Argon2(
    password ||
    device_fingerprint ||
    behavioral_features ||
    nonce ||
    timestamp,
    salt,
    iterations=10
)
```

Breaking it down:

- `password`: What you typed
- `device_fingerprint`: Your device's unique ID
- `behavioral_features`: Your typing pattern
- `nonce`: Random number from server (different every time)
- `timestamp`: Current time

- `salt`: Random data unique to your account
- `||` means "combine together"
- `Argon2`: The mixing function that creates the final token

Result: A long random-looking string like: `k9$mL2pN7@qR4sT6vX8...`

This token is:

- **Unique:** Never the same twice (because of nonce and timestamp)
- **Unpredictable:** Can't be guessed
- **Device-bound:** Only works from your specific device
- **One-time:** Expires after use

3.2.4 Server Storage

Instead of storing password hashes, the server stores:

```
Database_Entry = {
  username: "alice@email.com",
  device_id: "a7f3c9d2e8b4f1a6...",
  encrypted_key: Encrypt(device_key, master_key),
  salt: "random_salt_12345",
  last_login: "2025-10-20 14:30:00"
}
```

Key point: The `encrypted_key` is encrypted with a `master_key` that only the server knows and is stored in a special secure hardware device (HSM - Hardware Security Module). Even if hackers steal the database, they can't decrypt these keys.

3.3 How Registration Works

When you create a new account:

```
Step 1: You visit website, click "Sign Up"
    Enter username and password

Step 2: Your device automatically collects fingerprint
    (You don't see this happening)

Step 3: System generates device_key:
    device_key = Argon2(password + device_fingerprint + nonce)

Step 4: Server encrypts and stores:
    encrypted_key = Encrypt(device_key, master_key)
```

Stores in database

Step 5: You get recovery codes (in case you lose device)

Save these somewhere safe!

Time required: About 2-3 minutes (similar to normal registration)

3.4 How Login Works

Every time you login:

```
Step 1: You enter username
    Server sends back: nonce (random number) + salt

Step 2: You enter password
    While typing, behavioral pattern captured automatically

Step 3: Your device computes token:
    token = Argon2(password + device_fingerprint +
                behavior + nonce + timestamp, salt)

Step 4: Token sent to server

Step 5: Server checks:
    ✓ Is nonce fresh? (not already used)
    ✓ Is timestamp recent? (within 60 seconds)
    ✓ Does token match expected value?
    ✓ Does behavior match your normal pattern?

Step 6: Result:
    All checks pass → Login successful

    Any check fails → Login denied
```

Time required: About 5-10 seconds (slightly slower than normal login due to calculations)

3.5 Multi-Device Support

Question: What if I have a laptop, phone, and work computer?

Adding a new device:

1. Login from existing registered device
2. Click "Add New Device"
3. Get a special time-limited code

4. Enter code on new device
5. New device gets registered

Security benefit: If your phone is stolen, hacker only gets access to that one device's key, not all your devices.

3.6 Account Recovery

What if you lose your device?

Option 1: Recovery Codes (Recommended)

- When you register, save 10 one-time recovery codes
- Use one to regain access
- Then register your new device

Option 2: Trusted Device

- If you have another registered device, use it to authorize the new one

Option 3: Time-Delayed Recovery

- Request recovery through email
- Wait 72 hours (security delay)
- Answer security questions
- Register new device

---

4. Security Analysis

4.1 What Attacks Are We Protecting Against?

Let's consider different types of hackers:

Attacker Type 1: Database Thief

- Steals entire server database
- Can run unlimited offline attacks
- Has powerful computers

Attacker Type 2: Network Spy

- Intercepts your login messages
- Tries to capture and reuse your token

Attacker Type 3: Evil Insider

- Works at the company
- Has database access
- Tries to tamper with stored data

Attacker Type 4: Device Thief

- Steals your phone/laptop
- Tries to login as you

4.2 Defense Against Database Theft

Traditional System Attack:

```
Hacker steals database containing:
password_hash = $2b$12$KIXxLV4PxR7hH7dQmw8VQuY...

