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 BALAZEVIĆ\(^2\), 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)

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...
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.

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, 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 PKB, and only user B can decrypt this message with his own SKB. 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测量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.

**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 (keysizes: Int)
fun init (keysizes: 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.

#### 2) 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

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

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

### Listing 3: Retrieve AES key from keystore.

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

### Table 2: Supported KeyPairGenerator algorithms with AndroidKeyStore provider.

| Algorithm | Minimum API | Additional information |
|-----------|-------------|------------------------|
| DSA       | 19-22       | n/a                    |
| BC        | 23+         | Supported sizes: 224, 256, 384, 512; 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) |

### Listing 4: EC Key pair generation.

```
val kpg = KeyPairGenerator.getInstance()
val parameterSpec = KeyGenParameterSpec.Builder("EC_KEY", KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT)
.param(KeystoreValidationDurationSeconds(300))
.build()
kpg.initialize(parameterSpec)
val kp = kpg.generateKeyPair()
```

### Listing 5: RSA key and certificate generation.

```
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
```

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 Distiguished Encoding Rules (DER) [22] key pairs without 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.

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...
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`.

### Listing 8. Generate RSA Key for encryption and decryption.

```java
val privByteArray = resources.openRawResource(R.raw.private_key)
val readBytes () = 
val spec = PKCS8EncodedKeySpec(privByteArray)
val k1 = KeyFactory.getInstance("RSA")
val privKey = k1.generatePrivate(spec) as RSAKey
val publicInputStream = resources.openRawResource(R.raw.publicKey)
val cert = CertificateFactory.getInstance("X.509")
val generateCertificate(publicInputStream)
val ks = KeyStore.getInstance("AndroidKeyStore")
apply(load(null))
ks.setKeyEntry("MyImportedRSAKey", privKey, null)
val privateKey = ks.getKey("MyImportedRSAKey", null)
```

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

### Listing 9. Encrypt data with RSA.

```java
val entry = ks.getEntry("EC_KEY", null)
val cert = entry.getCertificate()
val base64Cert = Base64.encodeToString(cert.getEncoded(), Base64.NO_WRAP)
val publicKey = Base64.encodeToString(cert.getPublicKey().getEncoded(), Base64.NO_WRAP)
```

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.

### Listing 10. Decrypt data with RSA.

```java
val dataToEncrypt = "Data", toByteArray ()
val encryptedData = Cipher.getInstance("RSA/ECB/PKCS1Padding")
.init{
   Cipher.ENCRYPT_MODE,
   entry.getCertificate().getPublicKey()}
doFinal(dataToEncrypt)
```

### Listing 11. Generate RSA Key for encryption and decryption.

```java
val kp = KeyPairGenerator.getInstance("RSA")
val parameterSpec = KeyGenParameterSpec.BUILDER
   .RSA_KEY
   ._PURPOSE_ENCRYPT
   .build()
val kp.generateKeyPair()
val kS = KeyStore.getInstance("AndroidKeyStore")
apply(load(null))
ks.setKeyEntry("RSA_KEY", null)
```

### Listing 12. Encrypt data with RSA.

```java
val data = Cipher.getInstance("RSA/ECB/PKCS1Padding")
.run{
   Cipher.ENCRYPT_MODE,
   entry.getPrivateKey()}
doFinal(encryptedData)
```

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.

### Listing 13. Sign data with RSA.

```java
val entry = ks.getEntry("EC_KEY", null)
```

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()` 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+ |

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.

LISTING 11. Generate EC Key for sign and verify.

```scala
val kpg = KeyPairGenerator.getInstance("EC", "AndroidKeyStore")
val parameterSpec = KeyGenParameterSpec.Builder("
    "EC_KEY",
    KeyProperties.PURPOSE_SIGN
    or
    KeyProperties.PURPOSE_VERIFY
    .setDigests(KeyProperties.DIGEST_SHA512)
    .build()
)
kpg.initialize(parameterSpec)
val kS = KeyStore.getInstance("AndroidKeyStore")
    .load(null)
val entry = kS.getEntry("EC_KEY", null)
as KeyStore.PrivateKeyEntry
```

LISTING 12. Sign data with ECDSA.

```scala
val signature = ByteString.getData("data").toByteArray()
val signatureInstance = Signature.getInstance("SHA512withECDSA")
    .run(
        initSign(entry, privateKey)
        update(dataToSign)
        sign()
    )
```

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 / GCM / NoPadding transformation, which corresponds to AES’s key purpose. Data input is in a byte array format. The doFinal() 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.

