Evaluation of Cryptographic Capabilities for the Android Platform

David González, Oscar Esparza(✉), Jose L. Muñoz, Juanjo Alins, and Jorge Mata

Network Engineering Department, Universitat Politècnica de Catalunya, Jordi Girona 1-3, Campus Nord UPC, 08034 Barcelona, Spain
oscar.esparza@entel.upc.edu

Abstract. Future networks will be formed by millions of devices, many of them mobile, sharing information and running applications. Android is currently the most widely used operating system in smartphones, and it is becoming more and more popular in other devices. Providing security to these mobile devices and applications is a must for the proper deployment of future networks. For this reason, this paper studies the cryptographic structure and built-in tools in Android, and shows that the operating system has been specially designed for plugging-in external cryptographic modules. We conclude that the best option for providing cryptographic capabilities is using these external modules. We show the existent options and compare some features, like licensing, source code availability and price. We define some requirements, evaluate each module, and provide guidelines for developers who want to use properly security primitives.

1 Introduction

The use of mobile devices has increased intensely over the last years, and it is becoming ubiquitous. A recent report, developed by the American research company eMarketer, states that mobile users are picking up smartphones instead of traditional mobile devices as they become more affordable [7]. So, trends carry us to a future where mobile applications will be involved in a lot of aspects of our daily life.

People use their mobile devices for more than the built-in functionalities, which has introduced a radical change in the concept of mobile device. Today, mobile devices are based on the idea that operating systems must support users installing embedded applications on their devices. This transformation has created a tempting marketplace for programmers and software companies. Thus, both smartphone manufacturers and developers of mobile operating systems have created centralized application market places. Well-known examples are Google Play, which hosts 950,000 applications and produces a daily revenue of about 12$ million for the top 200 applications [13], and App Store, which hosts 1 million applications and produces a daily revenue of 18$ million for the top 200 applications [13].

R. Doss et al. (Eds.): FNSS 2015, CCIS 523, pp. 16–30, 2015.
DOI: 10.1007/978-3-319-19210-9_2

From both development and research points of view, Android is the perfect target platform. On one hand, as market analysis show, Android is the most widespread mobile operating system [12]. Android is used by a wide range of devices of several manufacturers, e.g. Samsung, Sony and Huawei. It has a large number of applications available, and many software companies use one or more of its official marketplaces as a primary source for distribution. Android is free and open source. All development tools are available at no cost and run over all main operating systems [26]. Anyone can inspect the Android source code, and modify and recompile it to extend functionalities. The security of Android devices is becoming a hot research topic as mobile applications are managing more and more sensitive data every day.

The study carried out by William Enck et al. [8] clearly shows the need to improve security of Android applications. The authors analyzed 1,100 popular free Android applications and discovered several security flaws. The authors found that many developers fail to take the necessary security precautions and sensitive information was occasionally broadcast without being previously protected by means of encryption. This is even more important, since available information about how to develop secure applications for Android is quite reduced, especially regarding how cryptography works in Android and which are the available tools.

This paper is focused on analyzing the cryptographic features of Android, explaining the available tools to perform common cryptographic operations, and evaluating their performance using a testing methodology. Our assumption is that, given a set of security features, developers are going to map them to a set of strong cryptographic primitives. In our opinion, this study produces relevant results and could be of interest for both researchers and developers. This paper states that it is possible to provide security at a high level in Android applications today, but further efforts should be done to improve compatibility in previous, present and future operating system versions, as Android will probably be the leading operating system in future networks devices.

This paper is organized as follows: Sect. 1 introduces the problem of security in Android devices and applications; Sect. 2 summarizes some previous papers that studied performance of cryptographic tools on mobile devices, Java and Android; Sect. 3 briefly describes how cryptography works in Android; Sect. 4 introduces our testing application and methodology; Sect. 5 includes some of the most relevant performance results obtained with the testing application; and finally, in Sect. 6 we can find the conclusions of the paper.

2 State of the Art

Some authors decided to analyze the level of security that mobile operating systems offer, as well as their weak points. For example, a pair of comprehensive Android security assessments has been published by William Enck et al. [9] and Asaf Shabtai et al. [29]. Joseph Packy Laverty et al. took a slightly different approach and developed a detailed comparative analysis of security models among Android, iOS, Black Berry and Windows Mobile [14].

