Received March 10, 2021, accepted March 22, 2021, date of publication March 30, 2021, date of current version April 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3069627

A Comprehensive and Reproducible Comparison of Cryptographic Primitives Execution on Android Devices

ALEKSANDR OMETOV[1], (Member, IEEE), KRYSTOF ZEMAN[2], PAVEL MASEK[2], (Member, IEEE), LUKAS BALAZEVIC[2], AND MIKHAIL KOMAROV[3], (Senior Member, IEEE)

[1]Tampere University, 33720 Tampere, Finland
[2]Brno University of Technology, 61600 Brno, Czech Republic
[3]National Research University Higher School of Economics, 119049 Moscow, Russia

Corresponding author: Aleksandr Ometov (aleksandr.ometov@tuni.fi)

Project Group was funded by the Graduate School of Business National Research University Higher School of Economics (HSE). This article is prepared with the support of Systems Applications and Products (SAP) University Alliances - the global academic initiative of SAP SE company. The authors would like to thank the SAP CIS market unit for sponsoring the publication of the paper.

ABSTRACT With technology evolving rapidly and proliferating, it is imperative to pay attention to mobile devices' security being currently responsible for various sensitive data processing. This phase is essential as an intermediate before the cloud or distributed ledger storage delivery and should be considered additional care due to its inevitability. This paper analyzes the security mechanisms applied for internal use in the Android OS and the communication between the Android OS and the remote server. Presented work aims to examine these mechanisms and evaluate which cryptographic methods and procedures are most advantageous in terms of energy efficiency derived from execution time. Nonetheless, the dataset with the measurements collected from 17 mobile devices and the code for reproducibility is also provided. After analyzing the collected data, specific cryptographic algorithms are recommended to implement an application that utilizes native cryptographic operations on modern Android devices. In particular, selected algorithms for symmetric encryption are AES256 / GCM / No Padding; for digital signature – SHA512 with RSA2048 / PSS, and for asymmetric encryption – RSA3072 / OAEP with SHA512 and MGF1 Padding.

INDEX TERMS Cryptographic protocols, software measurement, information security, cellular phones, wearable computers.

I. INTRODUCTION

The evolution of the modern Information and Communication Technology (ICT) ecosystem has paved the way for one of the smallest form factor devices, smartphones, and wearables, in various areas of our life [1]. These are computing devices that can function independently and are small enough to be held in the hand [2]. Mobile devices typically connect to the Internet or communicate with other devices in close proximity, forming personal clouds within the new Internet of Wearable Things (IoWT) paradigm [3].

Indeed, the connectivity and flexible interaction are essential since most of those devices process personal data and manage access to it if stored remotely, centrally, or distributed [4], [5]. To comply with current security standards, mobile devices may include biometric sensors and specialized cryptographic primitives [6] and provide a number of primitives for the data processing.

Smartphones and smartwatches are some of the fastest-growing and most widely available personal devices [7]. With the rapid growth in the number of users and hardware/software features, security issues become more pressing [8]. Today, people are more likely to use smartphones for everyday tasks such as browsing, emails, internet banking, and mobile payments but still want to have fast response times as well as the assurance of the security guarantees. Said devices also collect and process their unique vital signs for a variety of purposes. All of these tasks require sensitive user data stored directly on the smartphone (for example, for preprocessing) or uploaded to remote/distributed storage, and the system security levels are shown in Fig. 1.

With a device that comes where the user goes and contains sensitive data, security must go beyond its current level. To ensure a smooth experience for all users (even those who

The associate editor coordinating the review of this manuscript and approving it for publication was Aniello Castiglione.

are not security savvy or do not want to interact directly with it), mobile device manufacturers and app developers must implement security measures that protect user data, even if the smartphone is stolen.

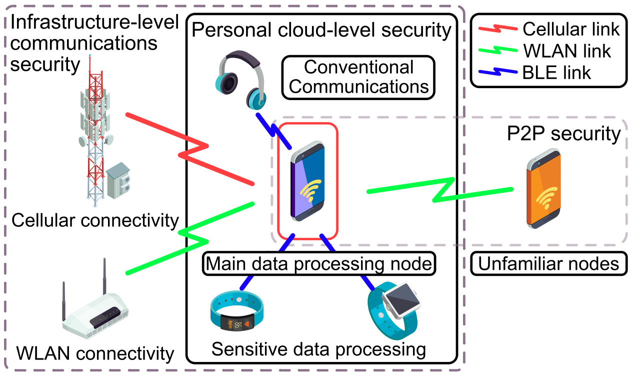

Infrastructure-level communications security
Personal cloud-level security
Conventional Communications
Cellular link
WLAN link
BLE link
P2P security
Cellular connectivity
Main data processing node
Unfamiliar nodes
WLAN connectivity
Sensitive data processing

FIGURE 1. The future wearable-oriented security ecosystem.

As a representative example, this paper focuses on the Android ecosystem as it is easier to develop and evaluate this market segment than other vendors.[1] The Android ecosystem must keep user data safe, and various techniques are integrated it. In particular, the authentication mechanisms are used to provide this kind of guarantee and deny access to unauthenticated users. Android uses the concept of user authentication cryptographic keys, which require a cryptographic key store, a cryptographic key service provider, and a user authenticator [9]. Although the ongoing research is not explicitly targeted at distributed ledger technology applications, the results obtained can also be used to develop appropriate [10] applications. Understanding latency for mobile devices serving as part of a distributed infrastructure with different consensus mechanisms is an integral part of application design, considering the trade-offs between different system characteristics.

This paper is the result of a research project that resulted in the successful defense of the thesis [11]. The presented text is intended to explain the security model on sixteen smartphones and one smartwatch using the Android OS and compare existing cryptographic protocols and the resulting run time as a direct translation to battery life. Unfortunately, providing the actual power consumption of the cryptographic primitives' execution becomes close to impossible due to the present devices' manufacturing concept of irremovable battery, but we also supplement the data with the battery properties that, along with the execution time, allows for better qualitative analysis of the results, similarly to [12], [10].

The main goals of this work are[2]:

- To provide the list of information security primitives available on modern Android devices;
- To give an example (in Kotlin language) of the primitives execution for the ease of other developers attempting to repeat this trial in the future;
- To visualize the comparison of the execution time of the same primitive on different mobile devices;
- To allow other researchers to reuse the collected measurement data by providing a complete dataset (available in IEEE DataPort [13]);
- To facilitate the assumptions on the primitives' executability compared to real-life measurements.

Broadly, we attempt to provide a toolbox and a dataset to be used by the researchers/integrators while selecting an appropriate configuration of one selected system, e.g., Rivest-Shamir-Adleman (RSA), depending on the execution time vs. key size trade-off rather than compare different types of the primitives between each other, which is a well-studied topic with expected outcomes.

The rest of the paper is organized as follows. The next section details what parts the Android security model comprises, their purpose, and how those work with other system components to provide sophisticated security for user data. The third section provides the description of the dataset as well as visualizes and summarizes the results obtained from the created benchmark application. Finally, the last section concludes this work.

II. BACKGROUND INFORMATION

This section outlines the main cryptographic primitives available on modern Android devices and provides sample implementation listings.

A. SUPPORTED CRYPTOGRAPHIC PRIMITIVES

Cryptographic primitives are well-established low-level cryptographic constructs [14]. The Android OS supports various categories of cryptographic primitives recommended by the National Institute of Standards and Technology (NIST) through its *Keystore*,[3] including:

- Hashing functions convert the data of different sizes to data of a fixed size that makes it irreversible, for example, the family of secure hashing algorithms (SHA) (including HMAC-SHA) [15].
- Symmetric key cryptography is an encryption scheme that allows the use of the same key to encrypt and decrypt messages (for example, Advanced Encryption Standard (AES)) [16].
- Asymmetric key cryptography (or sometimes Public Key Cryptography (PKI) is a scheme in which a public key for encryption and a private key for decryption is used for each node, i.e., user A can encrypt a message addressed to user B with PK_B, and only user B can decrypt this message with his own SK_B. Therefore, it becomes possible not only to establish an asymmetric communication environment for securely

[1]This paper is a continuation of the work done by the authors on previous generation smartphones, which could be found at [8].

[2]The main practical contribution in compassion to [11] are extended measurements campaign, open access to the public dataset, and open access to the main source code (both under CC-BY 4.0).

[3]Note, Android OS is designed in such a way so that the development/integration of *new* primitives becomes close to impossible or extremely resource consuming, thus, it is recommended to utilize the integrated primitives assuring the interoperability with other systems.

exchanging symmetric keys in the future by securely exchanging public keys between two users. This group includes, for example, RSA [17], Digital Signature Algorithm (DSA) [18] and various approaches to Elliptic Curve cryptography (EC) [19].

Cryptographic primitives are often used to build cryptographic protocols. On Android, the *Keystore* system uses cryptographic primitives to provide multifunctional cryptographic operations, which include but are not limited to:

- Key generation;
- Import and export of asymmetric keys;
- Import of raw symmetric keys;
- Asymmetric encryption and decryption with appropriate padding modes;
- Digital signature, and verification;
- Symmetric encryption and decryption in appropriate modes, including an Authenticated encryption (AEAD) mode;
- Generation and verification of symmetric message authentication codes;
- Random number generation.

The key target, padding, access control restrictions, or any other protocol element is defined when generating or importing a key and is permanently bound to the key. The protocol elements associated with the key ensure that the key cannot be used in any other way. Random number generation is not exposed to the public Application Programming Interface (API). It is used internally to generate keys, initialization vectors, random padding, and other security protocol elements that require randomness. *KeyStore* can be used as a provider and with supported algorithms that operate in their respective classes:

- *Cipher* allows for encryption and decryption;
- *KeyGenerator* allows for the generation of secret keys for symmetric algorithms;
- *KeyPairGenerator* allows for the generation of key-pairs for asymmetric algorithms;
- *Signature* provides support for the cryptographic digital signature algorithms;
- *KeyFactory* allows for interoperability of cryptographic keys and providing the key specifications;
- *SecretKeyFactory* has a similar functionality as the one above but operating solely with symmetric keys.

