Comparison of Password-Authenticated Key Exchange Schemes on Android

Jörn-Marc Schmidt
IU International University of Applied Sciences
Erfurt, Thüringen, Germany
email: joern-marc.schmidt@iu.org

Alexander Lawall
IU International University of Applied Sciences
Erfurt, Thüringen, Germany
email: alexander.lawall@iu.org

***Abstract*—Password-Authenticated Key Exchange (PAKE) protocols are critical for secure password-based authentication in various applications, including wireless networking, cloud services, secure messaging, and Internet of Things (IoT) ecosystems. This paper presents a systematic performance evaluation of classical and post-quantum PAKE protocols on a mobile platform, using a Google Pixel 7 Pro running Android 16. We implement a representative set of balanced PAKEs as well as augmented PAKEs. All schemes are implemented in Kotlin/Java using the Bouncy Castle cryptographic provider and evaluated using the Android Jetpack Benchmarking suite under controlled conditions. Our analysis reveals that post-quantum schemes, such as One-Way Key Encapsulation Method to PAKE (OCAKE) and an augmented PAKE scheme based on OCAKE, offer competitive or superior computational performance compared to their classical counterparts, while incurring significantly larger message sizes. We further identify mapping functions, cryptographic primitives, and protocol types as key factors influencing execution time. These results highlight the feasibility of deploying post-quantum PAKEs on constrained mobile devices and provide a benchmark for future optimizations. Future work will examine the impact of hardware acceleration and energy efficiency trade-offs for real-world deployment.**

Keywords-PAKE; post-quantum cryptography; Android; password-based authentication; mobile security.

I. INTRODUCTION

Typical credentials for authentication are passwords. They can be remembered and typed in by humans on various input devices. In terms of security, however, they are commonly easier to guess or to brute-force than cryptographic keys. Password-Authenticated Key Exchange (PAKE) schemes address this issue. They are interactive protocols for two or more parties to generate a joint session key based on a shared password. An adversary eavesdropping on the connection cannot discover the password. Active attacks are, in the best case, limited to one possible password guess per protocol run.

Hence, PAKE schemes can improve security in various cases of password use. For example, a PAKE is employed for password-based authentication in Wireless Fidelity (Wi-Fi) networks [1] and the Matter protocol [2]. Different applications, including 1Password [3] and messengers, such as WhatsApp [4] and Facebook Messenger [5] make use of such schemes. Furthermore, Apple relies on PAKE protocols for HomeKit device enrollment [6], iCloud Keychain escrow [7], and in the Car Key pairing process [8].

However, the PAKE schemes that are used in practical applications rely on the discrete logarithm problem, its elliptic-curve variant, or similar problems. As these problems cannot be considered secure in the presence of cryptographically relevant quantum computers, new protocols are required in this regard. Potential solutions are generic ways to transfer primitives, such as Key Encapsulation Mechanisms (KEMs) into secure PAKE protocols [9]. The resulting OCAKE scheme was recently implemented by Alnahawi et al. on SmartMX3 P71D600 smart card [10]. Lyu et al. published a method to transfer such schemes in asymmetric or augmented PAKE schemes to transfer a balanced PAKE scheme, where both parties know the password, in a client-server setting [11]. The authors also applied their method to lattice-based schemes, yielding lattice-based post-quantum-secure protocols.

In general, the design and analysis of PAKE schemes is an ongoing effort. The work of Alnahawi et al. lists 30 balanced schemes and 19 augmented schemes that have been published since 2015 [12]. They come with different security proofs and various levels of analysis by other researchers. Several also provide benchmark figures for their specific implementations.

In this paper, PAKES with known real-world applications are implemented for Android devices and their performance is measured and compared. In particular, we selected Dragonfly as a balanced PAKE scheme for its use in Wi-Fi Protected Access 3 (WPA3), Password-Authenticated Connection Establishment (PACE) as implemented in travel documents, and CPACE as it is used by Facebook Messenger. We implemented the One-Way Key Encapsulation Method to PAKE (OCAKE) scheme, using Module-Lattice Key Encapsulation Mechanism (ML-KEM) and, for comparison, One-Way Key Encapsulation Method (OEKE) to also include post-quantum-secure schemes.

For augmented schemes, the Secure Remote Password (SRP) protocol was chosen due to the possibility of integration with Transport Layer Security (TLS) [13] and its use with 1Password. In addition, SPAKE2+, together with its balanced version SPAKE2, was selected as it is used by Apple Homekit, Apple Car Key, and the Matter protocol. We also applied the transformation of Lyu et al. to OCAKE, to evaluate a post-quantum-secure augmented PAKE scheme.

The remainder of the paper is organized as follows. Section II gives a brief overview of the implemented PAKE schemes and their properties. The details of the test setup are given in Section III, before the results, together with implementation-specific choices, are presented in Section IV. Conclusions are drawn in Section V.