A very similar work is the proposal of Michael A. Walker [33], which proposes a standard method to evaluate the cryptographic capabilities and efficiency of Android devices. The author developed an Android testing application and determined a list of built-in available algorithms using a HTC G1 device with Android 1.6. As stated by the author, this initial project was meant as a starting point for future research. The study is very elaborate and presents, as of today, the unique available results about performance of cryptographic operations over Android. However, the scope of the study is limited.

Jeremy S. Nightingale studied six Java cryptographic providers and developed a comparative analysis for public key cryptography [15]. Nevertheless, this study is for the Java Platform Standard Edition (J2SE) [5], not for Android. Other similar studies [3,4] are rich in information and contain a detailed outline of the tools and methodologies used, but they target a different hardware.

3 Cryptography in Android

Android applications are written in a Java-based programming language, also called Android, which is not fully compatible with Java SE standards or applications. While Java core libraries are used, Android provides additional APIs to integrate with the operating system, the platform resources and the security model. It is also possible to use native code in C/C++ by means of the Java Native Interface (JNI) framework, or even writing a whole application in C/C++ [26], but this is discouraged because it makes the code non-portable.

Several versions of the operating system have been released (e.g. Froyo (2.2), Gingerbread (2.3), Ice Cream Sandwich (4.0), etc.), and version upgrade process has been difficult since its origins. Difficulties are mainly due to the lack of direct control of the operating system developers over the firmware, as well as the large range of device manufacturers. Although such fact is improving lately, statistics show how older versions are still running on older devices in greater or lesser extent, i.e. Gingerbread (17.8 %), Ice Cream Sandwich (14.3 %), Froyo (1.1 %), Honeycomb (0.1 %) [24].

Like in traditional Java, the overall design of the cryptography classes in Android is governed by the Java Cryptography Architecture (JCA), which was inherited from Apache Harmony. However, there are some significant differences, mainly originated by the limitations of mobile devices. The main difference is the non inclusion of an independent version of the SunJCE provider in Android. This means that Android does not include, by default, the same algorithms than Java. However, no list of discrepancies between the Android built-in cryptography and the traditional SunJCE is available at this moment.

Android source code is publicly available in a repository [1]. As usual, the source code follows a tree structure and is divided in branches. We are going to center in branches shown in Table 1. The core branch is libcore, which shows that Apache Harmony ships its own cryptographic provider, Crypto [19]. Additionally, Apache Harmony includes a Java Secure Socket Extension provider called HarmonyJSSE [21], which is based on the specifications of TLS v1 and

Table 1. Built-in provider/library by branch

Provider/Library	Branch
Crypto	platform/libcore [19]
HarmonyJSSE	platform/external/conscrypt [21]
Bouncy Castle	platform/external/bouncycastle [2]
OpenSSL	platform/external/openssl [23]
AndroidOpenSSL	platform/external/conscrypt [22]

SSL v3 protocols. However, the number of algorithms provided by Crypto is very limited, so Android engineers needed an alternative provider to cover all the security requirements that applications may need. Instead of programming their own provider from scratch, Android developers modified an existent Java provider, Bouncy Castle [16], and set it as the default provider. The original source code of Bouncy Castle is different from the modified version included in Android. In particular, some algorithms have been removed and some classes have been changed to improve both speed and memory consumption. The problem is that these changes vary depending on the Android version. In addition, some algorithms within Bouncy Castle have been combined with a well-known cryptographic library written in C, OpenSSL [32].

Our source code inspection revealed that the reduced set of capabilities provided by the providers Crypto, DRLCertFactory and HarmonyJSSE remains constant, while security services provided by Bouncy Castle vary from version to version. However, the support of SSL and TLS protocols is guaranteed since API level 1 [20]. Therefore, all versions of the platform provide built-in tools for establishing a secure communication between an Android device and a server machine. For implementing both SSL and TLS, Android uses code from both Bouncy Castle and OpenSSL. A list of all built-in providers, with the minimum version from which are included, is detailed in Table 2.