Indeed, the Android security system provides a wide range of next-generation information security systems' support. The following implementation options are presented detailing the supported OS versions for the above primitives.

B. PRIMITIVES IMPLEMENTATION EXAMPLES

The following framework will include all available implementation algorithms in Android *KeyStore*, including snapshots of source code to facilitate testing by other developers.

1) KEY GENERATION

To generate a key, one can use the *KeyGenerator* or *KeyPairGenerator* class. *KeyGenerator* provides the functionality of a symmetric key generator. *KeyPairGenerator* provides the functionality of an asymmetric key generator.

TABLE 1. Supported KeyGenerator algorithms with AndroidKeyStore provider. SHA sizes must be multiple of 8.

Algorithm	Minimum API	Key size	Default key size
AES	23+	128, 192, 256	n/a
HMAC SHA1		8–1024	160
HMAC SHA224			224
HMAC SHA256			256
HMAC SHA384			384
HMAC SHA512			512

a: KeyGenerator

There are two approaches to generate a key using *KeyGenerator*: an algorithm-independent and an algorithm-dependent way. The difference between them is generator initialization. Listing 1 lists all of the *KeyGenerator* initialization methods. The *Init* methods that do not use the *AlgorithmParameterSpec* are independent of the algorithm. The *AlgorithmParameterSpec Init* method is used in situations where a set of parameters for a particular algorithm already exists. In case the user does not utilize any of the available *Init* methods, the provider specified during creation must provide a default initialization.

```
// Algorithm-independent Initialization
fun init (random: SecureRandom )
fun init (keysize: Int)
fun init (keysize: Int, random: SecureRandom)
fun init (params: AlgorithmParameterSpec)
fun init (
    params: AlgorithmParameterSpec,
    random: SecureRandom
)
```

LISTING 1. KeyGenerator init methods supported algorithms of KeyGenerator are listed in Table 1.

b: EXAMPLE OF THE KEY GENERATION WITH KeyGenerator

Listing 2 shows the AES symmetric key generation procedure in Galois / Counter Mode (GCM) [20], which aims to encrypt and decrypt without padding encryption. *AndroidKeyStore* is defined as the *KeyGenerator* provider, so the key is generated in the *Keystore* hardware key store. Other restrictions may apply to the *KeyGenParameterSpec* builder. *SetUserAuthenticationRequired* and *SetUserAuthenticationValidityDurationSeconds* can be applied to the builder to condition the receipt of the key to a time window starting from the last unlocking of the phone, or the user can be prompted to log in directly into the application via *Lockscreen*.

The key can be obtained from *AndroidKeyStore*, as shown in Listing 3. *AndroidKeyStore* is run through a static function in the *Keystore* object. In contrast, *AndroidKeyStore* is passed

```
val keyGenerator = KeyGenerator
    .getInstance ("AES", "AndroidKeyStore")
val  keyGenParameterSpec =
    KeyGenParameterSpec.Builder(
        "AES_KEY_ALIAS",
        KeyProperties.PURPOSE_ENCRYPT
        or
        KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes
            (KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings (
            KeyProperties.ENCRYPTION_PADDING_NONE )
        .setKeySize (256)
        .build ()
keyGenerator.init (keyGenParameterSpec)
val secretKey = keyGenerator.generateKey ()
```

LISTING 2. AES Key generation.

TABLE 2. Supported KeyPairGenerator algorithms with AndroidKeyStore provider.

Algorithm	Minimum API	Additional information
DSA	19-22	n/a
EC	23+	Supported sizes: 224, 256, 384, 521; Supported named curves [21]: secp224r1, secp521r1, secp256r1, prime256v1, secp384r1
RSA	18+	Supported sizes: 512, 768, 1024, 2048, 3072, 4096; Supported public exponents: 3, 65537 (default)

as a type after the method *load()* is called for the *KeyStore* object being initialized. The *load()* method loads the *Keystore* using the given *LoadStoreParameter*, which can be *null*. After *KeyStore* is initialized, the key can be obtained by calling the *getKey()* method. The key alias passed to *getKey()* must match the alias during key creation.

```
val keystore = KeyStore.getInstance ("AndroidKeyStore")
                        .apply{load(null)}
val secretKey = keystore.getKey("AES_KEY_ALIAS", null)
```

LISTING 3. Retrieve AES key from keystore.

c: KeyPairGenerator

As with *KeyGenerator*, there are two ways to create a key pair: algorithm-independent or algorithm-dependent methods. The difference between the two is explained in subsection II-B1 and the supported algorithms are provided in Table 2.

d: EXAMPLE OF KEY PAIR GENERATION WITH KeyPairGenerator

Listing 4 depicts the procedure of an EC key pair generation for encryption and decryption. An authenticated user can only use the key within 5 minutes from the last successful authentication. *AndroidKeyStore* is defined as the *KeyPairGenerator* provider, so the key pair is generated in the hardware key storage *Keystore*.

2) IMPORT AND EXPORT OF ASYMMETRIC KEYS

Keystore supports importing PKCS8 standard Distinguished Encoding Rules (DER) [22] key pairs without

```
val kpg = KeyPairGenerator.getInstance (
    KeyProperties.KEY_ALGORITHM_EC,
    "AndroidKeyStore")
val parameterSpec: KeyGenParameterSpec =
    KeyGenParameterSpec.Builder("EC_KEY",
    KeyProperties.PURPOSE_ENCRYPT
    or
    KeyProperties.PURPOSE_DECRYPT)
    .setUserAuthenticationRequired (true)
    .setUserAuthenticationValidityDurationSeconds (300)
    .build ()
kpg.initialize (parameterSpec)
val kp = kpg.generateKeyPair ()
```

LISTING 4. EC Key pair generation.

password-based encryption. According to the X.509 [23], export is only supported for public keys. Two different origin tags are used to distinguish imported keys from reliably generated keys. Here, imported keys use the imported tag, and secure keys use the generated tag.

a: EXAMPLE OF RSA PRIVATE KEY IMPORT

In order to import the private key into *KeyStore*, the *PrivateKey* instance and X.509 certificate for the public key corresponding to the private key represented as the *X509Certificate* are needed. It is mainly because the *KeyStore* abstraction does not support storing private keys without a certificate. Listing 5 shows how to generate a DER-formatted RSA private key and an X.509 certificate or public key.

```
openssl genrsa -out private_key .pem 2048
openssl pkcs8 -topk8 -inform PEM -outform DER
-in private_key.pem -out private_key.der -nocrypt
openssl req -new -x509 -key private_key .pem
-out publickey.cer -days 365
```

LISTING 5. RSA key and certificate generation.

For demonstration purposes, the key and certificate files directly import raw application resources. Listing 6 shows the way how to convert a DER-encoded private key with an X.509 public-key certificate in the *PrivateKey* and *Certificate* instances that are used to import the private key into the *KeyStore*. Although it is possible to import externally generated keys into *Keystore*, it is not recommended. The private key accesses the main memory and can, therefore, be exploited by an attacker.

b: EXAMPLE OF PUBLIC KEY EXPORT

Exporting a public key is a straightforward procedure. Listing 7 shows how to obtain the private record the corresponding certificate from *Keystore*. The certificate can be converted to a *byte array* or *base64* string and sent to the recipient. Note that the private key record contains a private key field that refers to the private key. No sensitive information that could lead to key abuse is present in the private key record.

3) ENCRYPTION AND DECRYPTION USING AN ASYMMETRIC KEY

RSA in various modes and padding settings is the only asymmetric algorithm available on Android to encrypt and decrypt

```
val privByteArray =
    resources.openRawResource(R.raw.private_key)
    .readBytes ()
val spec =
    PKCS8EncodedKeySpec(privByteArray)
val kf = KeyFactory.getInstance ("RSA")
val privKey =
kf.generatePrivate (spec) as RSAPrivateKey
val publicInputStream =
resources.openRawResource(R.raw. publickey)
val cert = CertificateFactory
    .getInstance ("X.509")
    .generateCertificate(publicInputStream)
val ks = KeyStore.getInstance ("AndroidKeyStore")
    .apply{load(null)}
ks.setKeyEntry(
   "MyImportedRsaKey", privKey, null,
   arrayOf(cert))
val privateKey = ks.getKey("MyImportedRsaKey", null)
```

LISTING 6. Import of RSA private key.

```
val entry = kS.getEntry("EC_KEY", null)
    as KeyStore.PrivateKeyEntry
val certificate = entry.certificate
    as X509Certificate
val base64cert = Base64.encodeToString(
    certificate.encoded,
    Base64.NO_WRAP)
val base64PubKey = Base64.encodeToString(
    certificate.publicKey.encoded,
    Base64.NO_WRAP)
```

LISTING 7. Export of EC public key.

data securely. As of this writing, no other asymmetric algorithm is supported. Table 3 lists all combinations of encryption and padding modes. In addition, all combinations support all RSA key sizes that *KeyPairGenerator* generates (512, 768, 1024, 2048, 3072, 4096 bits). Listing 8 shows how to generate an RSA key for RSA / ECB / PKCS1 Padding, where ECB stands for encrypting unlinked blocks of text into the next block, the transformation used in *Cipher*.

```
val kpg = KeyPairGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_RSA,
    "AndroidKeyStore")
val parameterSpec: KeyGenParameterSpec =
    KeyGenParameterSpec.Builder(
        "RSA_KEY",
        KeyProperties.PURPOSE_ENCRYPT
        or
        KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_ECB)
    .setEncryptionPaddings(
       KeyProperties.ENCRYPTION_PADDING_RSA_PKCS1)
    .build()
kpg.initialize(parameterSpec)
val kp = kpg.generateKeyPair ()
val kS = KeyStore.getInstance("AndroidKeyStore")
    .apply{load(null)}
val entry = kS.getEntry("RSA_KEY", null)
    as KeyStore.PrivateKeyEntry
```

LISTING 8. Generate RSA Key for encryption and decryption.

a: EXAMPLE OF RSA DATA ENCRYPTION AND DECRYPTION

Listing 9 shows the process to encrypt data using an RSA key in ECB mode with PKCS1 padding. *Cipher* is initialized with an RSA / ECB / PKCS1 padding transformation that matches the RSA key. The encryption mode is set, and the encryption key is the RSA public key. Data input is in a byte array format.