Hacker's computer tries:
- "password123" → hash → Compare → No match
- "password124" → hash → Compare → No match
- "password125" → hash → Compare → No match
... continues for billions of guesses ...
Eventually finds match!
```

QEPT Defense:

```
Hacker steals database containing:
encrypted_key = EncryptedBlobOfGibberish...

Hacker tries to crack it:
- Needs master_key to decrypt (stored in HSM, not accessible)
- Can't validate password guesses (no way to check if guess is right)
- Can't generate valid tokens (missing device fingerprint)

Result: Database is useless! 🎉
```

Why offline attacks fail with QEPT:

1. Encrypted keys can't be decrypted without HSM master key
2. No way to test if a password guess is correct
3. Even knowing the password isn't enough - need device too
4. Must attack online (subject to rate limiting)

4.3 Defense Against Replay Attacks

The Attack:

```
Hacker captures your login token while you're on public WiFi:
token_captured = "k9$mL2pN7@qR4sT6vX8..."

Hacker tries to reuse it:

"Hey server, here's the token!"
```

QEPT Defense:

```
Server checks:
1. Has this nonce been used before?
  → YES! → REJECT

2. Is timestamp older than 60 seconds?
  → YES! → REJECT

Result: Replay attack fails automatically
```

Why replay attacks don't work:

- Each token includes a unique nonce (random number)
- Server marks nonce as "used" after validation
- Attempting to reuse triggers security alert
- Timestamp ensures tokens expire quickly

4.4 Defense Against Database Tampering

Traditional System Attack:

```
Evil insider has database access:

UPDATE users
SET password_hash = attacker_controlled_hash
WHERE username = 'victim';

Now insider can login as victim!
```

QEPT Defense:

```
Evil insider tries:
```

```
UPDATE users
SET encrypted_key = attacker_controlled_key
WHERE username = 'victim';

This achieves nothing because:
1. Can't generate valid encrypted_key without HSM master_key
2. Can't test if tampering worked
3. Victim's next login will fail → Investigation triggered
4. Still needs victim's device to login

Result: Tampering detected and useless!
```

4.5 Defense Against Device Theft

The Attack:

```
Thief steals your laptop
Has your device fingerprint
Tries to login
```

QEPT Defense:

```
Thief has: Device fingerprint ✓
Thief needs:
- Your password ✗
- Your typing pattern ✗

Thief tries random passwords:
- Rate limited to 10 attempts/hour
- After 5 failures: Account locked
- After 10 failures: Device blacklisted

Even with device, still needs password!
```

Additional protection:

- Behavioral biometrics (typing pattern) adds extra layer
- Thief's typing pattern won't match yours → Triggers MFA
- Can remotely revoke stolen device from another device

4.6 How Much Harder is QEPT to Attack?

Let's compare with actual numbers:

Scenario: Weak Password "password123"

Traditional Argon2:

- Attacker with 1000 GPUs
- Can test 10,000 passwords/second
- Time to crack: **3 days**

QEPT:

- Attacker needs: password + device fingerprint + behavior
- Must attack online (rate limited to 10 attempts/hour)
- Even if they have device: **120,000 years** to crack

Scenario: Medium Password "MyDog2023!"

Traditional Argon2:

- Time to crack: **580 years**

QEPT:

- Time to crack: **8.3 billion years**

Key Insight: QEPT makes even weak passwords incredibly hard to crack because of the added device and behavioral factors.

4.7 Entropy Comparison

Entropy = measure of randomness/unpredictability

Traditional password system:

```
Total entropy = password entropy only
- Weak password: 30 bits (1 billion combinations)
- Strong password: 80 bits (1,000,000,000,000,000,000,000,000 combinations)
```

QEPT system:

```
Total entropy = password + device + behavior + nonce
- Weak password: 30 + 160 + 35 + 128 = 353 bits
- Strong password: 80 + 160 + 35 + 128 = 403 bits