Even though Android ships with a set of built-in tools which cover most usual cryptographic algorithms and standards, we cannot be sure that such built-in libraries are updated with the last patches for all versions of the operating system. Moreover, there are discrepancies between versions, and older versions,

Table 2. Built-in cryptographic service providers

Provider	From version
Crypto	1.5 or previous
HarmonyJSSE	1.5 or previous
DRLCertFactory	1.5 or previous
Bouncy Castle	1.5 or previous
AndroidOpenSSL	3.0

which still have an important market share, lack some popular algorithms, e.g. Gingerbread (2.3) lacks SHA224 and ECDSA. For this reason we recommend developers to use in their applications the last version of a third-party cryptographic service provider, as this will allow them to control critical updates.

4 Testing Application

We recommend developers to include external cryptographic service providers in their applications. For this reason, we have developed a simple application to test these providers and rank their performance.

Cryptographic providers targeting the Android platform (or stating being compatible with it) do not seem to abound. We only found these two options:

- IAIK-JCE, which can be downloaded from [10].
- SpongyCastle: as previously mentioned, Android ships with a cut-down version of Bouncy Castle. But installing the classic Bouncy Castle library is impossible due to classloader conflicts, as the names for most packages and classes are the same. SpongyCastle [31] is a repackage of the classic Bouncy Castle library provided by Roberto Tyley, an independent developer. In our tests, we are going to use a repackage of the classic Bouncy Castle library, which was performed by following the guidelines of Roberto Tyley. In fact, we do not recommend developers to use SpongyCastle directly in applications, but as a guide for repackaging the classic Bouncy Castle. The main reason is that this project maintenance relies on this only developer.

We are also interested in traditional cryptographic providers for Java that may be compatible with Android. Four non-commercial cryptographic Java libraries were found:

- Logi Crypto [28], which was discarded because it was not designed to be compatible with the JCA/JCE structure, and hence cannot be easily integrated into Android.
- GNU Crypto [11], which was not well documented, and its interaction with the JCA/JCE structure was messy. After integrating the provider and making some quick encryption/decryption tests, we realized that GNU Crypto was slow and difficult to integrate into Android, so it was discarded too.
- Cryptix [27], which was inspected and tested and no compatibility problems were found.
- FlexiProvider [17], which was also tested with no problems.

The summary of the available providers compatible with Android is detailed in Table 3.

4.1 Cryptographic Requirements

Choosing an appropriate cryptographic algorithm is essential in any system with security requirements. A large number of cryptographic algorithms exist, but the

Table 3. External cryptographic service providers

Provider	Last version
Bouncy Castle	1.50
Cryptix	1.3
FlexiProvider	1.7.7
IAIK-JCE	5.2

devices used for communications sometimes have limited processing capacity and reduced storage capacity. So, it is important to have available algorithms that work correctly in devices with scarce resources, while maintaining a high level of security. Moreover, algorithms should be widely tested through time, and security holes should be solved with the help of the cryptographic community.

Thus, we prepared a list of requirements that cryptographic providers must accomplish in order to provide enough tools for implementing common security features. For completeness of the study, our proposal not only addresses common cryptographic algorithms, but other related features. The coverage of these features takes into account additional capabilities that are usually required in application security, e.g. opening and creating digital envelopes, securely distributing key pairs, establishing secure communications, etc. The complete list of requirements is specified in Table 4.

The proposal of algorithms and standards to evaluate is mostly based on the recommendations of the NIST [30]. We focused on the Federal Information Processing Standards (FIPS), a compilation of standards and guidelines issued by NIST for government use. Nevertheless, we have taken into account recommendations of other institutions (as detailed in Table 4), e.g. ITU, IETF, RSA Security Labs. Although Triple-DES and SSL are no longer recommended by NIST, we consider them for compatibility reasons since they still are widely used. Table 5 complements Table 4 by specifying recommended parameters.

4.2 Coverage of Requirements

By inspecting the official documentation of the selected providers, the classes in charge of registering the different algorithms, and the source code, we prepared Tables 6 and 7. These tables detail the coverage of algorithms and standards according to requirements introduced in Subsect. 4.1. Support of SSL and TLS is guaranteed since API level 1 [20], and covered by the operating system. Table 8 details the supported protocol versions for the built-in Android JSSE Provider.

