Offensive Technologies project

Weak key cracking of Android applications

Authors:
Cedric Van Bockhaven
University of Amsterdam

Sharon Gieske
University of Amsterdam

June 2014

UNIVERSITY OF AMSTERDAM

Abstract

Graduate School of Informatics

Master System and Network Engineering

Weak key cracking of Android applications

by C. Van Bockhaven
S.A. Gieske

The Google Play store serves more than 800,000 applications. Currently, no statistics are available on the certificate landscape. An inventory is made of 75,000 certificates of which most of the applications use SHA1 with RSA as signature algorithm. These certificates are tested for weak moduli in the case of RSA, and for reuse of signature parameters in the case of DSA. No weak moduli have been found that indicate a weak random number generator. However, two 512-bit RSA certificates were factored in a clustered effort for which the technology is publicly available.

Contents

Chapter 1

Introduction

The use of mobile apps gains an increasing popularity. Portio Research [1] estimates 1.2 billion people worldwide were using mobile apps at the end of 2012 and forecasts a growth of 29.8 percent each year. These mobile apps can be distributed via several channels, for example as a download on a website or through distribution platforms such as the Google Play store.

The Google Play store is a digital distribution platform for the Android operating system, operated by Google and considered as one of the largest stores with over 800,000 apps in research by Canalys[2]. Developers of mobile applications can publish their application on the Google Play store. These applications are created with the Android SDK and are published as self-contained Android Package files (APKs). These APKs contain a certificate and can be used to authenticate the origin of the application. The Android system requires that applications are digitally signed with a certificate of which the private key is held by the developer [1]. These certificates are often self-signed. Since it is not signed by another third party such as a certificate authority, an application can be updated solely with the private key.

This project assumes that there are weak keys used for certificates of APKs which can potentially be cracked to obtain the private key. The presence of weak keys may have different reasons: developers that do not use true random number generators, the system on which the key generation takes place has a broken cryptographic implementation (Debian weak keys), or the key length is too short which reduces the factoring effort. This could lead to cryptographic attacks with which the private key can be derived.

When the private key is obtained, a malicious user could create an application that is seemingly created by a trusted developer. This introduces security risks.

1.1 Research question

During this project, we focus on the following research questions:

- What is the overall quality of the cryptographic keys used to sign APKs?
- How can we identify (and possibly crack) weak cryptographic keys?
- What are possible attack vectors once the private key has been obtained?

[1] http://developer.android.com/tools/publishing/app-signing.html

1.1.1 Academic aspect

We believe that this research is M.Sc. worthy because security is an emerging issue, and this project hopes to demonstrate that even with nowadays cryptography, the weakest link may in fact be a careless developer or an implementation mistake.

In this research we will exploit mathematical properties of public key cryptography in order to derive private keys. Since there are no known algorithms yet for the underlying problems of public key cryptography, i.e. integer factorization, we rely on human error (the incorrect use of keys, the use of weak keys and so on).

1.1.2 Innovative aspect

No previous research has been done on how developers sign their applications and the overall quality of the used signing keys. For instance: developers are forgetful and may use development keys for publishing the production version of the app, or the implementation of the crypto library may contain flaws.

When obtaining the private key of the digital certificate, a malicious user could impersonate the developer and upload a trusted malevolent application.

1.2 Ethics

It is possible (and our intention) to crack the associated private keys of the certificates bundled into an Android application. These private keys are meant to be known by the developer only. However, when a malicious party has access to these keys the developer could be impersonated. We will only attempt impersonation in a closed test environment.

1.2.1 Responsible disclosure

We will not disclose personally identifiable information in the report. In the event that we find severe security issues with applications, we will attempt to notify the developers.

1.3 Related research

We have not found previous research on the current key landscape of Android applications. However, there is a lot of research done on how Android applications make use of

the provided cryptographic libraries that ship with the operating system. Fahl et al. [3] made an analysis of how well SSL functionality is implemented in Android applications.

Regarding the use of adequate key lengths, Cavallar et al. [4] managed to factor a 512-bit RSA modulus in 1999. Later on, a new record was set to factor a 768-bit modulus using the Number Field Sieve by Kleinjung et al. [5].

In 2002, Nguyen and Shparlinski [6] researched how the reuse of DSA parameters can lead to the recovery of the private key. In 2010, Sony reused signature parameters to sign authorized software for the PS3, which led to recovery of their private signing key [7].