TABLE 3. Supported RSA variants for encryption and decryption.

Algorithm	Minimum API
RSA / ECB / No Padding	18+
RSA / ECB / PKCS1	18+
RSA / ECB / OAEP with SHA-1 and MGF1	23+
RSA / ECB / OAEP with SHA-224 and MGF1	23+
RSA / ECB / OAEP with SHA-256 and MGF1	23+
RSA / ECB / OAEP with SHA-384 and MGF1	23+
RSA / ECB / OAEP with SHA-512 and MGF1	23+
RSA / ECB / OAEP	23+

For simplicity, the *doFinal(1)* method is called to encrypt the data. The result is a byte array of encrypted data.

```
val dataToEncrypt = "Data". toByteArray ()
val encryptedData:ByteArray =
    Cipher.getInstance ("RSA/ECB/PKCS1Padding")
        .run {
            init(
                Cipher.ENCRYPT_MODE,
                entry.certificate.publicKey)
        doFinal(dataToEncrypt)
            }
```

LISTING 9. Encrypt data with RSA.

Evidently, the RSA private key is used to decrypt the data. Listing 10 shows an approach to decryption. The *Cipher* instance is initialized with the same transformation as the encrypted data. Decryption mode is set, and the decryption key is the RSA private key.

```
val data =
    Cipher.getInstance("RSA/ECB/PKCS1Padding")
        .run {
            init (
                Cipher.DECRYPT_MODE,
                entry.privateKey)
                doFinal(encryptedData)
        }.let{String(it)}
```

LISTING 10. Decrypt data with RSA.

4) DIGITAL SIGNATURE AND VERIFICATION OF SIGNATURE

RSA, EC, DSA can be used in different modes and padding settings. Table 4 shows all the different configurations that can be used for signing and verification. Listing 11 shows how to generate an elliptic curve key pair with SHA512 digest for signing and verification.

a: EXAMPLE OF ECDSA DATA SIGNING AND VERIFICATION

Listing 12 shows how to sign data with an EC key using the SHA512. The signature is initialized with the SHA512 with ECDSA transformation. The signing key is the EC private key. The data to be signed is passed to the *update(1)* method, and the *sign()* method is called to sign the data. The result is a byte array.

The EC public key certificate is used to verify the signature and Listing 13 shows the corresponding procedure. The *Signature* instance is initialized with the same transformation as for data signing. The *verify(1)* method is called to verify the

TABLE 4. Supported algorithms for signing and verification.

Algorithm	Minimal API
MD5 with RSA	18+
NONE with ECDSA	23+
NONE with RSA	18+
SHA1 with DSA	19-22
SHA1 with ECDSA	19+
SHA1 with RSA	18+
SHA1 with RSA/PSS	23+
SHA224 with DSA	20-22
SHA224 with ECDSA	20+
SHA224 with RSA	20+
SHA224 with RSA/PSS	23+
SHA256 with DSA	19-22
SHA256 with ECDSA	19+
SHA256 with RSA	18+
SHA256 with RSA/PSS	23+
SHA384 with DSA	19-22
SHA384 with ECDSA	19+
SHA384 with RSA	18+
SHA384 with RSA/PSS	23+
SHA512 with DSA	19-22
SHA512 with ECDSA	19+
SHA512 with RSA	18+
SHA512 with RSA/PSS	23+

```
val kpg = KeyPairGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_EC,
    "AndroidKeyStore")
val parameterSpec:KeyGenParameterSpec =
    KeyGenParameterSpec.Builder(
        "EC_KEY",
        KeyProperties.PURPOSE_SIGN
        or
        KeyProperties.PURPOSE_VERIFY)
        setDigests(KeyProperties.DIGEST_SHA512)
    .build()
kpg.initialize(parameterSpec)
val kp = kpg.generateKeyPair()
val kS = KeyStore.getInstance("AndroidKeyStore")
    .apply{load(null)}
val entry = kS.getEntry("EC_KEY", null)
    as KeyStore.PrivateKeyEntry
```

LISTING 11. Generate EC Key for sign and verify.

```
val dataToSign = "data".toByteArray ()
val signature: ByteArray =
    Signature.getInstance ("SHA512withECDSA")
        .run {
            initSign(entry.privateKey)
            update(dataToSign)
            sign()
        }
```

LISTING 12. Sign data with ECDSA.

signature passed to the method. The data is then passed to the *update(1)* method. The result is a Boolean value indicating whether the signature is valid.

5) IMPORT OF 'RAW' SYMMETRIC KEYS

Importing symmetric keys is much easier than importing asymmetric keys. The symmetric key is placed in *SecretKeyEntry* and imported directly into *KeyStore*.

```
val valid: Boolean =
    Signature.getInstance ("SHA512withECDSA")
        .run {
            initVerify(entry.certificate)
            update(dataToSign)
            verify(signature)
        }
```

LISTING 13. Verify data with ECDSA.

Listing 14 shows an example of the AES key import with additional key properties defined.

```
fun importAESKey(byteArr: ByteArray) {
    val spec = SecretKeySpec(
        byteArr, 0, byteArr.size, "AES")
    val kS =
        KeyStore.getInstance("AndroidKeyStore")
        .apply { load(null) }
    kS.setEntry (
        "Imported_AES",
        KeyStore.SecretKeyEntry(spec),
        KeyProtection.Builder (
            KeyProperties.PURPOSE_ENCRYPT
            or
            KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes (KeyProperties.BLOCK_MODE_GCM)
            .setEncryptionPaddings (
                KeyProperties.ENCRYPTION_PADDING_NONE)
            .build())
}
```

LISTING 14. Import of AES key.

6) ENCRYPTION AND DECRYPTION USING A SYMMETRIC KEY

AES in various modes and padding settings is the only symmetric algorithm used in Android, no other symmetric algorithm is supported. Table 3 lists all combinations of encryption and padding modes. In addition, all combinations support all AES key sizes that *KeyGenerator* generates (128, 192, 256 bits). Listing 15 shows how to encrypt data using an AES key in GCM mode without padding.

The *Cipher* is initialized with the AES / GRCM / No Padding transformation, which corresponds to AES's key purpose. Data input is in a byte array format. The *doFinal(1)* method is called to encrypt the data. The result is a byte array of encrypted data. To decrypt the cryptogram, the *Cipher* initialization vector must be saved for later use.

```
val dataToEncrypt = "data".toByteArray()
val cipher =
    Cipher.getInstance("AES/GCM/NoPadding")

val encryptedData: ByteArray =
    cipher
        .run{
            init(
                Cipher.ENCRYPT_MODE,
                entry.secretKey)
            doFinal(dataToEncrypt)
        }
 val vector = cipher.iv
```

LISTING 15. Encrypt data with AES.

Listing 16 shows the decryption process. The *Cipher* instance is initialized with the same transformation as the encrypted data. Decryption mode is set, and *GCMParameterSpec* is initialized with an initialization vector and an authorization tag. The *doFinal(1)* method is called with the passed cryptogram as a parameter. The result is an array of bytes that can be converted to string format.

```
val spec = GCMParameterSpec(128, vector)
val data =
    Cipher.getInstance ("AES/GCM/NoPadding")
        .run {
            init(
                Cipher.DECRYPT_MODE,
                entry.secretKey , spec)
            doFinal(encryptedData)
        }.let {String(it)}
```

LISTING 16. Decrypt data with AES.

To summarize, the Android OS already has rich functionality to perform symmetric, asymmetric encryption, block cipher operations, signing, and verification, among others, however, many primitives are still missing, e.g., the post-quantum candidates already highlighted in the NIST final selection round. The examples of the available primitives show that their use is relatively straightforward, and the examples of code can be found in the repository. The next section compares the primitives on flagship and legacy mobile devices and discusses some of the interesting observations made during this benchmarking campaign.

III. CRYPTOGRAPHIC ALGORITHMS COMPARISON

This section uses information from the previous one to execute the tests for the listed cryptographic algorithms. The measurements campaign contained 280 tests each that measure the running time of cryptographic algorithms on 16 smartphones and 1 smartwatch, shown in Fig. 2. We attempted to cover such a broad variety of devices to cover both different hardware characteristics as well as different operating systems, which allows to understand which execution could be expected on a particular generation of devices. This section presents the results of the analysis of the measurements.

A. DATASET DESCRIPTOR

The dataset for the oncoming results is currently available at IEEE DataPort [13]. The primary data related to the collected data is located in folder *Measurement* and subfolder with the measurement file.

The developed application was designed to gather the data and then generate the dataset. The dataset consists of JSON files, each containing measurements of available devices' security primitives execution times. The data was gathered in a span of multiple 250 iterations. Each measurement was taken with a 50 repetitions interval for every primitive. We define the main components of the dataset in the following:

1) *context[]* – provides the details about the device and OS including device name, model, battery-related information, Software Development Kit (SDK) version, and basic technical specification.

FIGURE 2. Mobile devices used for the measurements campaign (technical details are listed in Table 5).