Even weak password in QEPT > strong password in traditional system!
```

What this means: QEPT adds so much extra randomness that even terrible passwords become very secure.

---

5. Evaluation and Comparison

5.1 Security Comparison

Let's see how QEPT stacks up against other methods:

Security Feature	bcrypt	Argon2	OPAQUE	WebAuthn	QEPT
Prevents offline DB attacks	Slows down	Slows down	Yes	Yes	Yes
Stops replay attacks	N/A	N/A	Yes	Yes	Yes
Prevents DB tampering	No	No	Partial	Yes	Yes
Works if device stolen	No protection	No protection	No protection	Needs PIN	Needs password
Stops phishing	No	No	Partial	Yes	Partial
Protects weak passwords	No	No	No	N/A	Yes

Winner: QEPT and WebAuthn are tied for most secure, but QEPT doesn't require special hardware.

5.2 Usability Comparison

User Experience - How annoying is it to use?

System	Setup Time	Login Time	Hardware Needed	Multi-Device Easy?
Password only	30 sec	3 sec	None	Very easy
Password + SMS	1 min	15 sec	Phone	Moderate

Password + TOTP	2 min	8 sec	Phone app	Moderate
WebAuthn	1 min	4 sec	Security key ($50)	Difficult
QEPT	**2-3 min**	**6-10 sec**	**None**	**Moderate**

Analysis:

- QEPT is slightly slower than basic passwords
- But much faster than SMS codes
- No extra hardware needed (unlike WebAuthn)
- Multi-device support is reasonable

5.3 Practical Considerations

Advantages of QEPT:

1. **No special hardware required**
    - Works on any device with standard features
    - No need to buy security keys
2. **Stronger security even with weak passwords**
    - Device and behavior add protection
    - Database theft becomes useless
3. **Transparent security**
    - Device fingerprinting happens automatically
    - Behavioral biometrics collected while typing
    - User doesn't need to do extra steps
4. **Granular device control**
    - Can revoke individual devices
    - See all logged-in devices
    - Get alerts for new device logins

Disadvantages of QEPT:

1. **Device dependency**
    - Must use registered devices
    - New device requires registration process
    - Can be inconvenient when traveling
2. **Slightly slower logins**
    - Token generation takes 1-2 seconds
    - Noticeable but acceptable
3. **Behavioral false rejections**
    - If you type very differently (injured hand, drunk)
    - System might reject you

    - Solution: Falls back to additional verification
4. **Complex server implementation**
    - Requires HSM for master key
    - More complicated than simple password hashing
    - Higher initial setup cost

5.4 When to Use QEPT

Good for:

- Banking and financial services
- Healthcare systems (patient data)
- Corporate accounts with sensitive data
- Government services
- Any high-value account

Not necessary for:

- Low-security accounts (forum registrations)
- Throwaway accounts
- Public/shared computers (library, internet cafe)

---

6. Discussion

6.1 Implementation Challenges

Challenge 1: Device Fingerprint Stability

Problem: What if user upgrades OS or changes hardware?

Solution:

- Use multiple features, allow some to change
- Fuzzy matching (accept 80% match)
- Gradual updates to fingerprint over time

Challenge 2: Behavioral Biometric Variance

Problem: People don't type exactly the same every time

Solution:

- Set tolerant thresholds (accept 85% similarity)

- Continuous learning (update template over time)
- Graceful degradation (trigger MFA instead of blocking)

Challenge 3: HSM Deployment Cost

Problem: Hardware Security Modules cost $5,000-$50,000

Solution:

- Cloud HSM services (AWS, Azure) for small companies
- Software-based secure enclaves (Intel SGX)
- Shared HSM for multiple applications

6.2 Privacy Concerns

Concern 1: Device fingerprinting feels invasive

Response:

- Only collect necessary identifiers
- Hash all values (server never sees raw IDs)
- User explicitly consents during registration
- Clear privacy policy explaining what's collected

Concern 2: Behavioral monitoring

Response:

- Only active during authentication
- Not continuous monitoring
- Data stays on device, only features sent
- Can opt-out of behavioral component

6.3 Real-World Deployment