Research has been done on weak random number generators and weak RSA moduli. In 2006, an incorrect patch for Debian's OpenSSL package resulted in guessable cryptographic key material, which also affected other Debian-based operating systems [8]. Nadia Heninger discovered a shocking number (64,000) of vulnerable certificates during a widespread SSH scan over the internet to look for co-prime public keys [9].

Chapter 2

Approach

In order to answer the research questions posed in section 1.1, the following steps are taken:

1. Retrieval of key data
2. Identification of weak keys
3. Derivation of private key from weak keys
4. Attack vectors

First, a collection of freely available Android application files (APKs) will be downloaded from the Google Play store. Key data will be extracted from their certificates, after which we identify the potentially weak keys. We then come up with attack scenarios on how the private key could be derived/cracked, and when the private key is obtained, how they could be used in a practical attack for malicious purposes.

2.1 Retrieval of key data

The collection of APKs is retrieved with the use of the *googleplay-api*[1] code. There is no index found which can be used to crawl for all APKs, therefore APKs are downloaded from multiple categories (described in table A.1) with the subcategories *Top Grossing*, *Top Selling Free* and *Top Selling New Free*.

APK files are essentially JAR files with a different extension, whereas JAR files are the same as ZIP files. They are different in that they contain a META-INF directory which contains information about the JAR file itself, such as signing and certificate data. We can distinguish three different types of files in the META-INF directory which are described in table 2.1

META-INF/	**Description**
CERT.SF	signature file, contains signed hashes
CERT.RSA CERT.DSA	contains the certificate in PKCS#7 DER format
MANIFEST.MF	contains index of files with corresponding hashes

Table 2.1: META-INF directory of APK

[1] https://github.com/egirault/googleplay-api

For every APK file, the certificate file is extracted and analysed for its signature algorithm and parameters.

2.1.1 Signature Algorithms

The Android System requires installed applications to be signed with a certificate by the developer. The certificate is created with public key cryptography which uses asymmetric cryptography using a public and private key. There are different signature algorithms that can be used to create certificates of which the three most common are Digital Signature Algorithm (DSA), RSA, and elliptic curve (ECC). There are standard tools such as Keytool[2] and Jarsigner[3] to generate keys and sign the APK. These standard tools are commonly used among developers and support the DSA and RSA algorithms, which are described below.

2.1.1.1 Digital Signature Algorithm

The Digital Signature Algorithm (DSA) is a United States Federal Government standard or FIPS for digital signatures proposed by The National Institute of Standards and Technology (NIST). DSA is attributed to David Kravitz and covered by U.S. Patent 5,231,668 [10].

In order to create the public and private key for the user, four algorithm parameters are chosen: p, q, g, x.

- q is chosen as an N-bit prime
- p is chosen as a L-bit prime modulus such that p–1 is a multiple of q
- g is chosen as a number whose multiplicative order modulo p is q
- x is the private key and chosen as a random number which holds the condition $0 < x < q$

The parameter y is calculated with the formula $y = g^x \ mod \ p$. The public key is (p, q, g, y) and the private key is x.

A signature (r, s) is created for message m:

- chose k where $0 < k < q$
- calculate $r = (g^k \ mod \ p) \ mod \ q$ with $r \neq 0$
- calculate $s = k^{-}1 \ H(m) \ + \ xr \ ) \ mod \ q$ with $s \neq 0$

[2] http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html
[3] http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html

2.1.1.2 RSA

RSA is an asymmetrical encryption algorithm developed by Ron Rivest, Adi Shamir and Len Adleman. It bases its security on the mathematical difficulty of factoring big integers and is without doubt the most popular public key cryptosystem in use today.

- Choose two distinct, big, random prime numbers of similar bit length: p, q
- Calculate the modulus $n = p \cdot q$
- Calculate $\varphi(n) = n - p - q + 1$
- Choose e such that $1 < e < \varphi(n)$, and e is coprime with $(p-1) \cdot (q-1)$
  Parameter e is kept as the public key exponent
- Choose d such that $d \cdot e \equiv 1 \mod \varphi(n)$
  Parameter d is kept as the private key exponent

The public key consists of (n, e) while the private key consists of (n, d). p, q and $\varphi(n)$ must also be kept secret since they can be used to calculate d.