2) *benchmarks[]* – provides entries per primitive, such as:
   - *name* – the overall identification title of the primitive, including paddung and other optional fields;
   - *params* – additional parameters unilized for the execution if any;
   - *totalRunTimeNs* – the overall time of the primitive's execution time;
   - *metrics[]* – provides entries per execution, such as:
     a) *timeNs[]* – the collected/processed information of the collected data inluding entries per execution in *runs[]* and statistical parameters in *maximum*, *minimum* and *median*.
     b) *warmupIterations* – number of iterations of warmup before measurements started;
     c) *repeatIterations* – the number of iterations;
     d) *thermalThrottleSleepSeconds* – the duration of sleep due to thermal throttling.

An example of the dataset entry is provided further in Listing 17.

The following subsections provide a discussion on the obtained results.

B. CREATION OF ASYMMETRIC KEY

Asymmetric key generation testing measurement data shows the time it takes to generate an asymmetric key. The measured key types are RSA and EC with different key size options.

```
{
    "context": {
        "build": {
            "device": "mooneye",
            "fingerprint": "mobvoi/mooneye/mooneye:8.0.0/
                OWDR.180307.020/5000261:user/release-keys
                ",
            "model": "Ticwatch E",
            "version": {
                "sdk": 26
            }
        },
        "cpuCoreCount": 2,
        "cpuLocked": true,
        "cpuMaxFreqHz": -1,
        "batteryCapacity, mAh": 300,
        "memTotalBytes": 514560000,
        "sustainedPerformanceModeEnabled": false
    },
    "benchmarks": [
        {
            "name": "benchmarkRsa4096EcbOaepSHA1AndMgf1
                Padding",
            "params": {},
            "className": "cz.vutbr.benchmark.
                AsymmetricDecryptionBenchmark",
            "totalRunTimeNs": 20873248463,
            "metrics": {
                "timeNs": {
                    "minimum": 242466000,
                    "maximum": 284698307,
                    "median": 245293231,
                    "runs": [
                        284698307,
                        <...>
                        252363077
                    ]
                }
            },
            "warmupIterations": 33,
            "repeatIterations": 1,
            "thermalThrottleSleepSeconds": 0
        },
        <...>
    ]
}
```

LISTING 17. Example dataset entry.

Evidently, as the key size increases, the algorithm's computational complexity also increases. Thus, a larger key size is expected to have a longer execution time than a smaller key size. The heatmap 3 shows the results of the asymmetric key generation. The results of the RSA algorithm support this assumption on all devices.

Using the EC-based algorithm, if we compare the runtimes of EC224 and EC256 on different devices, the results unexpectedly show that a larger key size on six devices results in shorter runtimes, which contradicts the assumption. This could be caused by a small difference between the key size used in the algorithms.

C. ENCRYPTION/DECRYPTION USING AN ASYMMETRIC KEY

Asymmetric key encryption is currently only supported for RSA, as mentioned in subsection II-B3. RSA for encryption can be used in eight different ways. The difference between the two is in the fill mode used. Android *Keystore* and generic java *Keystore* do not implement ECB mode for RSA, so encryption / decryption can only be used for data smaller than the key size. Interestingly, the encryption modes have ECB in their name, although it is not implemented.

The heatmaps 4 and 6 show that the most consistent runtimes across devices are achieved with the RSA with PKCS1 padding option. PKCS1 Padding adds the least overhead of all padding schemes supported (at least 11 bytes). Completing the OAEP adds even more overhead. The OAEP padding scheme requires two hash functions with different properties to work. One hash function must map an arbitrary size input to a fixed size output. Another hash function maps an arbitrary size input to an arbitrary size output.

Such a hash function is called the Mask Generation Function (MFG). OAEP adds at least 42 bytes, which is 31 bytes more than the minimum PKCS1 padding. The results support the assumptions, and the overall execution time for schemes using OAEP padding is longer than for PKCS1 schemes. Based on the measurements, we can conclude that the Samsung Galaxy S6 shows significantly slower encryption with a key size of 4096 bits on all encryption schemes. Huawei P9 Lite and Asus Zenphone 3 MAX, when used with OAEP padding, result in slower performance than other devices. From a security point of view, the OAEP padding scheme is recommended [12]. Comparing the algorithms' results using the OAEP padding scheme shows that RSA3072 / OAEP with SHA-512 and MGF1 padding yields the best result.

As far as decryption is concerned, the inevitable statement is that it should be slower. The advantage of encryption is that the public figure is usually relatively small. The private metric, decryption, is more extensive, so decrypting data is a slower operation. The results in heatmap 5 and 7 summarize the results of the assumption that data decryption in most RSA implementations is slower than encryption. Same algorithm for best overall execution time as encryption, RSA / ECB / PKCS1 Padding.

D. DIGITAL SIGNATURE

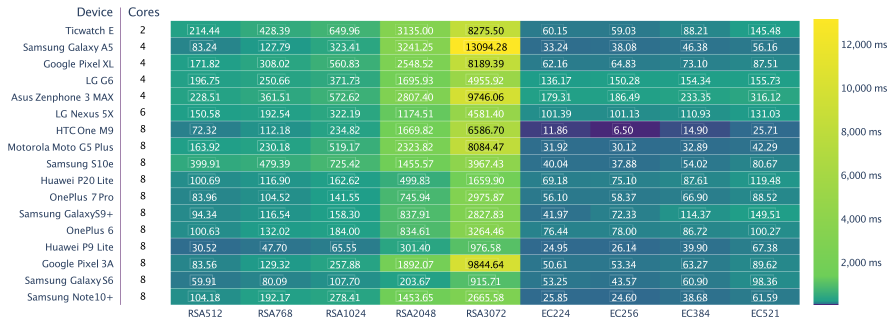
Unlike asymmetric key encryption and decryption, digital signatures are supported by RSA and EC. Benchmarks are categorized by the hash function that RSA or EC uses. The dataset provides RSA results without hashing with MD5, SHA1, SHA224, SHA256, SHA384, and SHA512. All results demonstrate similar behavior for each device, and, in order not to overload the reader with repetitive data, this article provides an example with the most complex of them, SHA512, shown in the heatmap 8.

Overall, measurement results show that the difference in runtime between signature algorithms using different numbers of bits in the SHA function is minimal. Comparing the algorithms' results using the SHA hash function shows that SHA512 with RSA2048 / PSS shows the best results.

A similar procedure was used to verify the digital signature, see example results for SHA512 in heatmap 9. The difference in the execution time of the verification algorithms using the SHA hash function is the same as in the digital signature, i.e., it is minimal. Evidently, the verification procedure on a smartwatch requires more than an order of

TABLE 5. Devices used for benchmarking (sorted according to CPU).

Vendor	Model	Chipset	OS	RAM	Overall cores #	Processor	Cores #	Clocks, GHz
Ticwatch	E Series	MediaTek MT2601	Wear 2.0	0.5	2	Cortex-A7	2	1.2
Samsung	Galaxy A5	Exynos 7880	8	3	4	Cortex-A53	4	1.2
Google	Pixel XL	Qualcomm MSM8996 Snapdragon 821	10	4	4	Kryo	2	2.15
						Kryo	2	1.6
LG	G6	Qualcomm MSM8996 Snapdragon 821	9	4	4	Kryo	2	1.6
						Kryo	2	2.35
Asus	Zenfone 3 max	Mediatek MT6737M	7	2	4	Cortex-A53	4	1.25
LG	Nexus 5X	Qualcomm MSM8992 Snapdragon 808	8.1	2	6	Cortex-A53	4	1.4
						Cortex-A57	2	1.8
HTC	One M9	Qualcomm MSM8994 Snapdragon 810	7	3	8	Cortex-A53	4	1.5
						Cortex-A57	4	2.0
Motorola	Moto G5 Plus	Qualcomm MSM8953 Snapdragon 625	8.1	4	8	Cortex-A53	8	2.0
Samsung	S10e	Exynos 9820	9	8	8	Mongoose M4	2	2.73
						Cortex-A75	2	2.31
						Cortex-A55	4	1.95
Huawei	P20 Lite	HiSilicon Kirin 659	9	4	8	Cortex-A53	4	1.7
						Cortex-A53	4	2.36
OnePlus	7 Pro	Qualcomm SDM855 Snapdragon 855+	10	8	8	Kryo 485	4	1.78
						Kryo 485	3	2.42
						Kryo 485	1	2.96
Samsung	Galaxy S9+	Exynos 9810	9	6	8	Mongoose M3	4	2.7
						Cortex-A55	4	1.8
OnePlus	6	Qualcomm SDM845 Snapdragon 845	10	8	8	Kryo 385 Silver	4	1.7
						Kryo 385 Gold	4	2.8
Huawei	P9 Lite	HiSilicon Kirin 650	7	3	8	Cortex-A53	4	1.7
						Cortex-A53	4	2.0
Google	Pixel 3a	Qualcomm SDM670 Snapdragon 670	10	4	8	Kryo 360 Gold	2	2.0
						Kryo 360 Silver	6	1.7
Samsung	Galaxy S6	Exynos 7420 Octa	7	3	8	Cortex-A53	4	1.5
						Cortex-A57	4	2.1
Samsung	Note10+	Exynos 9825	9	12	8	Cortex-A55	4	1.9
						Cortex-A75	2	2.4
						Mongoose M4	2	2.73