II. PAKE SCHEMES

Details on how a PAKE achieves its objective of agreeing on a strong cryptographic key based on a shared password

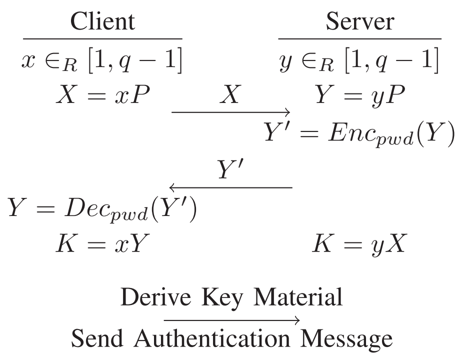

Figure 1. Simplified version of OEKE [17], using a shared password *pwd* and a generator P of order q of an additive group

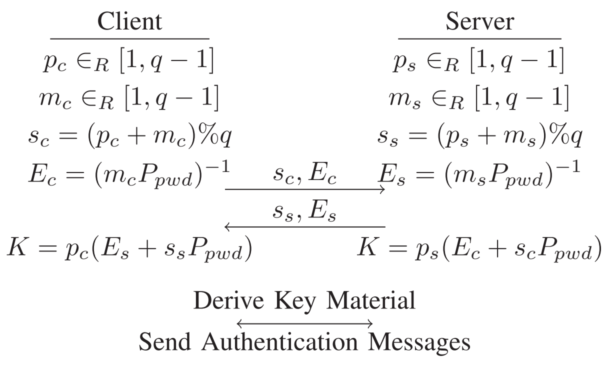

Figure 2. Simplified version of Dragonfly [18], using a shared password element P_{pwd} in an additive group of order q

differ from scheme to scheme. In general, it is possible to distinguish between balanced schemes, where both parties know the password, and augmented schemes, where one party, often called prover, knows the password and the other party, often called verifier, possesses a verification value but not the password itself. This prevents the verifier from impersonating the prover towards another third party. A discussion of the properties of PAKE schemes and related security considerations can be found in [14].

The following notations will be used in the remainder of the paper. Let $Enc_{key}(\cdot)$ and $Dec_{key}(\cdot)$ denote encryption and decryption of a message with a symmetric cipher using a shared key. A hash function is denoted as $H(\cdot)$. The simplified protocols show only the agreement of a shared secret – in order to derive key material, further steps, like applying key derivations, are required. In addition, every protocol requires a verification phase to ensure that both parties followed the protocol and agreed on the same key material. This can be achieved by exchanging hash or message authentication code (MAC) values over protocol data.

In 1992, Bellovin and Merritt published the first PAKE called *Encrypted Key Exchange (EKE)* [15]. It is a balanced scheme. Following different security analysis, including [16], the variant *One-Encryption Key Exchange (OEKE)* was proposed [17]. The underlying idea of those schemes is to derive a secret key from the password and use it to encrypt a public key that is used for key agreement. Hence, the receiver requires the password to decrypt the public key and continue with the protocol. Figure 1 gives a simplified version of the scheme. EKE and OEKE are the inspiration for various post-quantum (PQ) PAKE schemes, including OCAKE [9]. Instead of encrypting a key for a (Elliptic Curve) Diffie-Hellman key agreement method with the password, it uses a KEM and encrypts the related public key with the password.

Another approach is followed by the protocol Dragonfly, defined in RFC 7664 [18]. It maps the password to a group element and uses a random mask to blind it. This blinded value is exchanged and can be used for the next steps towards key agreement. The mapping into the group element requires a specific process. RFC 7664 defines an algorithm called *Hunting and Pecking*, which searches for such an element. In order to ensure a time-constant behavior to prevent side-channel leakage, a constant number of attempts is made. In addition, critical checks are masked using random values. A Dragonfly sample run is shown in Figure 2.

Another PAKE that relies on mapping the password to a group element is the PACE protocol used in travel documents [19]. PACE relies on a random value that is encrypted using the shared password. After mapping the value to the group, a key agreement is performed. For mapping the point to the group, the standard mandates support of at least two mechanisms, a) the *Generic Mapping* based on an (Elliptic Curve (EC)) Diffie-Hellman key agreement, and b) the *Integrated Mapping*, which directly maps a value into the group. A third version, called *Chip Authentication Mapping* is optional.

A similar approach is followed by the Composable Password Authenticated Connection Establishment (CPACE) [20] scheme. Both parties share group parameters and a password. For the mapping, a function specified in RFC 9380 [21] should be used, which corresponds to the algorithm used for the Integrated Mapping. Its first step at both ends is to derive a group element from the password, instead of choosing a random value as input for the mapping as in the PACE protocol. The mapped element is then used for key agreement using a (EC) Diffie-Hellman protocol.