4.3 Testing Framework

We developed a testing application to carry out performance tests in an easy and efficient way. We also established a testing methodology to set up a fitting benchmarking environment. The testing application has been designed so that

Table 4. Requirements for development of secure applications

Requirement	Type	Standard
SHA-1	Hash Function	FIPS 180
SHA-256	Hash Function	FIPS 180
HMAC	Message Authentication Code	FIPS 198
PBKDF2	Key Derivation Function	PKCS#5 v2.0
AES	Symmetric Cipher	FIPS 197
3DES	Symmetric Cipher	ANSI X9.52
RSA	Asymmetric Cipher	PKCS#1 v1.5, PKCS#1 v2.1
RSA	Signature Algorithm	PKCS#1 v1.5, PKCS#1 v2.1
DSA	Signature Algorithm	FIPS 186
X509	Digital Certificate	RFC 5280
PKCS #7	Digital Envelope	PKCS#7
PKCS #12	Information Exchange Syntax	PKCS#12
SSL	Secure Transport Protocol	RFC 6101
TLS	Secure Transport Protocol	RFC 2246
		RFC 4346
		RFC 5246

Table 5. Required algorithms with parameters

Algorithm	Key length (bits)	Operation mode	Padding
SHA-1	N/A	N/A	N/A
SHA-256	N/A	N/A	N/A
HMAC	Variable	N/A	N/A
PBKDF2	Variable	N/A	N/A
AES	128, 192, 256	CBC, OFB, CFB, CTR	PKCS#5 or PKCS#7
3DES	192	CBC, OFB, CFB, CTR	PKCS#5 or PKCS#7
RSA	2048	ECB	PKCS#1, OAEP
RSA	2048	N/A	PKCS#1, PSS
DSA	224(key)	N/A	PKCS#1
	2048(group)		

a user can select a subset of the algorithms, depending on the variants of the algorithms to be tested, as well as the parameters to use, e.g. length of the input, key size, number of samples, etc. Once the user finishes the set up, the application runs the tests, and it stores all the performance data on the memory of the device for later processing.

Table 6. Coverage of algorithms

Requirement	BC	Cryptix	Flexi	IAIK
SHA1	X	X	X	X
SHA256	X	X	X	X
HMAC/SHA1	X		X	X
HMAC/SHA256	X		X	X
PBKDF2/HMAC/SHA1	X		X[1]	X
PBKDF2/HMAC/SHA256	X			X
AES/CBC/PKCS5	X		X	X
AES/CFB/PKCS5	X		X	X
AES/OFB/PKCS5	X		X	X
AES/CTR/PKCS5	X		X	X
3DES/CBC/PKCS5	X	X	X	X
3DES/CFB/PKCS5	X	X	X	X
3DES/OFB/PKCS5	X	X	X	X
3DES/CTR/PKCS5	X		X	X
RSA/ECB/PKCS1	X	X	X	X
RSA/ECB/OAEP	X	X	X	X
RSA/SHA1/PKCS1	X	X	X	X
RSA/SHA256/PKCS1	X	X	X	X
RSA/SHA1/PSS	X	X	X	X
RSA/SHA256/PSS	X	X		X
DSA/SHA1/PKCS1	X	X	X	X
DSA/SHA256/PKCS1	X		X	X

Table 7. Coverage of cryptographic standards

Requirement	BC	Cryptix	Flexi	IAIK
X509	X		X	X
PKCS7	X			X
PKCS12	X		X	X

Table 8. Built-in secure communication protocols

Requirement	HarmonyJSSE
SSL	v2, v3
TLS	v1

We use the built-in function nanoTime to determine the amount of time consumed by a given cryptographic operation. According to the official documentation, nanoTime returns the value of the most precise system timer available with nanosecond precision. The clock accessed is guaranteed to be monotonic and suitable for interval timing when the interval does not span device sleep.

The key generation process required by public key cryptography is more complex than the one required for secret key cryptography, since it involves more costly mathematical operations which tend to be long in time. Nevertheless, background execution and memory management are very important in Android, because they are tightly related with power consumption and memory efficiency [25]. For this reason, we decided to generate the secret keys in the mobile devices, while the tests using public key algorithms took pre-generated keys. These pre-generated keys were stored using personal identity information standards and included beforehand on the testing application.