Deployment Strategy:

Phase 1: High-Security Accounts

- Banks, healthcare first
- Users expect more security here
- Worth the slight inconvenience

Phase 2: Enterprise

- Corporate accounts
- IT can help with registration

- Device management already exists

Phase 3: Consumer Services

- Gradually roll out to public
- Optional initially
- Market as "premium security"

Migration Path:

Year 1: Develop and test QEPT system
Year 2: Pilot with 1000 beta users
Year 3: Offer as optional enhanced security
Year 4: Make default for new accounts

Year 5: Migrate all accounts gradually

6.4 Comparison with Future Technologies

Quantum Computing Threat:

Traditional encryption might be broken by quantum computers in 10-20 years. How does QEPT compare?

QEPT Advantages:

- Uses hash functions (quantum-resistant)
- Not based on public-key crypto (vulnerable to quantum)
- Can upgrade KDF to quantum-safe versions
- Architecture naturally supports post-quantum algorithms

Future Integration:

QEPT can work alongside:

- **Passkeys:** Use QEPT for fallback authentication
- **Biometric authentication:** Add face/fingerprint as additional factor
- **Blockchain:** Store audit logs in distributed ledger
- **AI security:** Machine learning for anomaly detection

6.5 Limitations of This Research

This is a conceptual paper:

- No working prototype yet
- Security analysis is theoretical

- Needs real-world testing
- Performance estimates are projections

What's needed next:

1. Build proof-of-concept implementation
2. User studies for usability testing
3. Security audit by professionals
4. Performance benchmarking
5. Large-scale pilot deployment

6.6 Open Questions for Future Research

1. **Optimal behavioral biometric features:** Which typing patterns work best?
2. **Adaptive security:** How to automatically adjust security level based on risk?
3. **Cross-platform compatibility:** How to handle iOS, Android, Windows, Mac differences?
4. **Scalability:** Can this work for billions of users?
5. **Legal and compliance:** Does this meet GDPR, CCPA, and other regulations?

---

7. Conclusion

7.1 Summary

This paper introduced **Quantum-Entropy Password Tokens (QEPT)**, a new approach to password security that addresses fundamental weaknesses in current systems. Instead of storing password hashes that can be attacked offline, QEPT generates ephemeral one-time tokens by combining passwords with device fingerprints and behavioral biometrics.

Key innovations:

1. **Eliminates offline attacks:** Stolen databases are useless without device context
2. **Strengthens weak passwords:** Device and behavioral factors add 300+ bits of entropy
3. **Prevents tampering:** Encrypted key architecture stops insider attacks
4. **Practical usability:** Works without special hardware, transparent to users

7.2 Contributions

This research makes several important contributions:

Theoretical:

- Novel authentication architecture eliminating stored password hashes

- Formal security analysis showing exponential improvement over existing methods
- Framework for multi-factor cryptographic binding

Practical:

- Detailed protocol specifications for implementation
- Analysis of usability tradeoffs
- Migration path for real-world deployment

Future-oriented:

- Foundation for quantum-resistant authentication
- Compatible with emerging passwordless technologies
- Extensible architecture for new security features

7.3 Impact

If widely adopted, QEPT could significantly improve password security worldwide:

For users:

- Even weak passwords become very secure
- Transparent additional security
- Better protection against database breaches

For companies:

- Reduced liability from password breaches
- Better compliance with security regulations
- Lower risk of account takeovers

For society:

- Fewer identity theft incidents
- More trust in online services
- Foundation for more secure digital infrastructure

7.4 Future Work

This conceptual research opens many directions for future investigation:

Immediate next steps:

1. Build prototype implementation
2. Conduct user studies
3. Professional security audit

4. Performance benchmarking