SPAKE2 follows a different approach [22]. It does not require such a mapping function but allows one to create two elements of the used group beforehand. They are defined for the set of parameters used and are independent of the password. The password is mapped to a scalar using a memory-hard hash function (e.g., scrypt or PBKDF2). This scalar is blinded using random values and the pre-defined elements on both ends. The results are exchanged, allowing to agree on a shared element. Its flow is shown in Figure 3.

In applications with a clear client-server relationship, an asymmetric or augmented PAKE can be beneficial. Those schemes provide the server or verifier with the possibility to ensure that the client or prover knows the password, without being able to impersonate the client. This requires a registration

$$\begin{array}{ccc}
\underline{\text{Client}} & & \underline{\text{Server}} \\
x \in_R [1, q-1] & & y \in_R [1, q-1] \\
X = xP & & Y = yP \\
p_A = wM + X & \xrightarrow{p_A} & p_B = wN + Y \\
 & \xleftarrow{p_B} & \\
K = x(p_B - wN) & & K = y(p_A - wM)
\end{array}$$

$$\text{Derive Key Material} \overset{}{\longleftrightarrow} \text{Send Authentication Messages}$$

Figure 3. Simplified version of SPAKE2 [22], using a password element w, a generator P of order q of an additive group G and fixed elements M, N

step where a record that is stored by the verifier is generated.

An example of such an augmented PAKE is SRP. The registration phase of the SRP scheme consists of hashing the password together with a salt value and generating a password verifier in a prime field, using the hashed value as exponent of a group generator. The verifier knows only the verification value; the prover can generate it using the password and the salt. This allows the parties to perform a key agreement based on the password. The flow of the process is shown in Figure 4. Note that SRP relies on multiplying group elements and exponentiating elements in a finite field, which does not map straightforwardly onto elliptic curve groups. Hence, the protocol does not have a simple elliptic curve analogue.

$$\begin{array}{c}
\underline{\text{Registration}} \\
s \in_R [1, 2^{64}] \\
x = H(s|H(U|\text{“:”}|p)) \\
v = g^x \\
\text{Store}(s, v) \\
\underline{\text{Authentication}}
\end{array}$$

$$\begin{array}{ccc}
\underline{\text{Client}} & & \underline{\text{Server}} \\
a \in_R [1, q-1] & & b \in_R [1, q-1] \\
A = g^a & \xrightarrow{A} & B = v + g^b \\
 & \xleftarrow{s, B} & \\
u = H(A, B) & & u = H(A, B) \;\; x = H(s|pwd)) \\
e = (a + u * x) & & \\
K = (B - g^x)^e & & K = (A * v^u)^b
\end{array}$$

$$\text{Derive Key Material} \longleftrightarrow \text{Send Authentication Messages}$$

Figure 4. Simplified version of SRP [23], using a finite field G with a generator g of order q

SPAKE2+[24] is an augmented version of SPAKE2. The authentication flow is similar to SPAKE2. The augmented part is achieved via a registration step, producing a registration record. This record is used in the protocol by the verifier that does not know the password itself.

In order to transform a balanced post-quantum PAKE into an augmented post-quantum PAKE, the transformation of Lyu et al. can be applied [11]. In addition to the balanced PAKE scheme, it uses a KEM and authenticated encryption. During the registration phase, the password is hashed and used to generate a key pair for the KEM scheme, while only the hash and public key are stored. During the authentication phase, the hashed password is used as input to the balanced PAKE to agree on a key. This is followed by an authentication process where the client derives the KEM key pair from the password by following the steps of the registration procedure. Hence, the client now has the private key to the public key contained in the registration record. This enables finalization of the protocol by ensuring that the client knows the password. Using a quantum-secure balanced PAKE and a post-quantum KEM, the protocol provides quantum security.

III. TESTING SETUP

All benchmarks were performed on a Google Pixel 7 Pro smartphone, that is, the physical device, not the emulator. This device features a Google Tensor G2 SoC, 12GB RAM, and runs the Android 16 operating system. Its hardware and up-to-date system software ensure that performance measurements are representative of modern Android platforms. The Pixel 7 Pro device was connected to a Windows test PC via USB with developer options enabled, airplane mode turned on, and all connectivity (Wi-Fi, Bluetooth, mobile data) disabled to reduce interference. The battery saver mode and adaptive battery features were kept off. Using the developer options, the limit for background processes was set to zero. The implementations are written in Kotlin/JAVA and use, in addition to native libraries, a Bouncy Castle provider in version 1.81. Their build target was Android API 34 and ProGuard/R8 minification was enabled.

For performance measurements, microbenchmarking using the Android Jetpack Benchmark Library (version 1.3.4) was used [25]. The library orchestrates test runs, performs warm-up iterations, and leverages the platform's trace-based timing mechanism to reliably capture execution durations. The benchmark process pins the test process to a foreground priority and requests sustained performance mode to reduce CPU/GPU thermal throttling. During this process, 50 measurement runs are conducted. The whole measurement was repeated 100 times, leading to 5000 data points per test. However, it turned out that the measurements contained some outliers, as it was not possible to prevent all side effects during the measurement process. In order to cope with those, measurements that took more than 10 times the mean of the current set are removed. After this procedure, every set contains between 4,974 and 5,000 data points, on average 4,993 data points. The following results are the mean values and the 95% confidence interval of these measurements.