Encryption and decryption of messages can be done according to the following schema:

Encryption	Decryption
$c \equiv m^e \mod n$	$m \equiv c^d \mod n$

Encryption/decryption schema with m being the message

2.1.2 Hashing Algorithms

The message that is signed by the signature algorithm is first hashed. This is done for efficiency since the signature will be shorter, for compatibility since different key (sizes) can be used, and for integrity of the message.

Hashing algorithms should pertain to the following properties:

Preimage resistance
: It must be difficult, given y, to find x such that $H(x) = y$.

Second preimage resistance
: It must be difficult, given x, to find x' such that $H(x) = H(x'), x \neq x'$.

Collision resistance
: It must be difficult to find a pair (x, x') such that $H(x) = H(x')$, $x \neq x'$. Collision resistance implies second preimage resistance, but does not guarantee preimage resistance.

We also declare the following attack definitions:

Collision attack

Find two different messages m_1 and m_2 such that $H(m_1) = H(m_2)$.

Chosen-prefix attack

Given two prefixes p_1 and p_2, the attack finds two appendages m_1 and m_2 such that $H(p_1||m_1) = H(p_2||m_2)$, where $||$ is the concatenation operator.

Using a second preimage attack, a file F' could be created that has the same hash as the original file F: $H(F) = H(F')$. If that file is then replaced in the JAR archive, the JAR would still appear to be valid as the cryptographic signing is only applied afterwards.

In this research a short literature analysis will be given on different hash functions found in the data collection and possible attacks. However, these specific attacks lie beyond the scope of this research.

2.2 Identification of Weak Keys and Derivation of its Private Key

As described in section 1.3, a great deal of research has been done on the derivation of private keys. During our research, we identify the following cateogires of weak keys:

1. Bad random number generators (bad RNG's)
2. Partial similar signatures for DSA certificates
3. Factorization for small RSA keys

2.2.1 Bad random number generators

Many cryptographic algorithms require good random numbers for key generation. These are made by random number generators and require uniqueness and unpredictability. A bad random number generator creates predictable 'random' numbers or non-unique numbers. This feature creates the possibility to recreate the random numbers and thereby create a database of potential public and private key pairs.

The identification of keys created with bad RNGs is done for DSA and RSA by comparing the retrieved key set with several databases[4] consisting of keys created with bad RNGs. A database containing Debian weak keys is also used for the comparison.

[4]Provided by Jeroen van Beek and Univ. of Michigan - http://www.scans.io

2.2.2 Partial similar signatures for DSA certificates

Weak parameters in DSA keys use similar signatures, and can be identified by finding two signatures (r_A, s_A), (r_B, s_B) in our data set, where $r_A = r_B$ *and* $s_A \neq s_B$. This is the result of a repeated k value for different messages. With the use of these two signatures k can be derived and consequently the private key x can be calculated. This derivation is described below:

Given that:

$$\begin{aligned} S_A &= k-1\ (H(M_A)\ +\ x*r)\ mod\ q \\ S_B &= k-1\ (H(M_B)\ +\ x*r)\ mod\ q \end{aligned} \tag{2.1}$$

The following holds:

$$S_A\text{–}S_B = k-1(H_A + x*r)\text{–}k-1(H_B + x*r) \tag{2.2}$$

$$S_A\text{–}S_B = k-1(H_A + x*r\text{–}H_B\text{–}x*r) \tag{2.3}$$

$$S_A\text{–}S_B = k-1(H_A\text{–}H_B) \tag{2.4}$$

Therefore k can be derived as:

$$k = (H_A\text{–}H_B)/(S_A\text{–}S_B) \tag{2.5}$$

Now the private key x can be derived by:

$$x = ((s\ *\ k)\ \text{–}H(m)) * r - 1\ mod\ q \tag{2.6}$$

2.2.3 Factorization of RSA keys

Pertaining to weak random number generators, when parameters p and q aren't sufficiently random prime numbers, the resulting modulus may be weak. Efforts by Heninger et al. [9] have shown that 1 in 200 TLS hosts served public keys that could be factored due to poor randomness. If two integers share a single prime factor, it is very easy to compute that prime factor by calculating the greatest common divisor. Heninger et al. calculated the common divisor for every pair of RSA moduli in their database, and found tens of thousands of factorable RSA keys which shared common factors with each other, allowing them to compute the private keys.