Device	Cores	RSA512	RSA768	RSA1024	RSA2048	RSA3072	EC224	EC256	EC384	EC521
Ticwatch E	2	214.44	428.39	649.96	3135.00	8275.50	60.15	59.03	88.21	145.48
Samsung Galaxy A5	4	83.24	127.79	323.41	3241.25	13094.28	33.24	38.08	46.38	56.16
Google Pixel XL	4	171.82	308.02	560.83	2548.52	8189.39	62.16	64.83	73.10	87.51
LG G6	4	196.75	250.66	371.73	1695.93	4955.92	136.17	150.28	154.34	155.73
Asus Zenphone 3 MAX	4	228.51	361.51	572.62	2807.40	9746.06	179.31	186.49	233.35	316.12
LG Nexus 5X	6	150.58	192.54	322.19	1174.51	4581.40	101.39	101.13	110.93	131.03
HTC One M9	8	72.32	112.18	234.82	1669.82	6586.70	11.86	6.50	14.90	25.71
Motorola Moto G5 Plus	8	163.92	230.18	519.17	2323.82	8084.47	31.92	30.12	32.89	42.29
Samsung S10e	8	399.91	479.39	725.42	1455.57	3967.43	40.04	37.88	54.02	80.67
Huawei P20 Lite	8	100.69	116.90	162.62	499.83	1659.90	69.18	75.10	87.61	119.48
OnePlus 7 Pro	8	83.96	104.52	141.55	745.94	2975.87	56.10	58.37	66.90	88.52
Samsung GalaxyS9+	8	94.34	116.54	158.30	837.91	2827.83	41.97	72.33	114.37	149.51
OnePlus 6	8	100.63	132.02	184.00	834.61	3264.46	76.44	78.00	86.72	100.27
Huawei P9 Lite	8	30.52	47.70	65.55	301.40	976.58	24.95	26.14	39.90	67.38
Google Pixel 3A	8	83.56	129.32	257.88	1892.07	9844.64	50.61	53.34	63.27	89.62
Samsung Galaxy S6	8	59.91	80.09	107.70	203.67	915.71	53.25	43.57	60.90	98.36
Samsung Note10+	8	104.18	192.17	278.41	1453.65	2665.58	25.85	24.60	38.68	61.59

FIGURE 3. Asymmetric key creation.

magnitude more time than the same procedure on a smartphone when the mouse cursor is hovering over a sufficiently low level, that is, tens of milliseconds.

More heatmaps for the utilization of MD5, SHA1, SHA224, SHA256, and SHA384 with RSA are available along with the dataset.

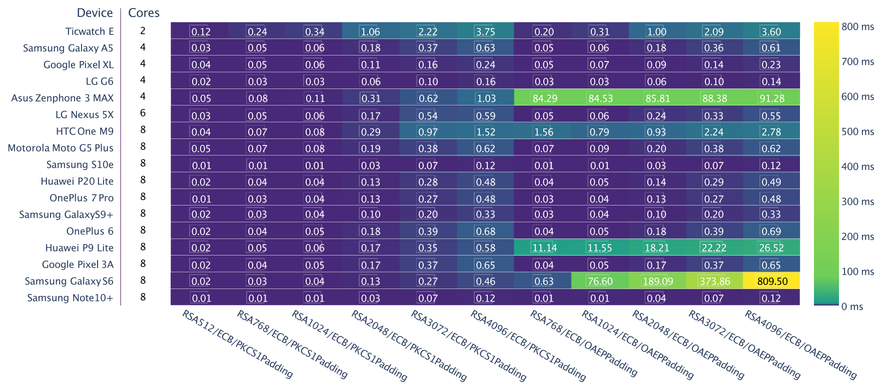


FIGURE 4. Encryption using RSA / ECB with Padding.

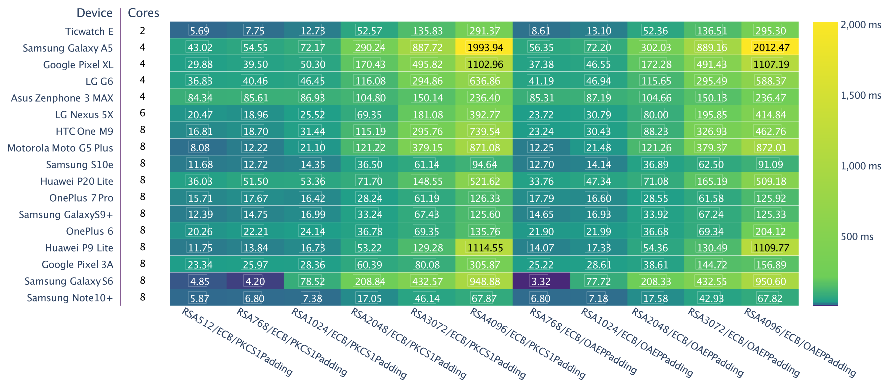


FIGURE 5. Decryption using RSA / ECB with Padding.

E. CREATION OF SYMMETRIC KEY

The symmetric key generation performance test measures the time it takes to generate a symmetric key. The measured key type is AES with different key sizes. As the key size increases, the computational complexity of the algorithm also increases. Thus, a larger key size is expected to have a longer execution time than a smaller key size. The histogram 10 summarizes the results of creating a symmetric key. The difference in battery life on Google Pixel 3A, Huawei P20 Lite, LG Nexus 5X, Google Pixel XL is expected in line with the previous trend. On other devices, the execution time is equal, or the execution time does not increase with increasing key size. It could be caused by hardware optimization or statistical error.

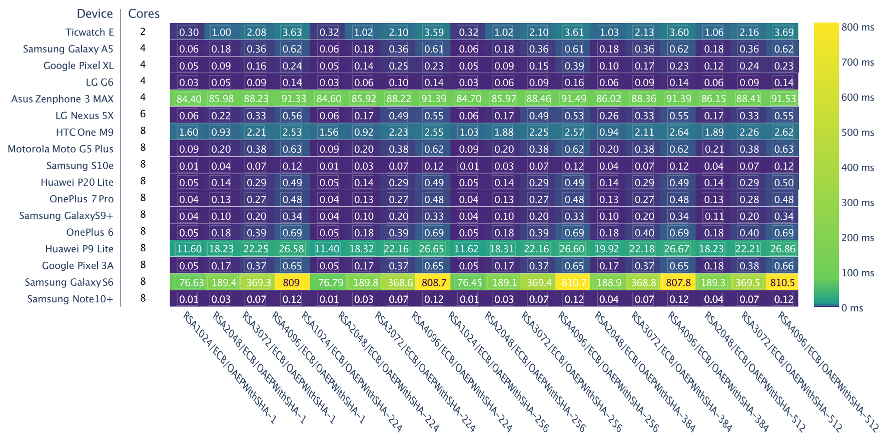
F. ENCRYPTION/DECRYPTION USING AN SYMMETRIC KEY

AES is the only algorithm that supports symmetric encryption with different key sizes and variations. Heatmaps 11 and 12 show that the runtime on the device remains the same for all variants and key sizes. Based on results and in terms of security, AES256 / GCM without padding provides the best value.

It is assumed that decryption and encryption should be approximately the same due to the same keys for both operations. Compared to RSA decryption, AES decryption should be faster. The results in the heatmap 12 show an overall slower execution time than the symmetric encryption execution time, which does not support the assumption of the same execution time. Compared to the RSA decryption execution time, the AES decryption execution time is faster, confirming the assumption.

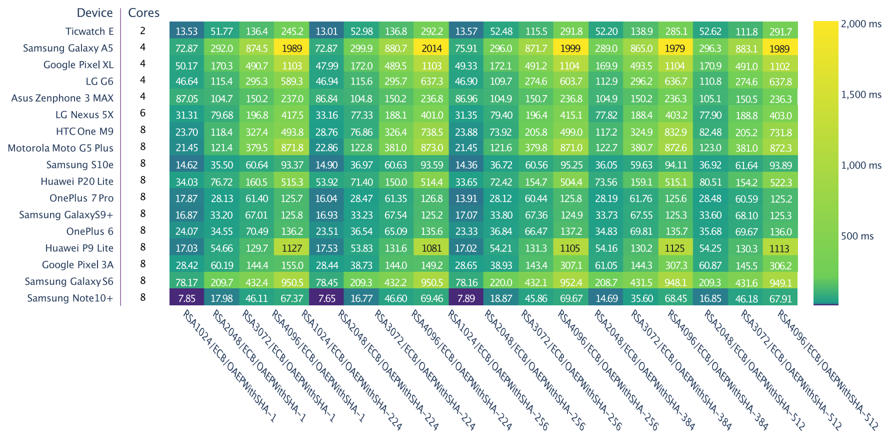
G. ADDITIONAL DISCUSSION AND LESSONS LEARNED

Interestingly, one may question what are the effects of RAM, OS version, or CPU (given in Table 5) on the execution of the primitives. Let us consider Figure 3 as an example. On the one hand, the OS version does not provide any representative information at all, see, e.g., for RSA-512: HTC One M9 shows 72 ms, Huawei P9 Lite shows 30 ms, and Asus Zenphone 3 Max has 228 ms – there is no correlation with OS version. RAM impact has a similar pattern. On the other hand, the execution time is indeed mostly influenced by the CPU characteristics. However, it seems rather unfair to compare the execution of various processor types with different numbers of processors with completely different clock rates since engineers/researchers would not have any real knowledge about the processor's design and would only face the execution time in the end. Moreover,