Energy consumption is measured using test functions that cover the whole scheme, that is, client and server operations. For every microbenchmark of a test function, which involves

50 runs of that function, a Perfetto trace [26] is created and analyzed. In particular, the power rail data for the large central processing unit (CPU) core is used. The data points associated with a function run are extracted, yielding an accumulated energy consumption value. Hence, the delta between the first and last measurement run a datapoint is available, gives the number of measurements that are conducted using the energy given by the value difference of those data points. Note that not every power trace contains two values that can be associated with measurement runs. In such cases, the whole run, i.e., this microbenchmark of all PAKE schemes, is discarded. Otherwise, this procedure gives an energy consumption datapoint for every test function, each covering a complete PAKE scheme. In order to prevent thermal effects on the power consumption, the order of the tests within a microbenchmark is randomized. Overall, 250 measurements were conducted, 38 of them were discarded due to missing data points, resulting in 222 energy consumption results per function to be evaluated.

IV. Implementation and Results

In order to allow a fair comparison of the different implementations, all underlying primitives were chosen with parameters for a 128 bit security level. In particular, Advanced Encryption Standard (AES) with 128 bit keys for encryption, Secure Hash Algorithm (SHA) 256 as hash function, a discrete logarithm group with 3072 bits [27], secp256r1 [28] as 256 bit elliptic curve and ML-KEM512 [29]. For modular operations, the native BigInteger library is used. For cryptographic algorithms, native implementations like javax.crypto.Cipher for AES and java.security.MessageDigest for SHA are employed. For all others, including HMAC, HKDR, PBKDF2, scrypt, ML-KEM, and ECC operations, implementations provided by Bouncy Castle are used. In this context, Bouncy Castle uses a Window Non-Adjacent Form (WNAF) multiplier for EC scalar multipication. For the implementations, the client and server components were tested on the same device. All elements that require transfer between the parties were encoded as byte array; compression was used for elliptic curve points.

As described in Section II, several schemes, especially those based on Elliptic Curve Cryptography (ECC), require a mapping from a random string to a point in the group used. Hence, different mapping functions were implemented. In particular, *Hunting and Pecking* as specified in RFC 7664 [18], *Generic Mapping*, and *Integrated Mapping* as specified in [19]. A performance comparison can be found in Table I. Note that the Generic Mapping requires a message exchange between two parties to agree on a common result of the mapping. The figures in the table only reflect the computational effort and not the potential network latency for exchanging those messages. The Hunting and Pecking was configured with a minimum of 40 iterations as suggested in the respective RFC. If the iteration ends as soon as a suitable element is found, the result is obtained on an average of $645 \pm 0.9\mu s$; however, this approach does not ensure constant-time execution. As the integrated mapping computes the result directly–without iterative steps like Hunting and Pecking or key generation and exchange as in Generic Mapping–it is, as expected, the most efficient among the evaluated methods.

TABLE I
Performance of different mapping functions from a random value to secp256r1.

Mapping	Time in μs
Hunting and Pecking	4352 ± 4.1
Generic Mapping	700 ± 0.5
Integrated Mapping	62 ± 0.0

Note that potential overhead for the generic mapping for exchanging messages is not reflected in this number.

In order to compare the performance of balanced PAKEs that provide quantum security and those relying on *traditional* asymmetric primitives, OEKE was implemented and tested using a discrete logarithm group and an elliptic curve. OCAKE was tested using ML-KEM. The implementation of OEKE follows the definition in [17]. Instances using a discrete logarithm group with 3072 bits as defined in RFC 3526[27], and using secp256r1 [28] are evaluated. Both use AES with Cipher Block Chaining (CBC) for the encryption of the public key with the password and SHA256 for computing the authentication tags. The realization of OCAKE is based on the design of Beguinet et al. [9]. A secret key is derived from the shared password using Password-Based Key Derivation Function 2 (PBKDF2) with Hash-based Message Authentication Code (HMAC) SHA256. This key is used to encrypt and transfer a ML-KEM512 public key. Again, SHA256 is used to compute the authentication tags, ensuring that both parties derive the same key material. Using elliptic curves has a notable performance advantage compared to a discrete logarithm group, whereas the OCAKE implementation is even faster. As the client in the OCAKE protocol needs to generate a key pair, it requires more computational effort than the server. In detail, OEKE required $4948 \pm 1.1\mu s$/$4,955 \pm 1.5\mu s$ on the client/server compared to $535 \pm 0.7\mu s$/$507 \pm 0.6\mu s$ when using an elliptic curve. The OCAKE scheme completed in $242 \pm 0.2\mu s$/$151 \pm 0.2\mu s$. However, this speedup comes at the cost of larger messages, as OCAKE requires exchanging the encrypted KEM public key and the ciphertext of 16+816+768 bytes in addition to two 32-byte authentication tags. In contrast, OEKE(ECC)/OEKE exchanges one public key of 33/384 bytes, one encrypted public key of 16+48/16+400 bytes in addition to a 32-byte authentication tag.