Another problem we found in Android was the presence of outlier measures. Since the operating system has been devised as an application ecosystem, all applications are kept alive as long as possible, and the scheduler considers them all when dispensing execution time. As a consequence, we found that, even shutting down all unnecessary applications, tests contained outlier measures, and these deteriorated both mean and standard deviation values. This effect was also noticed in the study lead by Michael A. Walker [33].

We analyzed the samples and noticed that the generation of outliers was not periodic. Then, we redesigned the testing methodology, so that the test application could be able to mitigate the effect of the outlier measures by itself. For each atomic cryptographic operation to benchmark, the application performs a previous round of measures and computes an initial estimation of the measure. Then, uses such estimation to fix a threshold value, i.e. by multiplying the estimation value by a threshold factor defined by the user. This outlier threshold will be used later to check if the measures are out of the range of expected values. Then, the application proceeds and gathers a second round of samples. Now the application knows the range of expected values, so it is capable of disscarding the outlier measures using the threshold. Once included the new methodology in our Android testing application, we found that with a small number of discarded measures (less than a 5 %) the standard deviation is considerably reduced, around 10 and 50 times smaller depending on the parameters of the test.

5 Measures

5.1 Tuning

In Android, the CPU frequency may change depending on the power consumption, affecting the device performance [6] and the reliability of the results. We carried out some test trials using CPU Spy, confirming that the frequency kept stable, without fluctuations, while executing the tests. For reducing the effects of possible sources of interference, we forced quit all, non-critical, running applications and services in the device. In addition, we disconnected both Wi-Fi

and Bluetooth services in order to avoid external interferences that could have affected the behavior of the device during the tests.

Table 9. Testing Hardware

Device	Hardware	Operating System
Nexus S	ARM Cortex-A8 single-core 1GHz 512 MB RAM	Android 2.3 upgraded to Android 4.1.2

Tests were carried out in a Samsung Nexus S, upgraded to Android 4.1.2. Detailed characteristics of the testing hardware are summarized in Table 9. All tests were conducted by gathering 10.000 samples by operation. This number of samples does not include the additional 1.000 samples used for estimating the value beforehand and discarding outlier measures. These numbers were chosen after a short period of trial and error, in which we confirmed how fixing these parameters the standard deviation of the measures decreased considerably.

5.2 Performance Evaluation

From the collected data we generated more than 20 graphics for different operations and algorithms, varying parameters, key sizes and lengths of inputted data. Tables 10, 11 and 12 and Figs. 1, 2, 3 and 4 show a representative sample for the most common algorithms, i.e. SHA-2, AES, RSA and DSA. Even from these samples, one can discern several interesting trends.

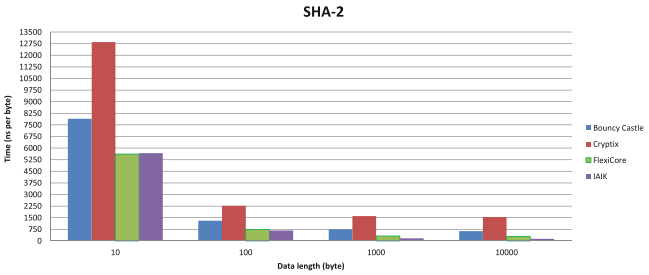

Fig. 1. Mean micro-second time per byte when hashing using SHA-2 for varying data lengths (10,000 samples, less 1 % of outliers)

For secret key cryptography, FlexiCore and IAIK-JCE are faster than Bouncy Castle and Cryptix. FlexiCore performed better for SHA-1 while IAIK-JCE performed better for SHA-2. For example, according to Table 10, FlexiCore is

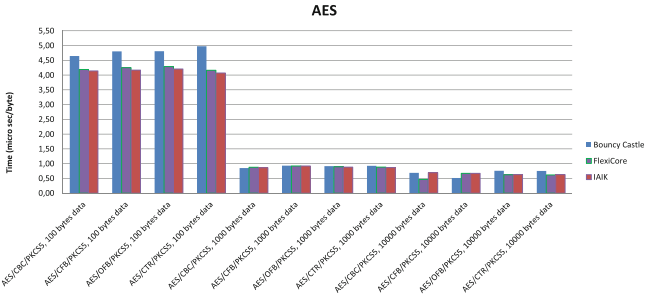

Fig. 2. Mean micro-second time per byte when encrypting using AES with a 128-bit key for varying data lengths and operation modes (10,000 samples, less 2 % of outliers)