LISTING 13. Verify data with ECDSA.

```scala
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) 
            .setEncryptionPadding ( 
                KeyProperties.ENCRYPTION_PADDING_NONE) 
            .build())
}
```

LISTING 14. Import of AES key.

```scala
fun encryptData(dataToEncrypt: ByteArray) {
    val cipher = Cipher.getInstance("AES/GCM/NoPadding")
    val encryptedData = cipher
        .run({
            init(Cipher.ENCRYPT_MODE,
            entry, privateKey)
            doFinal(dataToEncrypt)
        })
    val vector = cipher.iv
```

LISTING 15. Encrypt data with AES.
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.

```java
val spec = GCMParameterSpec(128, vector)
val data = Cipher.getInstance("AES/GCM/NoPadding")
    .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.

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 utilized 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 including 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.

![Fig. 2. Mobile devices used for the measurements campaign (technical details are listed in Table 5).](image-url)
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 algorithm 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 |
|--------|-------|---------|----|-----|-----------------|-----------|---------|-------------|
| Texas Instruments | 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 |
| LG | G6 | Qualcomm MSM8996 Snapdragon 821 | 9 | 4 | 4 | Kryo | 2 | 1.6 |
| 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 |
| HTC | One M9 | Qualcomm MSM8994 Snapdragon 810 | 7 | 3 | 8 | Cortex-A53 | 4 | 1.5 |
| Motorola | Moto G5 Plus | Qualcomm MSM8953 Snapdragon 625 | 8.1 | 4 | 8 | Cortex-A53 | 4 | 2.0 |
| Samsung | S10e | Exynos 9820 | 9 | 8 | 8 | Mongoose M4 | 2 | 2.73 |
| Huawei | P20 Lite | HiSilicon Kirin 659 | 9 | 4 | 8 | Cortex-A53 | 4 | 1.7 |
| OnePlus | 7 Pro | Qualcomm SDM855 Snapdragon 855+ | 10 | 8 | 8 | Kryo 485 | 4 | 1.78 |
| Samsung | Galaxy S9+ | Exynos 9810 | 9 | 6 | 8 | Mongoose M3 | 4 | 2.7 |
| OnePlus | 6 | Qualcomm SDM845 Snapdragon 845 | 10 | 8 | 8 | Kryo 385 Silver | 4 | 1.7 |
| Huawei | P9 Lite | HiSilicon Kirin 650 | 7 | 3 | 8 | Cortex-A53 | 4 | 2.8 |
| Google | Pixel 3a | Qualcomm SDM670 Snapdragon 670 | 10 | 4 | 8 | Kryo 360 Gold | 2 | 2.0 |
| Samsung | Galaxy S6 | Exynos 7420 Octa | 7 | 3 | 8 | Cortex-A53 | 4 | 2.1 |
| Samsung | Note10+ | Exynos 9825 | 9 | 12 | 8 | Cortex-A53 | 4 | 1.9 |

**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.
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 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 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 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) 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,
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,
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. 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
A. Ometov et al.: Comprehensive and Reproducible Comparison of Cryptographic Primitives Execution

**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

VOLUME 9, 2021

**LIST OF ACRONYMS**

- **AES**
  - Advanced Encryption Standard
- **API**
  - Application Programming Interface
- **BLE**
  - Bluetooth Low Energy
- **DER**
  - Distinguished Encoding Rules
- **DSA**
  - Digital Signature Algorithm
- **EC**
  - Elliptic 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
- **MFG**
  - 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
A. Ometov, P. Masek, L. Malina, R. Florea, J. Hosek, S. Andreew, J. Hajny, J. Niutane, 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. Chatzikontantinou, 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/Mcr-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. Yi, 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.

**A. Ometov et al.: Comprehensive and Reproducible Comparison of Cryptographic Primitives Execution**

**A. 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.

**Y. Bardinova** received the Master’s degree in electronic engineering from the Saint Petersburg State University of Aerospace Instrumentation (SUAI), Russia, in 2016. He is currently a Researcher with the Brno University of Technology (BUT), Czech Republic. His research interests include wireless communications, information security, computing paradigms, blockchain technology, and wearable applications.

**K. Zhidanov** received the Specialist degree from the National Research University Higher School of Economics, National Research University Higher School of Economics, in 2015. He is currently a Researcher with the Brno University of Technology (BUT), Czech Republic. His research interests include wireless communications, information security, computing paradigms, blockchain technology, and wearable applications.

**M. Komarov** 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 co-authored 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.