Dragonfly was implemented according to RFC [18] using secp256r1. It uses the Hunting and Pecking mapping with at least 40 iterations. For derivation of the secret from the negotiated point, HMAC-based Key Derivation Function (HKDF) SHA256 is used. In order to blind specific checks of the protocol, specific elements need to be found. This can be done once during an initialization phase that took $20 \pm 0.0\mu s$. As client and server perform the same operations, they require the same time. If blinding is omitted and a non-constant-time mapping is used, the agreement process could be reduced from $5,062 \pm 6.8\mu s$ to $1,327 \pm 1.5\mu s$. In terms of message size, Dragonfly exchanges a message containing a scalar and an

TABLE II
OVERVIEW OF USED PARAMETERS FOR IMPLEMENTED SCHEMES

Scheme	Group	Cipher	Hash	Mapping	Others
Balanced Schemes					
OEKE	3072-bit MODP Group	AES-CBC	SHA256	-	-
OEKE (ECC)	secp256r1	AES-CBC	SHA256	-	-
OCAKE	ML-KEM512	AES-CBC	SHA256	-	PBKDF2-HMAC-SHA256
Dragonfly	secp256r1	-	SHA256	Hunting and Pecking	HKDF-SHA256
SPAKE2	secp256r1	-	SHA256	-	HKDF-SHA256
PACE (IM)	secp256r1	AES-CBC	SHA256	Integrated Mapping	PBKDF2-HMAC-SHA256
PACE (GM)	secp256r1	AES-CBC	SHA256	Generic Mapping	PBKDF2-HMAC-SHA256
CPACE	secp256r1	-	SHA256	Integrated Mapping	-
Augmented Schemes					
SRP	3072-bit MODP Group	-	SHA1/SHA256	-	
SPAKE2+	secp256r1	-	SHA256	-	HKDF-SHA256/scrypt
aPAKE-PQC	ML-KEM512	AES-GCM	SHA256	-	-

ECC point from server to client and the other way round, i.e., $2 \times (33 + 33)$ bytes in addition to two authentication tags of 32 bytes each.

The implementation of SPAKE2 follows RFC 9382 [22]. It uses secp256r1 together with the points M,N as defined in the RFC. As hash SHA256 is used. The required computation time is $1,241 \pm 1,6\mu s$/$1,271 \pm 1.7\mu s$ for client/server, that is, around one-fourth of the time required for Dragonfly. SPAKE2 mutually exchanges 33-byte ECC points and 16-byte authentication tags.

PACE [19] was implemented with the Generic Mapping as well as the Integrated Mapping. It uses AES-CBC and PBKDF2-HMAC-SHA256 to derive an AES key from the password. For the confirmation tag of the agreed key material, SHA256 was used. The performance difference from the different mappings, see Table I, translates to the difference in the execution time of PACE. The runtime of the protocol using the Generic Mapping was $1,463 \pm 1.9\mu s$/$1,504 \pm 2.0\mu s$ on the client/server, compared to $797 \pm 1.0\mu s$/$856 \pm 1.1\mu s$ using the Integrated Mapping. In addition, the Generic Mapping requires exchanging the related public keys, which is not required for the Integrated Mapping. Both versions send an encrypted random of size 16+48 bytes from the client to the server and mutually exchange an EC Diffie-Hellmann (ECDH) key of 33 bytes. If the Generic Mapping is used, another ECDH key is mutually exchanged. Note that our implementation exchanges a serialized SubjectPublicKeyInfo object instead of the raw points, which increases the message size from 33 to 335 bytes. For the implementation of CPACE [20], the integrated mapping was used. This scheme performs the same operations on both parties and outputs a hashed transcript and a shared point. Hence, no dedicated verification step is performed in the implementation. Avoiding the exchange of an encrypted value, as is done in the PACE protocol, reduces the computation time to $715 \mp 0.7\mu s$ and the size of the exchanged messages to two times 33 bytes. For the implementation of CPACE [20], the integrated mapping was used. This scheme performs the same operations on both parties and outputs a hashed transcript and a shared point. Hence, no dedicated verification step is performed in the implementation. Avoiding the exchange of an encrypted value, as is done in the PACE protocol, reduces the computation time to $715 \mp 0.7\mu s$ and the size of the exchanged messages to two times 33 bytes.