With regard to the key length, a length should be chosen that cannot be factored at this moment. The current biggest RSA key that has been factored is RSA-768 (232 decimal digits), and was factored in December 2009 by Kleinjung et al. [5].

The attacks that are carried out during this research don't try to attack the signing algorithm, but are based on factoring weak moduli. They can be factored either due to a weak random number generator or their length.

2.3 Attack vectors

When weak keys are found in the APK collection, the private key can be derived. This derivation is dependent on the type of weak key found. The methods for the derivation process are described in the previous sections.

After the private key has been recreated, a (slightly modified) application will be signed with this key and a new APK is created. Different ways to upload this new APK on a smartphone will be tested. A man-in-the-middle attack is performed in order to change the requested APK to the forged APK during a download from the Google Play store.

Chapter 3

Results

This chapter describes the results of this research. First, the collection of retrieved certificates from the APKs is analysed for their algorithms and properties. Then, weak keys are identified by looking for bad RNGs, partially similar DSA signatures, and small key sizes for RSA moduli. After the identification of a weak key, its private key will be derived. Finally, the private key will be used to forge a signature on a modified APK and attempts are made to download the modified APK on the smartphone without warnings.

3.1 Overview of certificates

In this section, the information gained from the certificates is analysed. It describes the signature algorithms and hashing algorithms found for the APKs. It also shows the different key sizes used for the found signature algorithms.

3.1.1 Signature Algorithms

In total 75,126 APKs were retrieved from the Google Play store. The certificates found are mostly created using the RSA algorithm. Approximately 97.25 % of the total APKs use the RSA algorithm and 2.75% use the DSA algorithm. No other signature algorithms are found. Several different hashing algorithms are found, with SHA-1 being the most used algorithm. The distribution of the different signature and hashing algorithms is found in table 3.1.

Signature algorithm		Total
DSA	SHA1	2,068
RSA	MD2	11
	MD5	4,099
	SHA1	57,224
	SHA256	11,718
	SHA512	6
Total apk		75,126

Table 3.1: Distribution signature algorithms

Table 3.2 shows the distinct certificates found with the specified signature and hashing algorithms. This shows that many apps are created by the same developers, as the certificate is developer bound.

Signature algorithm	**Distinct certificates**
SHA1withDSA	1116
MD2withRSA	7
MD5withRSA	1701
SHA1withRSA	33577
SHA256withRSA	8572
SHA512withRSA	3

TABLE 3.2: Distinct signature algorithms as found in the APKs

3.1.2 Hashing Algorithms

An interesting aspect in these results are the older hashing algorithms that are still used. Research by Wang and Yu [11] demonstrates the feasibility of finding a collision (i.e. two inputs producing the same hash value) of MD5 hashes. The MD2 hashing algorithm is also proven to not be unidirectional in research by Muller [12]. These two hashing algorithms are not in the approved algorithms list for generating a condensed representation of a message (message digest) as defined by NIST[13].

For MD5, a chosen-prefix attack has been developed and improved by Stevens et al. [14] that can find colliding messages in approximately 2^{39} calls to the MD5 compression function. However, such an attack could not be mounted in the forging of a file that has the same hash as a predefined input file. A preimage attack, that would allow to find a collision for a hashed file in the JAR archive, is at the moment still too computationally difficult. The best available attack that is better than an exhaustive search, is described by Sasaki and Aoki [15] and requires about $2^{123.4}$ calls.

In 2009, MD2 was shown to be vulnerable to a collision attack by Rogaway and Shrimpton [16] with a time complexity of $2^{63.3}$ compression function calls. This is slightly better than the birthday attack, which has a complexity of $2^{65.5}$ calls. The currently best known preimage attack that could be used to forge MD2 hashed files in the JAR archive, is described by Thomsen [17] with a time complexity of 2^{73} compression function calls. This is not yet usable in a practical attack, but comes closer to a reasonable time complexity.

For SHA1, as of 2012, the most efficient attack against SHA-1 is considered to be the one by Stevens [18], which has an estimated complexity of 2^{61} to find a full collision.

A generic second-preimage attack as exposed by Kelsey and Schneier [19] is applicable to hashes that use the Merkle–Damgård construction, such as MD2, MD5 and SHA1. The cost of this is $2^{n-\log_2 K} + 2^{\frac{n}{2}+1}$ hash rounds for these hashing algorithms, where K is the data size, and n is the digest size. However, that's not exactly cheap at all either.