Long-term research:

1. Integration with blockchain for audit trails
2. AI-powered anomaly detection
3. Quantum-safe cryptographic upgrades
4. Standardization efforts (IETF, NIST)

7.5 Final Thoughts

Password security has been a persistent problem for decades. While we've made progress with better hashing algorithms and multi-factor authentication, the fundamental architecture remains vulnerable: stolen password databases can be attacked offline.

QEPT represents a paradigm shift by eliminating stored password secrets entirely, binding authentication to specific devices, and generating ephemeral tokens that expire after use. This makes password attacks thousands of times harder while maintaining reasonable usability.

As we move toward a more digital world with increasing cyber threats, innovations like QEPT will be essential for protecting user accounts and sensitive data. This research provides a conceptual foundation and roadmap for building the next generation of authentication systems.

References

[1] Provos, N., & Mazières, D. (1999). "A Future-Adaptable Password Scheme." *USENIX Annual Technical Conference*.

[2] Percival, C. (2009). "Stronger Key Derivation via Sequential Memory-Hard Functions." *BSDCan*.

[3] Biryukov, A., Dinu, D., & Khovratovich, D. (2016). "Argon2: The Memory-Hard Function for Password Hashing and Other Applications." *Password Hashing Competition*.

[4] Hunt, T. (2020). "Have I Been Pwned: Check if your email has been compromised in a data breach." Available: https://haveibeenpwned.com

[5] Das, A., Bonneau, J., Caesar, M., Borisov, N., & Wang, X. (2014). "The Tangled Web of Password Reuse." *NDSS Symposium*.

[6] Bonneau, J., Herley, C., Van Oorschot, P. C., & Stajano, F. (2012). "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes." *IEEE Symposium on Security and Privacy*.

[7] W3C (2021). "Web Authentication: An API for accessing Public Key Credentials." *W3C Recommendation*.

[8] Bellovin, S. M., & Merritt, M. (1992). "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks." *IEEE Symposium on Security and Privacy*.

[9] Eckersley, P. (2010). "How Unique Is Your Web Browser?" *Privacy Enhancing Technologies Symposium*.

[10] Yampolskiy, R. V., & Govindaraju, V. (2008). "Behavioural Biometrics: A Survey and Classification." *International Journal of Biometrics*.

[11] OAuth Working Group (2012). "The OAuth 2.0 Authorization Framework." *RFC 6749*.

[12] Wu, T. (1998). "The Secure Remote Password Protocol." *NDSS Symposium*.

[13] Jarecki, S., Krawczyk, H., & Xu, J. (2018). "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks." *EUROCRYPT*.

[14] FIDO Alliance (2022). "Multi-Device FIDO Credentials (Passkeys)." *FIDO Alliance Specification*.

[15] Bojinov, H., Bursztein, E., Boyen, X., & Boneh, D. (2010). "Kamouflage: Loss-Resistant Password Management." *ESORICS*.

[16] Killourhy, K. S., & Maxion, R. A. (2009). "Comparing Anomaly-Detection Algorithms for Keystroke Dynamics." *IEEE/IFIP International Conference on Dependable Systems & Networks*.

[17] Zheng, N., Paloski, A., & Wang, H. (2011). "An Efficient User Verification System via Mouse Movements." *ACM Conference on Computer and Communications Security*.

[18] Frank, M., Biedert, R., Ma, E., Martinovic, I., & Song, D. (2013). "Touchalytics: On the Applicability of Touchscreen Input as a Behavioral Biometric for Continuous Authentication." *IEEE Transactions on Information Forensics and Security*.

[19] NIST (2017). "Digital Identity Guidelines: Authentication and Lifecycle Management." *NIST Special Publication 800-63B*.

[20] Grassi, P. A., et al. (2020). "NIST Special Publication 800-63-3: Digital Identity Guidelines." *National Institute of Standards and Technology*.

Appendix A: Token Generation Pseudocode

This section provides detailed pseudocode for implementing QEPT token generation and validation.

A.1 Device Fingerprint Collection

python
```
def collect_device_fingerprint():
    """
    Collects device-specific characteristics and creates a fingerprint.
    Returns: Hexadecimal string representing device fingerprint
    """
    features = []