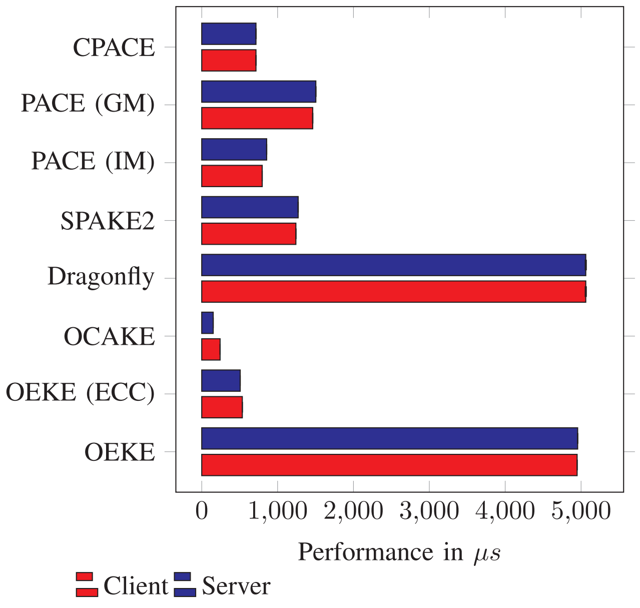

Figure 5. Performance Comparison of the balanced PAKE Schemes

A comparison of the performance of the different balanced schemes is shown in Figure 5. Due to their balanced nature, the computational effort for both parties is comparable. It should be noted that the OCAKE post-quantum scheme that uses KEM is very fast compared to the other schemes at the cost of larger exchanged messages.

In contrast to balanced schemes, where both parties use the password, augmented schemes require a registration phase to construct a verification value that is stored on the server side.

The implementation of the augmented PAKE scheme SRP uses the 3072-bit MODP Group defined in RFC 3526 [27], since a direct translation to elliptic curves is not possible. The measured implementation follows the specification of RFC 2945 [23]. The RFC specifies the use of SHA-1. In order to a) follow the specification and b) allow for a comparable result,

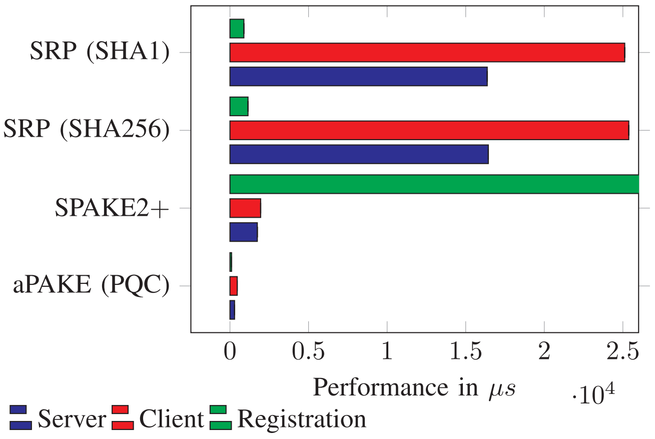

Figure 6. Performance comparison of the augmented PAKE schemes.

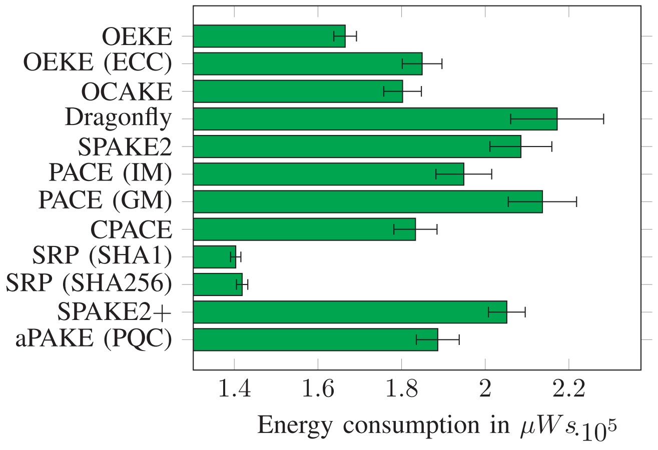

Figure 7. Energy consumption of the implemented schemes on the big CPU power rail with a 95% confidence interval

an instance using SHA1 and an instance using SHA256 were considered. The result shows that the registration process for the SHA256 variant takes about a third longer, $885 \pm 1.3\mu s$ compared to $1,148 \pm 1.4\mu s$, while during the authentication phase, there is only a slight difference, $25,117 \pm 9.3\mu s$/$16,369 \pm 4.4\mu s$ for SHA1 compared to $25,376 \pm 9.1\mu s$/$16,438 \pm 3.8\mu s$ for SHA256 on the client/server. In terms of message sizes, SRP requires the storage of a salt value, in our implementation 9 bytes and a group element of 385 bytes. The parties mutually exchange group 385-byte elements. In addition, the server shares the salt value with the client. In the verification step, every party creates a 20-byte challenge value that is confirmed by the other party with a 20-byte response. When switching from SHA-1 to SHA256, the size of the messages in the verification step increases from 20 to 32 byte.