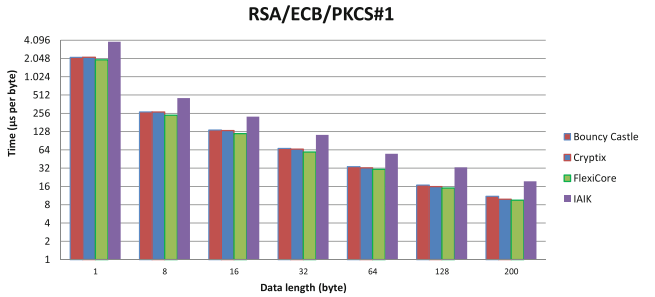

Fig. 3. Mean micro-second time per byte for encrypting using RSA with PKCS#1 padding (10.000 samples, less 1 % of outliers)

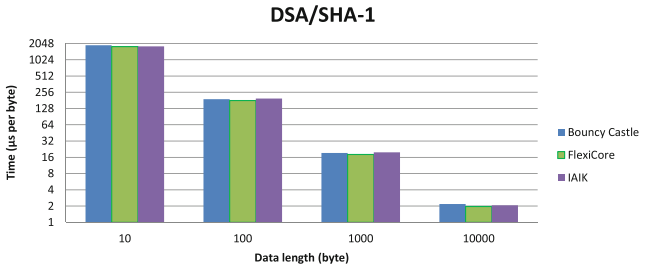

Fig. 4. Mean micro-second time per byte for signing using DSA/SHA-1 (10.000 samples, less 0.1 % of outliers)

Table 10. Average time for hashing a 100-bytes input data with SHA-1 & SHA-2 (ns/byte)

Algorithm	BC	Cryptix	Flexi	IAIK
SHA-1	724	715	572	603
SHA-2	1309	2270	731	671

Table 11. Average time for encrypting a 100-bytes input data with AES & 3DES (μs/byte)

Algorithm	BC	Cryptix	Flexi	IAIK
AES/CBC/PKCS#5, 128-bits key	4.65	N/A	4.19	4.14
3DES/CBC/PKCS#5, 192-bits key	8.73	5.66	6.42	7.33

Table 12. Average time for encrypting a 128-bytes input data with RSA & for signing a 100-bytes input data with DSA (μs/byte)

Algorithm	BC	Cryptix	Flexi	IAIK
RSA/PKCS#1, 2048-bits key	16.88	15.87	15.08	32.96
DSA/SHA-1, 224-bits key	190.73	N/A	179.96	196.02

35 ns/byte faster than IAIK-JCE when hashing 100 bytes using SHA-1, and both FlexiCore and IAIK-JCE are more than 100 ns/byte faster when compared with Bouncy Castle and Cryptix. By the contrary, when hashing 100 bytes using SHA-2 the differences increase: IAIK-JCE is 60 ns/byte faster than FlexiCore, and both FlexiCore and IAIK-JCE are about 600 ns/byte faster than Bouncy Castle. Figure 1 shows how such differences remain similar for smaller and longer lengths of the input data. Although it is not included in this data sample, our study showed that Bouncy Castle is the fastest option when generating HMACs.

When encrypting, IAIK-JCE performed better for AES and Cryptix for 3DES. Table 11 shows how, when encrypting a 100-bytes datablock using CBC mode and PKCS#5 padding, IAIK-JCE performed 0.05 μs/byte better for AES than FlexiCore, the second best, and Cryptix performed 0.76 μs/byte better for 3DES, being FlexiCore the second best too. Just remark the small differences of speed between IAIK-JCE and FlexiCore, as shown in Figs. 1 and 2. From Figs. 1 and 2 one can deduce that Cryptix is the slowest provider. Meanwhile, Bouncy Castle keeps decent differences with IAIK-JCE and FlexiCore.