We can conclude that at this moment, there is no viable attack vector to find collisions on these hashes since the original file is not under our control. Otherwise, a reasonably trivial MD5 chosen-prefix attack could have been mounted.

3.1.3 Key Sizes

The certificates using the DSA signature algorithm all use the same key size, i.e. 1024-bit for p and 160-bit for q. This can be a result of the recommendation for DSA key sizes by NIST[20] in 2000, where the key size needs to be between 512 and 1024 bits, and a multiple of 64. However, the latest recommendations by NIST[21] in 2013 also recommends higher key sizes to be used.

In contrast, the RSA key sizes used for the certificates range between the length of 512-bit and 8192-bit. The key size distribution can be found in table 3.3. The NIST recommendations specify three choices for the length of the modulus (i.e., nlen): 1024, 2048 and 3072 bits for Federal Government. However, other key sizes are usable and, as shown in table 3.3, lengths of multiples of 1024 are commonly used. There are also some outliers. These can be explained as developers use these key sizes to protect themselves (or win time against) from database lookup attacks when a popular key size, such as 1024, is compromised. The most interesting find is the presence of 512-bit RSA keys, since these keys have been factored before [4].

Size	**Total**		**Size**	**Total**
512	3		2049	4
1023	53		2560	2
1024	52,509		2860	1
2000	1		3072	1
2024	1		4096	172
2040	4		4192	3
2047	6		4196	1
2048	20,287		8192	10

TABLE 3.3: Overview RSA key sizes

3.2 Identification of Weak Keys

There are no keys found that use a bad RNG. There are also no partial similar signatures found between the DSA certificates. This does not fully exclude the fact that the DSA certificates are made without repeated k values but states that these are not found within the obtained data set.

There are weak RSA keys found which have a length of 512 bits. Keys of this length are proven to be factorable to obtain the private key. This is explained in section 3.2.2.

Note that the crawled categories (*Top Grossing*, *Top Selling Free*, *Top Selling New Free*) contain mostly newer APKs, which have a higher probability of being signed using new versions of either OpenSSL or Jarsigner, and therefore may be less likely to contain weak keys. This limits the scope for derivation of private keys since only one type of weak key is found. In contrast, the crawled applications will be more relevant as they are currently being used by a larger audience.

3.2.1 Factoring keys with regard to weak RNGs

Using a setup much like the study by Heninger et al. [9], a database of moduli was made available for the research, as explained in section 2.2.1. The total database size consists of 43.3 million certificates. The common divisor was then calculated for every pair of RSA moduli in the database. No factorable Android key was found which shared common factors.

The same was done for the r and s signature parameters of DSA certificates using the certificate database of the University of Michigan which had approximately 100,000 DSA certificates. No reuse of r was found.

3.2.2 Factoring of RSA keys with small key length

After making a key size inventory of applications in the Google Play store, two 512-bit keys surfaced. The moduli of the found keys are specified in table 3.4. Modulus n_1 was found in two applications, *com.ocmec.tinyclock.widget* and *com.ocmec.tinyclock.widget*, whilst the 155-digit modulus n_2 was found in *com.ninegame.client4*. The latter appears to be quite popular, with tens of thousands of downloads immediately after launch, and a Facebook fan page with more than 100k likes.

Modulus		**Digits**
$n_1 =$	8059795184305920616788640586667583769866873216239418 9812878348892217197219688169810292236873156199323504 4544583411796830432675582188431067275600095695389	RSA-154
$n_2 =$	1004525187687624619402975731107332433915189429952344 9313866406492921871818375886471816716639115289411800 2958812435241112445272019067484351907704355735978l7	RSA-155

TABLE 3.4: Moduli of two 512-bit keys

During the research, the CADO-NFS software was used, which implements the general number field sieve algorithm. CADO-NFS is publicly available and requires only minimal configuration to operate in a clustered environment.

The general number field sieve (GNFS) algorithm is known as the fastest algorithm for factoring general integers larger than 100 digits. Factoring integers generally consists of three main phases of which a concise overview is given below. The GNFS algorithm that comprises these steps is furthermore less relevant to the practical execution of our research.