    # Hardware identifiers
    features.append(get_cpu_id())
    features.append(get_mac_address())
    features.append(get_screen_resolution())

    # Software characteristics
    features.append(get_os_version())
    features.append(get_browser_version())
    features.append(get_timezone())
    features.append(get_language_settings())

    # Combine all features
    combined = '||'.join(features)

    # Hash to create fingerprint
    fingerprint = SHA256(combined)

    return fingerprint
```

A.2 Behavioral Biometric Collection

python
```
def collect_behavioral_features(password_input_events):
    """
    Analyzes keystroke dynamics during password entry.
    Returns: Feature vector representing typing pattern
    """
    features = {
        'hold_times': [],      # How long each key held
        'flight_times': [],    # Time between keystrokes
        'typing_speed': 0,     # Average characters per second
        'variance': 0          # Consistency metric
```

```
}

# Process keystroke events
for i in range(len(password_input_events) - 1):
    current = password_input_events[i]
    next_key = password_input_events[i + 1]

    # Hold time: key down to key up
    hold_time = current.key_up_time - current.key_down_time
    features['hold_times'].append(hold_time)

    # Flight time: key up to next key down
    flight_time = next_key.key_down_time - current.key_up_time
    features['flight_times'].append(flight_time)

# Calculate statistics
total_time = password_input_events[-1].key_up_time - \
            password_input_events[0].key_down_time
features['typing_speed'] = len(password_input_events) / total_time
features['variance'] = calculate_variance(features['flight_times'])

# Normalize and quantize features
normalized_features = normalize_features(features)


return normalized_features
```

A.3 Token Generation

python

```
def generate_auth_token(password, device_fingerprint,
                        behavioral_features, nonce, salt):
    """
    Generates ephemeral authentication token.

    Parameters:
        password: User's password string
        device_fingerprint: Device fingerprint hex string
        behavioral_features: Dictionary of behavioral metrics
        nonce: Server-provided random value
        salt: User-specific salt

    Returns: Authentication token string
    """
    # Get current timestamp
```

```python
    timestamp = get_current_timestamp()

    # Serialize behavioral features
    behavior_str = json.dumps(behavioral_features, sort_keys=True)

    # Combine all inputs
    combined_input = (
        password.encode('utf-8') +
        device_fingerprint.encode('utf-8') +
        behavior_str.encode('utf-8') +
        nonce.encode('utf-8') +
        str(timestamp).encode('utf-8')
    )

    # Apply Argon2id key derivation
    token = argon2id(
        password=combined_input,
        salt=salt,
        time_cost=3,          # Iterations
        memory_cost=65536,  # 64 MB memory
        parallelism=4,       # 4 parallel threads
        hash_len=64          # 512-bit output
    )

    # Encode as base64 for transmission
    token_b64 = base64_encode(token)

    return token_b64, timestamp
```

A.4 Token Validation

python

```python
def validate_auth_token(username, device_id, token,
                        timestamp, nonce):
    """
    Validates authentication token on server.

    Parameters:
        username: User identifier
        device_id: Hashed device fingerprint
        token: Received authentication token
        timestamp: Timestamp from client
        nonce: Nonce that was sent to client
```

```
Returns: (success: bool, message: string)
"""
# Retrieve user's encrypted device key from database
db_entry = database.query(
    "SELECT * FROM users WHERE username=? AND device_id=?",
    username, device_id
)

if not db_entry:
    log_security_event("Unknown device attempt", username)
    return False, "Device not registered"

# Check nonce hasn't been used (replay prevention)
if nonce_cache.exists(nonce):
    log_security_event("Replay attack detected", username)
    return False, "Invalid nonce - possible replay attack"

# Verify timestamp freshness (prevent delayed replay)
current_time = get_current_timestamp()
time_diff = abs(current_time - timestamp)
MAX_CLOCK_SKEW = 60  # 60 seconds tolerance

if time_diff > MAX_CLOCK_SKEW:
    return False, "Timestamp out of acceptable range"

# Decrypt device key using HSM master key
device_key = hsm.decrypt(
    ciphertext=db_entry.encrypted_key,
    key_id="MASTER_KEY_ID"
)

# Recompute expected token
expected_token = recompute_token(
    device_key=device_key,
    nonce=nonce,
    timestamp=timestamp,
    salt=db_entry.salt
)

# Constant-time comparison (prevents timing attacks)
if not constant_time_compare(token, expected_token):
    # Increment failed attempt counter
    database.execute(
        "UPDATE users SET failed_attempts = failed_attempts + 1 "
```