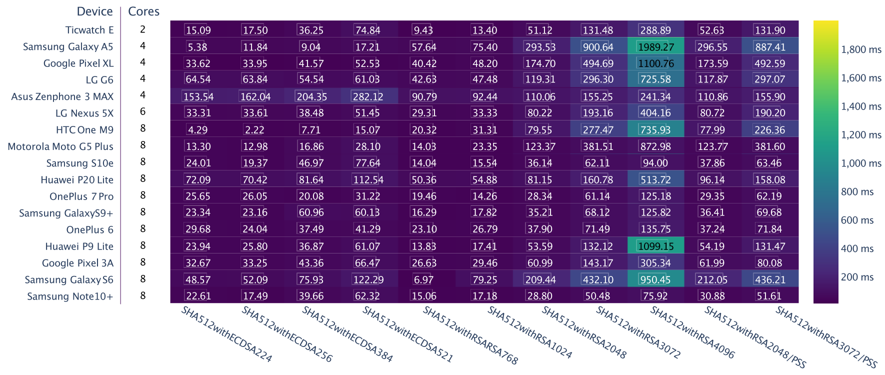
Device	Cores	RSA1024/ECB/OAEPWithSHA-1	RSA2048/ECB/OAEPWithSHA-1	RSA3072/ECB/OAEPWithSHA-1	RSA4096/ECB/OAEPWithSHA-1	RSA1024/ECB/OAEPWithSHA-224	RSA2048/ECB/OAEPWithSHA-224	RSA3072/ECB/OAEPWithSHA-224	RSA4096/ECB/OAEPWithSHA-224	RSA1024/ECB/OAEPWithSHA-256	RSA2048/ECB/OAEPWithSHA-256	RSA3072/ECB/OAEPWithSHA-256	RSA4096/ECB/OAEPWithSHA-256	RSA2048/ECB/OAEPWithSHA-384	RSA3072/ECB/OAEPWithSHA-384	RSA4096/ECB/OAEPWithSHA-384	RSA2048/ECB/OAEPWithSHA-512	RSA3072/ECB/OAEPWithSHA-512	RSA4096/ECB/OAEPWithSHA-512
Ticwatch E	2	0.30	1.00	2.08	3.63	0.32	1.02	2.10	3.59	0.32	1.02	2.10	3.61	1.03	2.13	3.60	1.06	2.16	3.69
Samsung Galaxy A5	4	0.06	0.18	0.36	0.62	0.06	0.18	0.36	0.61	0.06	0.18	0.36	0.61	0.18	0.36	0.62	0.18	0.36	0.62
Google Pixel XL	4	0.05	0.09	0.16	0.24	0.05	0.14	0.25	0.23	0.05	0.09	0.15	0.39	0.10	0.17	0.23	0.12	0.24	0.23
LG G6	4	0.03	0.05	0.09	0.14	0.03	0.06	0.10	0.14	0.03	0.06	0.09	0.16	0.06	0.09	0.14	0.06	0.09	0.14
Asus Zenphone 3 MAX	4	84.40	85.98	88.23	91.33	84.60	85.92	88.22	91.39	84.70	85.97	88.46	91.49	86.02	88.36	91.39	86.15	88.41	91.53
LG Nexus 5X	6	0.06	0.22	0.33	0.56	0.06	0.17	0.49	0.55	0.06	0.17	0.49	0.53	0.26	0.33	0.55	0.17	0.33	0.55
HTC One M9	8	1.60	0.93	2.21	2.53	1.56	0.92	2.23	2.55	1.03	1.88	2.25	2.57	0.94	2.11	2.64	1.89	2.26	2.62
Motorola Moto G5 Plus	8	0.09	0.20	0.38	0.63	0.09	0.20	0.38	0.62	0.09	0.20	0.38	0.62	0.20	0.38	0.62	0.21	0.38	0.63
Samsung S10e	8	0.01	0.04	0.07	0.12	0.01	0.03	0.07	0.12	0.01	0.03	0.07	0.12	0.04	0.07	0.12	0.04	0.07	0.12
Huawei P20 Lite	8	0.05	0.14	0.29	0.49	0.05	0.14	0.29	0.49	0.05	0.14	0.29	0.49	0.14	0.29	0.49	0.14	0.29	0.50
OnePlus 7 Pro	8	0.04	0.13	0.27	0.48	0.04	0.13	0.27	0.48	0.04	0.13	0.27	0.48	0.13	0.27	0.48	0.13	0.28	0.48
Samsung GalaxyS9+	8	0.04	0.10	0.20	0.34	0.04	0.10	0.20	0.33	0.04	0.10	0.20	0.33	0.10	0.20	0.34	0.11	0.20	0.34
OnePlus 6	8	0.05	0.18	0.39	0.69	0.05	0.18	0.39	0.69	0.05	0.18	0.39	0.69	0.18	0.40	0.69	0.18	0.40	0.69
Huawei P9 Lite	8	11.60	18.23	22.25	26.58	11.40	18.32	22.16	26.65	11.62	18.31	22.16	26.60	19.92	22.18	26.67	18.23	22.21	26.86
Google Pixel 3A	8	0.05	0.17	0.37	0.65	0.05	0.17	0.37	0.65	0.05	0.17	0.37	0.65	0.17	0.37	0.65	0.18	0.38	0.66
Samsung Galaxy S6	8	76.63	189.4	369.3	809	76.79	189.8	368.6	808.7	76.45	189.1	369.4	810.7	188.9	368.8	807.8	189.3	369.5	810.5
Samsung Note10+	8	0.01	0.03	0.07	0.12	0.01	0.03	0.07	0.12	0.01	0.03	0.07	0.12	0.04	0.07	0.12	0.04	0.07	0.12

FIGURE 6. Encryption using RSA / ECB / OAEP with SHA and MGF1 Padding.



Device	Cores	RSA1024/ECB/OAEPWithSHA-1	RSA2048/ECB/OAEPWithSHA-1	RSA3072/ECB/OAEPWithSHA-1	RSA4096/ECB/OAEPWithSHA-1	RSA1024/ECB/OAEPWithSHA-224	RSA2048/ECB/OAEPWithSHA-224	RSA3072/ECB/OAEPWithSHA-224	RSA4096/ECB/OAEPWithSHA-224	RSA1024/ECB/OAEPWithSHA-256	RSA2048/ECB/OAEPWithSHA-256	RSA3072/ECB/OAEPWithSHA-256	RSA4096/ECB/OAEPWithSHA-256	RSA2048/ECB/OAEPWithSHA-384	RSA3072/ECB/OAEPWithSHA-384	RSA4096/ECB/OAEPWithSHA-384	RSA2048/ECB/OAEPWithSHA-512	RSA3072/ECB/OAEPWithSHA-512	RSA4096/ECB/OAEPWithSHA-512
Ticwatch E	2	13.53	51.77	136.4	245.2	13.01	52.98	136.8	292.2	13.57	52.48	115.5	291.8	52.20	138.9	285.1	52.62	111.8	291.7
Samsung Galaxy A5	4	72.87	292.0	874.5	1989	72.87	299.9	880.7	2014	75.91	296.0	871.7	1999	289.0	865.0	1979	296.3	883.1	1989
Google Pixel XL	4	50.17	170.3	490.7	1103	47.99	172.0	489.5	1103	49.33	172.1	491.2	1104	169.9	493.5	1104	170.9	491.0	1102
LG G6	4	46.64	115.4	295.3	589.3	46.94	115.6	295.7	637.3	46.90	109.7	274.6	603.7	112.9	296.2	636.7	110.8	274.6	637.8
Asus Zenphone 3 MAX	4	87.05	104.7	150.2	237.0	86.84	104.8	150.2	236.8	86.96	104.9	150.7	236.8	104.9	150.2	236.3	105.1	150.5	236.3
LG Nexus 5X	6	31.31	79.68	196.8	417.5	33.16	77.33	188.1	401.0	31.35	79.40	196.4	415.1	77.82	188.4	403.2	77.90	188.8	403.0
HTC One M9	8	23.70	118.4	327.4	493.8	28.76	76.86	326.4	738.5	23.88	73.92	205.8	499.0	117.2	324.9	832.9	82.48	205.2	731.8
Motorola Moto G5 Plus	8	21.45	121.4	379.5	871.8	22.86	122.8	381.0	873.0	21.45	121.6	379.8	871.0	122.7	380.7	872.6	123.0	381.0	872.3
Samsung S10e	8	14.62	35.50	60.64	93.37	14.90	36.97	60.63	93.59	14.36	36.72	60.56	95.25	36.05	59.63	94.11	36.92	61.64	93.89
Huawei P20 Lite	8	34.03	76.72	160.5	515.3	53.92	71.40	150.0	514.4	33.65	72.42	154.7	504.4	73.56	159.1	515.1	80.51	154.2	522.3
OnePlus 7 Pro	8	17.87	28.13	61.40	125.7	16.04	28.47	61.35	126.8	13.91	28.12	60.44	125.8	28.19	61.76	125.6	28.48	60.59	125.2
Samsung GalaxyS9+	8	16.87	33.20	67.01	125.8	16.93	33.23	67.54	125.2	17.07	33.80	67.36	124.9	33.73	67.55	125.3	33.60	68.10	125.3
OnePlus 6	8	24.07	34.55	70.49	136.2	23.51	36.54	65.09	135.6	23.33	36.84	66.47	137.2	34.83	69.81	135.7	35.68	69.67	136.0
Huawei P9 Lite	8	17.03	54.66	129.7	1127	17.53	53.83	131.6	1081	17.02	54.21	131.3	1105	54.16	130.2	1125	54.25	130.3	1113
Google Pixel 3A	8	28.42	60.19	144.4	155.0	28.44	38.73	144.0	149.2	28.65	38.93	143.4	307.1	61.05	144.3	307.3	60.87	145.5	306.2
Samsung Galaxy S6	8	78.17	209.7	432.4	950.5	78.45	209.3	432.2	950.5	78.16	220.0	432.1	952.4	208.7	431.5	948.1	209.3	431.6	949.1
Samsung Note10+	8	7.85	17.98	46.11	67.37	7.65	16.77	46.60	69.46	7.89	18.87	45.86	69.67	14.69	35.60	68.45	16.85	46.18	67.91

FIGURE 7. Decryption using RSA / ECB / OAEP with SHA and MGF1 Padding.