Polynomial selection

This step tries to find two polynomials $f, g \in \mathbb{Z}$ with common root $m \mod N$. The homogeneous[1] form of these polynomials is: $F_k(a, b) = b^{degree(f_k)} \cdot f_k(a/b)$.

The choice of polynomials can dramatically affect the time to complete the remainder of the algorithm. Many research has been done on the development of better methods [22, 23].

Lattice sieving

Find many integer pairs (a_i, b_i) where both homogeneous polynomial values $|F_k(a_i, b_i)|$ are smooth[2] $(k = 1, 2)$. These pairs are called relations.

Linear algebra

Having enough such relations, the block Wiedemann [24] algorithm is run, which is the last step before the prime factors can be recovered. Different algorithms are available, but block Wiedemann is the algorithm used by CADO-NFS.

The factoring of both our keys took 4 days each. In 1999, this process still took 7 months [4]. In the meantime, the algorithms that are used by the Number Field Sieve have been greatly improved.

[1] Homogeneous polynomial: a polynomial whose nonzero terms all have the same degree
[2] Smooth number: an integer which factors completely into small prime numbers

Table 3.5 gives an idea of the time each step of the process takes. Table 3.6 shows the hardware that was available for the factoring of the keys: roughly 576 cores @ 2.30Ghz. Please note that the last step of the algorithm (Linear Algebra) could currently not be parallelized over multiple machines using CADO-NFS, and ran sequentially using two threads.

Phase	**CPU time**	**Real time**
Polynomial Selection	6 days	2 hours
Lattice Sieving	188 days	<1 day
Linear Algebra	6 days	3 days
Total	200 days	4 days

TABLE 3.5: CPU and real execution time of the different steps for the RSA-155 integer

Hardware	**Nodes**	**Cores**	**Clock speed**
Blade	10 nodes	4 cores HT	1.80Ghz
Twin nodes	14 nodes	8 cores ST	2.50Ghz
DAS-4, UvA	6 nodes	8 cores HT	2.40Ghz
DAS-4, Astron	18 nodes	8 cores HT	2.40Ghz

TABLE 3.6: Clustered CADO-NFS hardware node specifics

The resulting p and q values for the RSA-154 (n_1) and RSA-155 (n_2) modulus:

$p_1 = 9316184841768334197115762199870321487163223774849171206557805000817968584013 1$
$q_1 = 8651390371915447817653032646007602053333645500561767644618425535814217716591 9$
$p_2 = 1048215678960993831818274331192393748830578781110825146276787439225957826593 79$
$q_2 = 9583191778654982114864649236724840573091307741320776685748818772886948302992 3$

TABLE 3.7: Factored p and q primes.

Once the p and q values are retrieved, it is trivial to reconstruct the private key pertaining to the public key.

3.3 Attack vectors

Having reconstructed the private key from the p and q parameters for RSA, it was possible to use that private key to create a valid signature for the APK using Jarsigner. The resulting APK file with new signature got accepted by the Android OS without problems.

The Google Play store application downloads APKs over HTTP. Due to this behaviour, it is possible to mount a man-in-the-middle attack and serve the forged APK file from a different source. The Play store first downloads details of the APK over a secured HTTPS connection, which includes the file size and a SHA-1 hash of the APK. Although it is possible to create a forged APK file with the same file size, it is not possible to create a hash collision for SHA-1 at this time. Serving the original file from a different source will work without problems, since the SHA-1 hash still matches.

Although right now the installation of a forged APK requires some manual intervention, creating an APK with a forged signature (in se: deriving the private key), is an important piece of the puzzle in mounting a viable attack.

Distribution channels similar to the Google Play store may not provide the same hash checks, which could lead to a successful installation of the forged application.

Chapter 4

Conclusions

4.1 Conclusions

Considering that (1) most APKs are using a key length greater than 1024-bits, (2) no weak moduli were detected that could be attributed to a weak random number generator, (3) no reuse of DSA parameters was found, (4) and the private key could be obtained for only two applications, the overall quality of the cryptographic keys is rated as very good. However, there is still an extensive amount of applications that use outdated hashing algorithms, i.e. MD2, MD5, SHA1. These algorithms are considered cryptographically broken even though it is not possible to create colliding hashes at this moment.

The Google Play store does not currently seem to restrict developer certificates to ensure that a secure key length and signature algorithm were chosen. Mounting a man-in-the-middle attack on the Google Play store to replace the original APK with a malicious counterpart that has a valid signature does not work, since extra countermeasures were taken to specifically prevent this type of attack. However, performing the same type of attack may work for different distribution channels that are similar to the Google Play store.