```python
            "WHERE username=?", username
        )

        # Check if account should be locked
        if db_entry.failed_attempts + 1 >= 5:
            lock_account(username)
            send_security_alert(username, "Multiple failed login attempts")

        return False, "Invalid authentication token"

    # Optional: Verify behavioral biometrics
    if db_entry.behavioral_enabled:
        behavior_score = compare_behavioral_features(
            current_behavior=extract_behavior_from_token(token),
            stored_template=db_entry.behavioral_template
        )

        BEHAVIORAL_THRESHOLD = 0.85  # 85% similarity required

        if behavior_score < BEHAVIORAL_THRESHOLD:
            # Don't reject, but trigger step-up authentication
            return True, "Additional verification required"

    # All checks passed - mark nonce as used
    nonce_cache.add(nonce, expiry_seconds=MAX_CLOCK_SKEW * 2)

    # Update last successful login
    database.execute(
        "UPDATE users SET last_login=?, failed_attempts=0 "
        "WHERE username=?", current_time, username
    )

    # Generate session token
    session_token = generate_session_token(username, device_id)

    return True, "Authentication successful"


def constant_time_compare(a, b):
    """
    Compares two strings in constant time to prevent timing attacks.
    """
    if len(a) != len(b):
        return False
```

```
result = 0
for x, y in zip(a, b):
    result |= ord(x) ^ ord(y)

return result == 0
```

Appendix B: Security Calculations

B.1 Entropy Calculation Details

Password Entropy:

For a random password of length L using character set of size N:

```
E_password = log₂(N^L)
```

Examples:

- 8 lowercase letters: $\log_2(26^8) = 37.6$ bits
- 8 mixed case + numbers: $\log_2(62^8) = 47.6$ bits
- 12 mixed + symbols: $\log_2(95^{12}) = 78.8$ bits

Device Fingerprint Entropy:

Assuming independent features:

```
E_device = Σ log₂(possible_values_i)

CPU ID: log₂(2^64) = 64 bits
MAC address: log₂(2^48) = 48 bits
Screen resolution: log₂(1000) ≈ 10 bits
OS version: log₂(100) ≈ 6.6 bits
Browser config: log₂(1000) ≈ 10 bits
Timezone: log₂(24) ≈ 4.6 bits