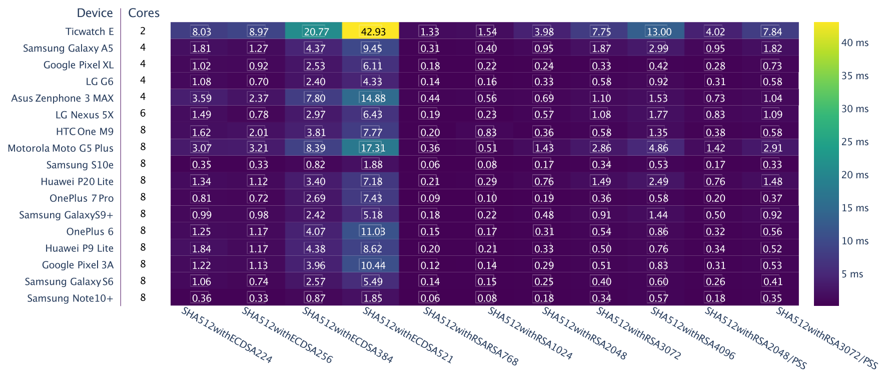


Device	Cores	SHA512withECDSA224	SHA512withECDSA256	SHA512withECDSA384	SHA512withECDSA521	SHA512withRSARSA768	SHA512withRSA1024	SHA512withRSA2048	SHA512withRSA3072	SHA512withRSA4096	SHA512withRSA2048/PSS	SHA512withRSA3072/PSS
Ticwatch E	2	15.09	17.50	36.25	74.84	9.43	13.40	51.12	131.48	288.89	52.63	131.90
Samsung Galaxy A5	4	5.38	11.84	9.04	17.21	57.64	75.40	293.53	900.64	1989.27	296.55	887.41
Google Pixel XL	4	33.62	33.95	41.57	52.53	40.42	48.20	174.70	494.69	1100.76	173.59	492.59
LG G6	4	64.54	63.84	54.54	61.03	42.63	47.48	119.31	296.30	725.58	117.87	297.07
Asus Zenphone 3 MAX	4	153.54	162.04	204.35	282.12	90.79	92.44	110.06	155.25	241.34	110.86	155.90
LG Nexus 5X	6	33.31	33.61	38.48	51.45	29.31	33.33	80.22	193.16	404.16	80.72	190.20
HTC One M9	8	4.29	2.22	7.71	15.07	20.32	31.31	79.55	277.47	735.93	77.99	226.36
Motorola Moto G5 Plus	8	13.30	12.98	16.86	28.10	14.03	23.35	123.37	381.51	872.98	123.77	381.60
Samsung S10e	8	24.01	19.37	46.97	77.64	14.04	15.54	36.14	62.11	94.00	37.86	63.46
Huawei P20 Lite	8	72.09	70.42	81.64	112.54	50.36	54.88	81.15	160.78	513.72	96.14	158.08
OnePlus 7 Pro	8	25.65	26.05	20.08	31.22	19.46	14.26	28.34	61.14	125.18	29.35	62.19
Samsung GalaxyS9+	8	23.34	23.16	60.96	60.13	16.29	17.82	35.21	68.12	125.82	36.41	69.68
OnePlus 6	8	29.68	24.04	37.49	41.29	23.10	26.79	37.90	71.49	135.75	37.24	71.84
Huawei P9 Lite	8	23.94	25.80	36.87	61.07	13.83	17.41	53.59	132.12	1099.15	54.19	131.47
Google Pixel 3A	8	32.67	33.25	43.36	66.47	26.63	29.46	60.99	143.17	305.34	61.99	80.08
Samsung Galaxy S6	8	48.57	52.09	75.93	122.29	6.97	79.25	209.44	432.10	950.45	212.05	436.21
Samsung Note10+	8	22.61	17.49	39.66	62.32	15.06	17.18	28.80	50.48	75.92	30.88	51.61

FIGURE 8. Signature using RSA with SHA512.

most mobile processors cannot utilize all the cores at once (usually 4 for ''background task'' and 4 for ''CPU demanding tasks'').

One of the most significant evaluation challenges is the actual energy consumption evaluation on modern mass-produced devices. There are two significant limitations. First, the only way to reliably measure energy consumption is to connect to the battery connectors physically. We have attempted to do that and faced two sub-challenges: (i) most of the devices have irremovable or shielded batteries that heavily limit the access to the connectors without physically damaging the device; (ii) even after accessing the connectors,



Device	Cores	SHA512withECDSA224	SHA512withECDSA256	SHA512withECDSA384	SHA512withECDSA521	SHA512withRSARSA768	SHA512withRSA1024	SHA512withRSA2048	SHA512withRSA3072	SHA512withRSA4096	SHA512withRSA2048/PSS	SHA512withRSA3072/PSS
Ticwatch E	2	8.03	8.97	20.77	42.93	1.33	1.54	3.98	7.75	13.00	4.02	7.84
Samsung Galaxy A5	4	1.81	1.27	4.37	9.45	0.31	0.40	0.95	1.87	2.99	0.95	1.82
Google Pixel XL	4	1.02	0.92	2.53	6.11	0.18	0.22	0.24	0.33	0.42	0.28	0.73
LG G6	4	1.08	0.70	2.40	4.33	0.14	0.16	0.33	0.58	0.92	0.31	0.58
Asus Zenphone 3 MAX	4	3.59	2.37	7.80	14.88	0.44	0.56	0.69	1.10	1.53	0.73	1.04
LG Nexus 5X	6	1.49	0.78	2.97	6.43	0.19	0.23	0.57	1.08	1.77	0.83	1.09
HTC One M9	8	1.62	2.01	3.81	7.77	0.20	0.83	0.36	0.58	1.35	0.38	0.58
Motorola Moto G5 Plus	8	3.07	3.21	8.39	17.31	0.36	0.51	1.43	2.86	4.86	1.42	2.91
Samsung S10e	8	0.35	0.33	0.82	1.88	0.06	0.08	0.17	0.34	0.53	0.17	0.33
Huawei P20 Lite	8	1.34	1.12	3.40	7.18	0.21	0.29	0.76	1.49	2.49	0.76	1.48
OnePlus 7 Pro	8	0.81	0.72	2.69	7.43	0.09	0.10	0.19	0.36	0.58	0.20	0.37
Samsung GalaxyS9+	8	0.99	0.98	2.42	5.18	0.18	0.22	0.48	0.91	1.44	0.50	0.92
OnePlus 6	8	1.25	1.17	4.07	11.03	0.15	0.17	0.31	0.54	0.86	0.32	0.56
Huawei P9 Lite	8	1.84	1.17	4.38	8.62	0.20	0.21	0.33	0.50	0.76	0.34	0.52
Google Pixel 3A	8	1.22	1.13	3.96	10.44	0.12	0.14	0.29	0.51	0.83	0.31	0.53
Samsung Galaxy S6	8	1.06	0.74	2.57	5.49	0.14	0.15	0.25	0.40	0.60	0.26	0.41
Samsung Note10+	8	0.36	0.33	0.87	1.85	0.06	0.08	0.18	0.34	0.57	0.18	0.35

FIGURE 9. Verification using RSA with SHA512.

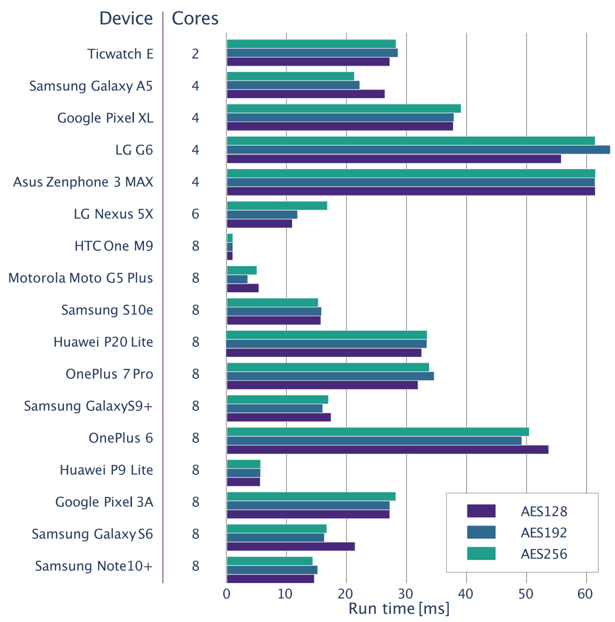


FIGURE 10. Symmetric key creation.

we found out that the impact of the primitives execution on energy consumption is miserable compared to, e.g., wireless transmission module or the display (the primary energy consumer).

Previously, we also attempted to measure the energy consumption for the execution of various blockchain consensus algorithms in [24], where we have proven that the impact on battery is not measurable only if the device is in the saturation of the cryptography-related executions, which is not real in daily life. Therefore, this paper attempted to highlight the metric, which is the most accessible for the developers from a user perspective – the average execution time causing uncomfortable delays while a human is interacting with the smartphone. As it could be seen from the collected data, some devices can produce a few seconds-level delays just for one execution, which may be unacceptable for close-to-real-time applications.

Notably, there are approaches how to approach the conversion of the execution time into relative computational energy.[4] However, those may have relatively low accuracy since even the comparison of self-discharge rate compared to one under additional tasks do not provide the necessary granularity [8].

IV. SUMMARY

The development of modern technologies and the overall improvements in the computing power are pushing towards evaluating cryptographic primitives used on devices available on the market. This article closes this white spot on the roadmap for information security in the field of Android devices. Along with this, it provides source code examples suitable for a future re-evaluation of new devices. Specifically, it summarizes the results from a benchmark app that ran 280 tests on 16 smartphones and 1 smartwatch in terms of execution time (as the most convenient battery life convention).

The results were further processed and visualized on heat maps and a histogram. Based on the results, it was concluded that not all natural assumptions regarding the executions of primitives were fulfilled. Some older devices with older processors execute some cryptographic algorithms faster than newer devices with newer processors. It can be explained by hardware acceleration for specific cryptographic algorithms.

After analyzing the collected data, specific cryptographic algorithms were selected to implement an application using cryptographic operations, see Section III. Selected algorithms: AES256 / GCM / No Padding for symmetric encryption, SHA512 with RSA2048 / PSS for digital signature and RSA3072 / OAEP with SHA512 and MGF1 Padding for asymmetric encryption.