For public key cryptography, FlexiCore is the fastest provider for encrypting and signing with RSA, and for signing with DSA. However, differences were small. For example, according to Table 12, when encrypting a datablock of 128 bytes with RSA and a 2048-bits key, there was a difference of 0.79 μs/byte with Cryptix and of 1.8 μs/byte with Bouncy Castle. Another example, are the differences of 10.77 μs/byte, between FlexiCore and Bouncy Castle, and of 16.06 μs/byte, between FlexiCore and IAIK-JCE, when signing a datablock of 100 bytes with DSA and a 224-bits key.

An important trend we observed in Fig. 3, is that IAIK-JCE is the slowest provider when encrypting and verifying signatures with RSA. Nevertheless, without being able to inspect the source code, we cannot come up with a logic explanation for this issue. IAIK-JCE performed as well as FlexiCore and Bouncy Castle when signing and verifying with DSA as indicates Fig. 4.

These small differences regarding performance when using public key algorithms (see Figs. 3 and 4) were expected. In public key algorithms, the modular product, exponentiation and inversion operations are more expensive in terms of time than other operations, e.g. hashing. Therefore, the speed of these modular operations is what ultimately defines the speed of the implementations. Speeds were so close because all four cryptographic providers use the Android native implementation of the class BigInteger [18]. One can corroborate this by inspecting the source code of Bouncy Castle, Cryptix and FlexiCore.

If we consider both performance results and coverage of requirements, as well as source code availability and license, FlexiCore and Bouncy Castle are, individually, very good choices. Both providers have permissive licenses, give access to the source code and offer a good coverage of usual cryptographic requirements: Bouncy Castle covers them all, while FlexiCore covers a large number. FlexiCore provides a very fast implementation, and Bouncy Castle is slower, but remains close in performance. Nevertheless, it is not clear how the inclusion of the ASN.1 CoDec package affects in the use of FlexiCore, since it is GPL licensed.

IAIK-JCE is also recommended in case developers already purchased a commercial license. IAIK-JCE covers all requirements, it shows a good performance and it is specially intended for Android, unlike the previous providers. Moreover, IAIK-JCE provides the fastest implementations for the common algorithms SHA-2 and AES. Despite differences are small, in general FlexiCore and IAIK showed better performance.

6 Conclusions

Providing security to Android devices and applications is, for sure, one of the main objectives to achieve prior to the proper deployment of future networks. Android has inherited the cryptographic design of Java. Cryptographic services are provided by a series of modules, some of them are built-in, e.g. Bouncy Castle, Crypto, HarmonyJSSE, etc. Nevertheless, we cannot guarantee that these built-in modules are updated with the last patches. Moreover, old versions of Android, some still with an important market share, are limited in functionality and lack some common algorithms. For this reason, our recommendation is using a third-party provider to provide security to applications.

All the evaluated providers have both positive and negative aspects. Bouncy Castle covers all usual cryptographic requirements, provides access to the source code and is free. So, Bouncy Castle would be a good option if we are interested on covering a wide range of algorithms and standards at no cost. FlexiCore provides access to the source code and it is free too, but covers a slightly smaller range of requirements. However, FlexiCore has proved its superiority in speed, being the fastest provider for a lot of common algorithms, e.g. SHA-1, RSA, DSA. So, FlexiCore is the best option if we are interested on speed at no cost. IAIK-JCE covers all usual cryptographic requirements and has proved to be very fast. Nevertheless, it requires purchasing a commercial license when it is used in commercial products. So, IAIK-JCE is the best option for those developers

who are interested on both speed and coverage of algorithms, and are willing to pay. We do not recommend using Cryptix, since covers a very limited range of requirements, and it is deprecated, meaning that its maintenance has been discontinued long time ago.

Acknowledgments. This work was supported partially by the Spanish Research Council with Project SERVET TEC2011-26452, and by Generalitat de Catalunya with Grant 2014-SGR-1504 and 2014-SGR-375 to consolidated research groups.