The implementation of SPAKE2+ makes use of secp256r1, together with SHA256 and an HMAC key derivation function. The registration phase uses the memory-hard hash function scrypt with parameters $(32768, 8, 1)$ as recommended in the RFC. This protects against offline dictionary attacks, but also leads to a significant effort during registration ($115,966 \pm 87.2\mu s$). Hence, Figure 6 does not show the full bar of the registration phase. The scrypt parameters have a direct impact on the performance of this first phase. Since the implementation relies on elliptic curves, it is, with $1,953 \pm 6.1\mu s$/$1,740 \pm 6.5\mu s$ on the client/server, faster than SRP. Note that the client measurement does not include the application of the scrypt function, which could be done during a preparation phase/the registration. Its execution would add an effort comparable to the server registration phase. The value that is stored on the server side for verification consists of a 33-byte compressed ECC point and a 32-byte value that is computed modulo the order of the base point. For the key agreement, ECC points are mutually exchanged, 33 bytes each, while verification uses 32-byte HMAC values on the client and on the server side.

The transformation published by Lyu et al. [11] was applied to the implemented OCAKE to compare augmented PAKE schemes with a quantum-secure scheme. In addition to ML-KEM512, AES-Galois/Counter Mode (GCM) was used for the additional implementations, while the OCAKE protocol still employs AES-CBC. The implemented scheme outperforms the others, requiring $103 \pm 0.1\mu s$ for registration, $461 \pm 0.6\mu s$ on the client, and $292 \pm 0.5\mu s$ on the server. As it employs the balanced PQC PAKE scheme OCAKE, the message sizes are larger compared to the other schemes. The stored record requires a 32-byte hash value and an 800-byte public key. The exchanged messages include those required for OCAKE (16+816+768) plus an encrypted key encapsulation of 16+784 bytes. The verification message uses a 32-byte hash value. It is called *aPAKE (PQC)* in Figure 6, where the performance figures of the different augmented schemes are shown.

The energy consumption of the big CPU powerrail is given in Figure 7. It shows that, when considering a whole run of a scheme, OCAKE and its transformation into an augmented version require the same order of magnitude of energy as the other schemes. An outlier in this regard is SRP, which does not use ECC and hence does not require the Bouncy Castle library, but only native JAVA implementations. The powerrails of the other CPUs, i.e., mid and little, show significantly smaller consumption, but the distribution is comparable, e.g., on the mid CPU, Dragonfly consumes with $12,669 \pm 1107$ μWs the most energy amoung the schemes.

V. Conclusion and Future Work

This study provides an implementation-based comparison of classical and post-quantum PAKE schemes on a modern Android device, highlighting how protocol structure, mapping functions, and cryptographic primitives influence performance. Elliptic curve–based schemes consistently outperform those based on modular arithmetic, while integrated mapping techniques significantly reduce overhead compared to iterative or interactive mappings.

Among the evaluated protocols, post-quantum candidates, such as OCAKE and aPAKE demonstrate strong computational efficiency, achieving sub-millisecond runtimes even without hardware acceleration. Although these schemes incur higher communication overhead, they show that quantum-secure PAKEs are practical for mobile environments. In the augmented setting, the results also emphasize the trade-offs introduced by memory-hard hash functions and the potential for optimization through parameter tuning.

Our current dataset is limited to a single flagship device using the Bouncy Castle library. Generalizing absolute runtimes across the Android ecosystem requires care, because SoCs differ substantially. For example, in CPU microarchitecture, the available instruction set extensions and memory hierarchy. Future work will explore the impact of different SoCs, hardware-accelerated cryptographic instructions, energy profiling under typical usage scenarios, and integration into complete authentication stacks. In addition, a comprehensive security and side-channel resilience evaluation will complement the performance perspective established here.

References

[1] "IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", *IEEE Std 802.11-2024 (Revision of IEEE Std 802.11-2020)*, pp. 1–5956, 2025. DOI: 10.1109/IEEESTD.2025.10979691.

[2] Connectivity Standards Alliance, *Matter Specification - Version 1.4*, Nov. 2004.

[3] R. Fillion, *Secure Remote Password (SRP): How 1Password uses it*, https://blog.1password.com/developers-how-we-use-srp-and-you-can-too/, retrieved: September, 2025.

[4] G. T. Davies *et al.*, "Security analysis of the whatsapp end-to-end encrypted backup protocol", in *Advances in Cryptology – CRYPTO 2023*, H. Handschuh and A. Lysyanskaya, Eds., Cham: Springer Nature Switzerland, 2023, pp. 330–361, ISBN: 978-3-031-38551-3.

[5] Meta, *The labyrinth encrypted message storage protocol*, https://engineering.fb.com/wp-content/uploads/2023/12/TheLabyrinthEncryptedMessageStorageProtocol_12-6-2023.pdf, retrieved: September, 2025.

[6] Apple, *Apple Platform Security - HomeKit communication security*, https://support.apple.com/en-gb/guide/security/sec3a881ccb1/web, retrieved: September, 2025.