Two RSA-512 certificates were found that were both cracked in little more than one week combined. This proves that, now that RSA-768 has been factored as well, RSA-1024 may not be too far off. The algorithms that are used for the GNFS are constantly being improved and may in the future allow to factor bigger integers of this size. Right now, RSA-1024 is still considered to be secure. The used software, CADO-NFS, is freely available and can be used to factor 512-bit integers by anyone with access to medium computing power.

Appendix A

APK Categories

App Categories	
GAME	BOOKS AND REFERENCE
BUSINESS	COMICS
COMMUNICATION	EDUCATION
ENTERTAINMENT	FINANCE
HEALTH AND FITNESS	LIBRARIES AND DEMO
LIFESTYLE	APP WALLPAPER
MEDIA AND VIDEO	MEDICAL
MUSIC AND AUDIO	NEWS AND MAGAZINES
PHOTOGRAPHY	PRODUCTIVITY
SHOPPING	SOCIAL
SPORTS	TOOLS
TRANSPORTATION	TRAVEL AND LOCAL
WEATHER	PERSONALIZATION
APP WIDGETS	

Table A.1: Crawled APK Categories in Google Play Store

Bibliography

[1] Portio Research. Mobile Applications Futures 2013-2017, 2013. Online report.

[2] Canalys. Top iOS and Android apps largely absent on Windows Phone and BlackBerry 10. Press Release, May 2013. `http://bit.ly/T4jRyX`.

[3] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[4] Stefania Cavallar, Bruce Dodson, Arjen K Lenstra, Walter Lioen, Peter L Montgomery, Brian Murphy, Herman Te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, et al. Factorization of a 512-bit RSA modulus. In *Advances in Cryptology - EUROCRYPT 2000*, pages 1–18. Springer, 2000.

[5] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thomé, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. Cryptology ePrint Archive, Report 2010/006, 2010. `http://eprint.iacr.org/`.

[6] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, 2002.

[7] Vitaly Shmatikov. Overview of Public-Key Cryptography.

[8] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 15–27. ACM, 2009.

[9] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[10] David W Kravitz. Digital signature algorithm, July 27 1993. US Patent 5,231,668.

[11] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, pages 19–35, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25910-4, 978-3-540-25910-7. doi: 10.1007/11426639_2. URL `http://dx.doi.org/10.1007/11426639_2`.

[12] Frédéric Muller. The MD2 Hash Function Is Not One-Way. In PilJoong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 214–229. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23975-8. doi: 10.1007/978-3-540-30539-2_16. URL `http://dx.doi.org/10.1007/978-3-540-30539-2_16`.

[13] Quynh H. Dang. Secure Hash Standard (SHS), 2013. NIST FIPS - 180-4.

[14] Marc Stevens, Arjen K Lenstra, and Benne De Weger. Chosen–prefix collisions for MD5 and applications. *International Journal of Applied Cryptography*, 2(4): 322–359, 2012.

[15] Yu Sasaki and Kazumaro Aoki. Finding preimages in full MD5 faster than exhaustive search. In *Advances in Cryptology – EUROCRYPT 2009*, pages 134–152. Springer, 2009.

[16] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption*, pages 371–388. Springer, 2004.

[17] Søren S Thomsen. An improved preimage attack on MD2. *IACR Cryptology ePrint Archive*, 2008:89, 2008.

[18] Marc Stevens. Cryptanalysis of MD5 and SHA-1.

[19] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2 n work. In *Advances in Cryptology – EUROCRYPT 2005*, pages 474–490. Springer, 2005.

[20] DIGITAL SIGNATURE STANDARD (DSS), 2000. FIPS PUB 186-2.

[21] Elaine B. Barker. DIGITAL SIGNATURE STANDARD (DSS), 2013. NIST FIPS - 186-4.

[22] Thorsten Kleinjung. On polynomial selection for the general number field sieve. *Mathematics of Computation*, 75(256):2037–2047, 2006.

[23] Brian Murphy, Richard P Brent, et al. On quadratic polynomials for the number field sieve. In *CATS*, pages 199–214. Citeseer, 1998.

[24] Don Coppersmith. Solving homogeneous linear equations via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.