References

1. Android Git repositories. https://android.googlesource.com/
2. Bouncy Castle repository. Android Git repositories. https://android.googlesource.com/platform/external/bouncycastle/
3. Abusharekh, A.: Comparative Analysis of Multi-Precision Arithmetic Libraries for Public Key Cryptography. Ph.D. thesis, George Mason University, Washington, DC (2004)
4. Bingmann, T.: Speedtest and comparsion of open-source cryptography libraries and compiler flags. https://panthema.net/2008/0714-cryptography-speedtest-comparison/
5. Campione, M., Walrath, K., Huml, A.: The Java Tutorial: A Short Course on the Basics, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
6. Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2010, pp. 21–21. USENIX Association, Berkeley (2010)
7. eMarketer Inc.: 2 billion consumers worldwide to get smartphones by 2016 (2014). http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694
8. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, pp. 21–21. USENIX Association, Berkeley (2011)
9. Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE Secur. Priv. **7**(1), 50–57 (2009)
10. Institute for Applied Information Processing and Communication. Gratz University of Technology. Core Crypto Toolkits. https://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits
11. Free Software Foundation. The GNU Crypto project. http://www.gnu.org/software/gnu-crypto/
12. Goasduff, L., Rivera, J.: Gartner says smartphone sales surpassed one billion units in 2014 (2015). http://www.gartner.com/newsroom/id/2623415
13. Jones, C.: Google Play catching up to Apple's App Store (2013). http://www.forbes.com/sites/chuckjones/2013/12/19/google-play-catching-up-to-apples-app-store/
14. Laverty, J.P., Wood, D.F., Kohun, F.G., Turchek, J.: Comparative analysis of mobile application development and security models. Issues Inf. Syst. **12**(1), 301–312 (2011)
15. Nightingale, J.S.: Comparative analysis of Java cryptographic libraries for public key cryptography (2006). http://teal.gmu.edu/courses/ECE746/project/reports_2006/JAVA_MULTIPRECISION_report.pdf

16. The Legion of Bouncy Castle. Boncy Castle. http://www.bouncycastle.org/java.html
17. Research Group of Prof. Dr. Johannes Buchmann. FlexiProvider. http://www.flexiprovider.de/
18. Android Open Source Project. BigInteger class, Android API. http://developer.android.com/reference/java/math/BigInteger.html
19. Android Open Source Project. Crypto Provider, Android platform 'libcore' repository. Android Git repositories. https://android.googlesource.com/platform/libcore/+/master/luni/src/main/java/org/apache/harmony/security/provider/crypto/CryptoProvider.java
20. Android Open Source Project. javax.net.ssl package, Android API. http://developer.android.com/reference/javax/net/ssl/package-summary.html
21. Android Open Source Project. JSSE Provider, Android platform 'conscrypt' repository. Android Git repositories. https://android.googlesource.com/platform/external/conscrypt/+/master/src/main/java/org/conscrypt/JSSEProvider.java
22. Android Open Source Project. OpenSSL Provider, Android platform 'conscrypt' repository. Android Git repositories. https://android.googlesource.com/platform/external/conscrypt/+/master/src/main/java/org/conscrypt/OpenSSLProvider.java
23. Android Open Source Project. OpenSSL repository. Android Git repositories. https://android.googlesource.com/platform/external/openssl/+/master
24. Android Open Source Project. Platform versions. http://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net#Platform
25. Android Open Source Project. Processes and threads. http://developer.android.com/guide/components/processes-and-threads.html
26. Android Open Source Project. Android Developers (2014). http://developer.android.com/index.html
27. The Cryptix Project. Cryptix. http://www.cryptix.org/
28. Ragnarsson, L.: The logi.crypto Java package. http://www.logi.org/logi.crypto/devel/
29. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google android: a comprehensive security assessment. IEEE Secur. Priv. **8**(2), 35–44 (2010)
30. National Institute Standards and Technology. Cryptographic Toolkit. http://csrc.nist.gov/groups/ST/toolkit/index.html
31. Roberto Tyley. SpongyCastle. http://rtyley.github.io/spongycastle/
32. Viega, J., Chandra, P., Messier, M.: Network Security with Openssl, 1st edn. O'Reilly & Associates Inc., Sebastopol (2002)
33. Walker, M.A.: Standard method of evaluating cryptographic capabilities and efficiency for devices with the Android platform (2010). https://www.truststc.org/education/reu/10/Papers/WalkerM_paper.pdf

http://www.springer.com/978-3-319-19209-3

Future Network Systems and Security
First International Conference, FNSS 2015, Paris, France, June 11-13, 2015, Proceedings
Doss, R.; Piramuthu, S.; ZHOU, W. (Eds.)
2015, X, 195 p. 75 illus., Softcover
ISBN: 978-3-319-19209-3