Total ≈ 143 bits (conservatively 128 bits after hashing)
```

Behavioral Entropy:

Keystroke dynamics research shows:

E_behavioral ≈ 20-40 bits (depending on features used)

For QEPT, we estimate conservatively: 35 bits

Total QEPT Entropy:

E_total = E_password + E_device + E_behavioral + E_nonce

Minimum (weak password):
E_total = 30 + 128 + 35 + 128 = 321 bits

Typical (medium password):
E_total = 50 + 128 + 35 + 128 = 341 bits

Strong (random password):

E_total = 80 + 128 + 35 + 128 = 371 bits

B.2 Attack Time Calculations

Offline Attack (Traditional Hash):

Given:

- Hash function: Argon2id with t=3, m=64MB
- Attack hardware: 1000 GPUs at 10 hash/sec each
- Total: 10,000 hash/sec

Time to test N passwords:

T = N / 10,000 seconds

Common password dictionary (10^8 passwords):
T = 10^8 / 10,000 = 10,000 seconds ≈ 2.8 hours

Full 8-char lowercase space (26^8 = 2×10^11):

T = 2×10^11 / 10,000 = 2×10^7 seconds ≈ 231 days

Online Attack (QEPT):

Given:

- Rate limit: 10 attempts/hour
- Must attack online (no offline validation)

Time to test N passwords:

T = N / 10 hours

Weak password from common dictionary (10^8):
T = 10^8 / 10 = 10^7 hours ≈ 1,142 years

Medium password space (62^10 = 8×10^17):

T = 8×10^17 / 10 = 8×10^16 hours ≈ 9×10^12 years

With Device Compromise:

Even if attacker steals device (gets device fingerprint):

Still needs password + behavior

Reduced rate limit with device match: 100 attempts/hour
Still requires password guessing

Weak password: 10^8 / 100 = 10^6 hours ≈ 114 years

Medium password: 8×10^17 / 100 ≈ 9×10^11 years

B.3 Database Compromise Scenarios

Scenario 1: Traditional System

Attacker obtains:

- 100 million password hashes
- Each hash can be attacked independently

Expected successful cracks:

Assuming 10% weak passwords:
Successful cracks = 10 million users

Time investment:
10 million × 2.8 hours = 28 million hours
With 1000 GPUs in parallel: 28,000 hours ≈ 3.2 years

But distributed attack:

With 100,000 attackers worldwide: 16.8 minutes average per crack

Scenario 2: QEPT System

Attacker obtains:

- 100 million encrypted device keys
- Cannot decrypt without HSM master key

Attack options:

Option 1: Try to crack HSM
- Requires physical access to server
- HSM has tamper-detection and key erasure
- Infeasible for remote attacker

Option 2: Online password guessing
- Must attack each account individually
- Subject to rate limiting
- Triggers security alerts
- Account lockouts after 5 failures
- Average time per account: 114+ years

Option 3: Social engineering
- Phish users for passwords
- Still need their specific device
- Behavioral biometrics may detect

- Limited scalability

Conclusion: Database theft is catastrophic for traditional systems but merely inconvenient for QEPT.

Appendix C: Deployment Considerations

C.1 System Requirements

Client-Side:

- Modern web browser or native app
- JavaScript/native code execution
- Access to device APIs (fingerprinting)
- ~1-2 MB storage for local crypto library

Server-Side:

- Database with encrypted storage

- Hardware Security Module (HSM) or cloud equivalent
- Redis/Memcached for nonce cache
- Rate limiting infrastructure
- Monitoring and alerting system

C.2 Performance Benchmarks

Token Generation (Client):

```
Device fingerprint collection: 50-100ms
Behavioral feature extraction: 10-20ms
Argon2 computation: 500-1000ms

Total: ~1-1.2 seconds
```

Token Validation (Server):

```
Database lookup: 10-20ms
HSM decryption: 50-100ms
Nonce check: 1-5ms
Token recomputation: 500-1000ms
Behavioral comparison: 20-50ms

Total: ~600-1200ms
```

Throughput:

```
Single server capacity: ~500-1000 auth/second
With caching optimizations: ~2000-5000 auth/second

Horizontal scaling: Linear with server count
```

C.3 Cost Analysis

Infrastructure Costs (for 1 million users):

Traditional system:

```
Database: $500/month
Web servers: $1000/month
Monitoring: $200/month

Total: $1,700/month
```

QEPT system:

```
Database: $500/month
```

```
Web servers: $1,500/month (higher compute)
HSM: $2,000/month (cloud HSM service)
Cache servers: $300/month
Monitoring: $400/month
Total: $4,700/month

Additional cost: $3,000/month

Per user: $0.003/month
```

Break-even Analysis:

Cost of single account takeover:

- Average: $1,000-$5,000 per incident
- Regulatory fines: $10,000-$100,000 per breach
- Reputation damage: Incalculable

If QEPT prevents even 1% of account takeovers:

```
Traditional system breaches: 1000 accounts/year
Cost: 1000 × $3,000 = $3,000,000/year

QEPT breaches: <10 accounts/year
Cost: 10 × $3,000 = $30,000/year

Savings: $2,970,000/year
Additional infrastructure: $36,000/year

Net benefit: $2,934,000/year
```

Author's Note

This research paper was developed as part of my undergraduate studies in *ECE* with a focus on cybersecurity. The goal was to explore how we can improve password security by combining multiple cryptographic and biometric concepts into a unified system.

While this paper is theoretical and doesn't include a working implementation, I believe the concepts presented could form the basis for future practical authentication systems. The security analysis shows significant theoretical improvements over current methods, though real-world deployment would require extensive testing and refinement.

I welcome feedback and suggestions for improving this work. If you're interested in collaborating on a prototype implementation or have questions about the concepts presented, please feel free to reach out.