[4]See "Monitoring Energy Hotspots in Software": https://hal.inria.fr/hal-01069142/document

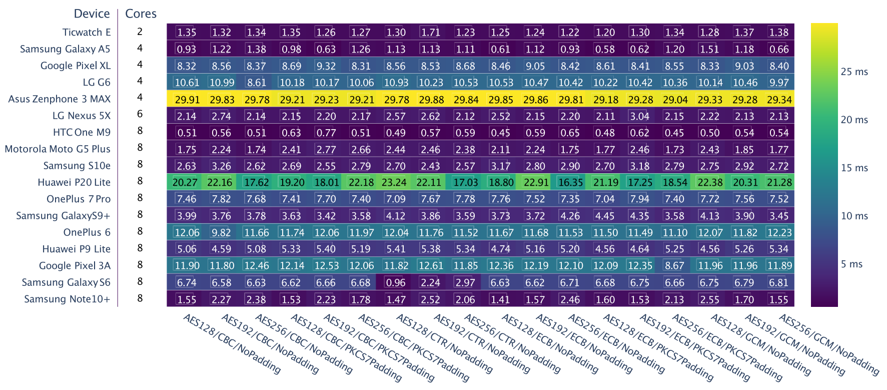


FIGURE 11. AES encryption.

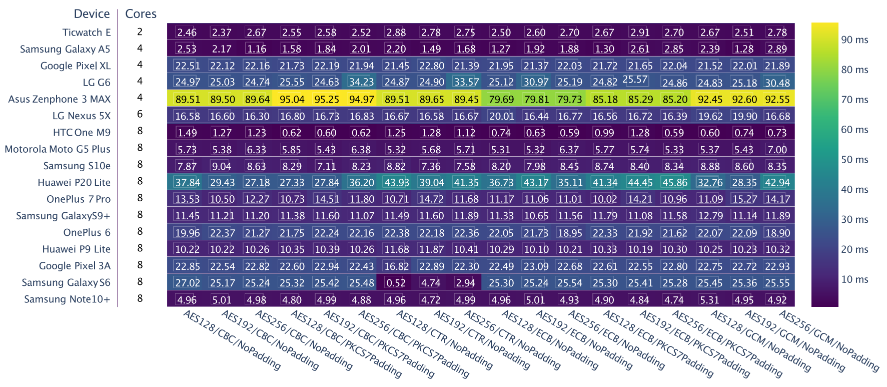


FIGURE 12. AES decryption.

LIST OF ACRONYMS

AEAD	Authenticated Encryption
AES	Advanced Encryption Standard
API	Application Programming Interface
BLE	Bluetooth Low Energy
DER	Distinguished Encoding Rules
DSA	Digital Signature Algorithm
EC	Eliptic Curve
ECB	Encrypting unlinked blocks
ECDSA	Elliptic Curve Digital Signature Algorithm
GCM	Galois / Counter Mode
HMAC	Keyed-Hashing for Message Authentication
ICT	Information and Communication Technology
JSON	JavaScript Object Notation
MD	Message-Digest algorithm
MGF	Mask Generation Function
OAEP	Optimal Asymmetric Encryption Padding
OS	Operating System
PKCS	Public Key Cryptography Standards
PSS	Probabilistic Signature Scheme
RSA	Rivest-Shamir-Adleman System
SDK	Software Development Kit
SHA	Secure Hash Algorithm
WLAN	Wireless Local Network

ACKNOWLEDGMENT

This article was presented as a thesis work by the author Lukas Balazevic in Security Mechanisms of OS Android utilizing the Kotlin Language at Brno University of Technology, 2020.

REFERENCES

[1] P. Sobreiro and A. Oliveira, "The importance of ICT and wearable devices in monitoring the health status of coronary patients," in *Proc. Int. Conf. Human Syst. Eng. Design, Future Trends Appl.* Cham, Switzerland: Springer, 2019, pp. 705–711.

[2] S. K. Kane, "Wearables," in *Web Accessibility*. London, U.K.: Springer, 2019, pp. 701–714.

[3] W. B. Qaim, A. Ometov, A. Molinaro, I. Lener, C. Campolo, E. S. Lohan, and J. Nurmi, "Towards energy efficiency in the Internet of Wearable Things: A systematic review," *IEEE Access*, vol. 8, pp. 175412–175435, 2020.

[4] D. He, S. Li, C. Li, S. Zhu, S. Chan, W. Min, and N. Guizani, "Security analysis of cryptocurrency wallets in Android-based applications," *IEEE Netw.*, vol. 34, no. 6, pp. 114–119, Nov. 2020.

[5] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "An empirical study of Android security bulletins in different vendors," in *Proc. Web Conf.*, 2020, pp. 3063–3069.

[6] K. Sowjanya, M. Dasgupta, and S. Ray, "An elliptic curve cryptography based enhanced anonymous authentication protocol for wearable health monitoring systems," *Int. J. Inf. Secur.*, vol. 19, no. 1, pp. 129–146, Feb. 2020.

[7] V. Cisco. (Feb. 2020). *Cisco Annual Internet Report (2018–2023) White Paper*. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[8] A. Ometov, P. Masek, L. Malina, R. Florea, J. Hosek, S. Andreev, J. Hajny, J. Niutanen, and Y. Koucheryavy, "Feasibility characterization of cryptographic primitives for constrained (wearable) IoT devices," in *Proc. IEEE Int. Conf. Pervas. Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2016, pp. 1–6.
[9] Android Open Source Project. (2020). *Secure an Android Device*. [Online]. Available: https://source.android.com/security/
[10] Y. Bardinova, K. Zhidanov, S. Bezzateev, M. Komarov, and A. Ometov, "Measurements of mobile blockchain execution impact on smartphone battery," *Data*, vol. 5, no. 3, p. 66, Jul. 2020.
[11] L. Balazevic, "Security mechanisms of OS Android utilizing the Kotlin language," M.S. thesis, Brno Univ. Technol., Brno, Czech Republic, 2020.
[12] A. Chatzikonstantinou, C. Ntantogian, G. Karopoulos, and C. Xenakis, "Evaluation of cryptography usage in Android applications," in *Proc. 9th EAI Int. Conf. Bio-inspired Inf. Commun. Technol.*, 2016, pp. 83–90.
[13] A. Ometov, K. Zeman, P. Masek, L. Balazevic, and M. Komarov, "Measurements of cryptographic primitives execution on Android devices," *IEEE Dataport*, 2021, doi: 10.21227/34cr-vs54.
[14] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook Application Cryptography*. Boca Raton, FL, USA: CRC Press, 2018.
[15] M. J. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. FIPS PUB 202, 2015.
[16] *FIPS 197: Advanced Encryption Standard (AES)*, Federal Inf. Process. Standards, 2001, vol. 197, no. 441, p. 0311.
[17] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS# 1: RSA cryptography specifications version 2.2," Internet Eng. Task Force, Request Comments, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Dec. 2016, vol. 8017, p. 78.
[18] C. F. Kerry and P. D. Gallagher, "Digital Signature Standard (DSS)," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. FIPS PUB 186-4, Mar. 2013.
[19] L. Chen, D. Moody, A. Regenscheid, and K. Randall, "Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. SP 800-186 (Draft), 2019.
[20] D. McGrew and J. Viega, "The Galois/Counter Mode of operation (GCM)," in *Proc. NIST*, vol. 20, 2004, pp. 70–78.
[21] S. Turner, D. Brown, K. Yiu, R. Housley, and T. Polk, *Elliptic Curve Cryptography Subject Public Key Information*, document RFC 5480, 2009.
[22] I. Rec, "X.690 information technology–ASN. 1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," ITU, Geneva, Switzerland, Tech. Rep. Recommendation X.690, 2015.
[23] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "RFC 5280: Internet X. 509 public key infrastructure certificate and Certificate Revocation List (CRL) profile," Internet Eng. Task Force, Fremont, CA, USA, Tech. Rep. RFC 5280, 2008.
[24] A. Ometov, Y. Bardinova, A. Afanasyeva, P. Masek, K. Zhidanov, S. Vanurin, M. Sayfullin, V. Shubina, M. Komarov, and S. Bezzateev, "An overview on blockchain for smartphones: State-of-the-art, consensus, implementation, challenges and future trends," *IEEE Access*, vol. 8, pp. 103994–104015, 2020.

ALEKSANDR OMETOV (Member, IEEE) received the Specialist degree in information security from the Saint Petersburg State University of Aerospace Instrumentation (SUAI), Russia, in 2013, and the M.Sc. degree in information technology and the D.Sc. (Tech) degree in telecommunications from the Tampere University of Technology (TUT), Finland, in 2016 and 2018, respectively. He is currently a Postdoctoral Research Fellow with Tampere University (TAU), Finland. He is also working on H2020 MCSA A-WEAR and APROPOS projects. His research interests include wireless communications, information security, computing paradigms, blockchain technology, and wearable applications.

KRYSTOF ZEMAN is currently a Postdoctoral Researcher with the Brno University of Technology (BUT), Czech Republic. He is a Senior Member of the WISLAB Research Group, where he is also focusing on millimeter-wave signal propagation and channel modeling and various aspects of the Internet of Things and Industry 4.0-driven research projects.

PAVEL MASEK (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering from the Faculty of Electrical Engineering and Communication, Brno University of Technology (BUT), Czech Republic, in 2013 and 2017, respectively. He is currently a Researcher with the Department of Telecommunications, BUT. He is also co-supervising the WISLAB Research Group. He has coauthored more than 90 research works on a variety of networking-related topics in internationally recognized venues, including those published in the *IEEE Communications Magazine*, as well as several technology products. His current research interests include heterogeneous wireless communication networks and systems, the Internet of Things, and Industry 4.0-driven research projects.

LUKAS BALAZEVIC received the M.Sc. degree in electrical engineering from the Faculty of Electrical Engineering and Communication, Brno University of Technology (BUT), Czech Republic, in 2020. He is currently a contracting Android Developer with Futured Apps Company.

MIKHAIL KOMAROV (Senior Member, IEEE) is currently a Professor with the Department of Business Informatics, Graduate School of Business, National Research University Higher School of Economics. He is a specialist in wireless data transmission and IT. He is the Vice-Chair of the Special Interest Group on IoT at the Internet Society.