[7] Apple, *Apple Platform Security - Escrow security for iCloud Keychain*, https://support.apple.com/en-gb/guide/security/sec3e341e75d/web, retrieved: September, 2025.

[8] Apple, *Apple Platform Security - Car key security in iOS*, https://support.apple.com/en-gb/guide/security/secf64471c16/web, retrieved: September, 2025.

[9] H. Beguinet, C. Chevalier, D. Pointcheval, T. Ricosset, and M. Rossi, "Get a cake: Generic transformations from key encaspulation mechanisms to password authenticated key exchanges", in *Applied Cryptography and Network Security*, M. Tibouchi and X. Wang, Eds., Cham: Springer Nature Switzerland, 2023, pp. 516–538, ISBN: 978-3-031-33491-7.

[10] N. Alnahawi *et al.*, *Post-quantum cryptography in eMRTDs: Evaluating PAKE and PKI for travel documents*, Cryptology ePrint Archive, Paper 2025/812, 2025.

[11] Y. Lyu, S. Liu, and S. Han, "Efficient Asymmetric PAKE Compiler from KEM and AE", in *Advances in Cryptology – ASIACRYPT 2024: 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9–13, 2024, Proceedings. Part V*, Kolkata, India: Springer-Verlag, 2024, pp. 34–65. DOI: 10.1007/978-981-96-0935-2_2.

[12] N. Alnahawi, D. Haas, E. Mauß, and A. Wiesmaier, *SoK: PQC PAKEs - cryptographic primitives, design and security*, Cryptology ePrint Archive, Paper 2025/119, 2025.

[13] D. Taylor, T. Perrin, T. Wu, and N. Mavrogiannopoulos, *Using the Secure Remote Password (SRP) Protocol for TLS Authentication*, RFC 5054, Nov. 2007. DOI: 10.17487/RFC5054.

[14] J.-M. Schmidt, *Requirements for Password-Authenticated Key Agreement (PAKE) Schemes*, RFC 8125, Apr. 2017. DOI: 10.17487/RFC8125.

[15] S. Bellovin and M. Merritt, "Encrypted key exchange: Password-based protocols secure against dictionary attacks", in *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, 1992, pp. 72–84. DOI: 10.1109/RISP.1992.213269.

[16] M. Bellare, D. Pointcheval, and P. Rogaway, "Authenticated key exchange secure against dictionary attacks", in *Advances in Cryptology — EUROCRYPT 2000*, B. Preneel, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 139–155, ISBN: 978-3-540-45539-4.

[17] E. Bresson, O. Chevassut, and D. Pointcheval, "Security proofs for an efficient password-based key exchange", in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, Washington D.C., USA: Association for Computing Machinery, 2003, pp. 241–250, ISBN: 1581137389. DOI: 10.1145/948109.948142.

[18] D. Harkins, *Dragonfly Key Exchange*, RFC 7664, Nov. 2015. DOI: 10.17487/RFC7664.

[19] ICAO - International Civil Aviation Organization, *Doc 9303 - machine readable travel documents- part 11: Security mechanisms for mrtds, eighth edition*, https://www.icao.int/publications/Documents/9303_p11_cons_en.pdf, 2021.

[20] M. Abdalla, B. Haase, and J. Hesse, "CPace, a balanced composable PAKE", Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-cpace-14, Apr. 2025, Work in Progress, 96 pp.

[21] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood, *Hashing to Elliptic Curves*, RFC 9380, Aug. 2023. DOI: 10.17487/RFC9380.

[22] W. Ladd, *SPAKE2, a Password-Authenticated Key Exchange*, RFC 9382, Sep. 2023. DOI: 10.17487/RFC9382.

[23] T. Wu, *The SRP Authentication and Key Exchange System*, RFC 2945, Sep. 2000. DOI: 10.17487/RFC2945.

[24] T. Taubert and C. A. Wood, *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol*, RFC 9383, Sep. 2023. DOI: 10.17487/RFC9383.

[25] Android Developers, *Microbenchmark*, https://developer.android.com/topic/performance/benchmarking/microbenchmark-overview, retrieved: September, 2025.

[26] Perfetto, *System profiling, app tracing and trace analysis*, https://perfetto.dev/, retrieved: September, 2025.

[27] M. Kojo and T. Kivinen, *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*, RFC 3526, May 2003. DOI: 10.17487/RFC3526.

[28] T. Polk, R. Housley, S. Turner, D. R. L. Brown, and K. Yiu, *Elliptic Curve Cryptography Subject Public Key Information*, RFC 5480, Mar. 2009. DOI: 10.17487/RFC5480.

[29] NIST, "Module-lattice-based key-encapsulation mechanism standard", U.S. Department of Commerce, Washington, D.C., Tech. Rep. Federal Information Processing Standards Publication (FIPS) 203, 2024. DOI: 10.6028/NIST.FIPS.203.