RSA Weak Public Keys Available on the Internet

Mihai Barbulescu[1(✉)], Adrian Stratulat[1], Vlad Traista-Popescu[1], and Emil Simion[2]

[1] Computer Science Department, Politehnica University of Bucharest, Bucharest, Romania
mbarbulescu@stud.acs.upb.ro, {adrian.stratluat,vlad.traista}@cti.pub.ro
[2] Faculty of Applied Sciences, Department of Mathematical Models and Methods, Politehnica University of Bucharest, Bucharest, Romania
esimion@fmi.unibuc.ro

Abstract. It is common knowledge that RSA can fail when used with weak random number generators. In this paper we present two algorithms that we used to find vulnerable public keys together with a simple procedure for recovering the private key from a broken public key. Our study focused on finding RSA keys with 512 and 1024 bit length, which are not considered safe, and finding a GCD is relatively fast. One database that we used in our study is made from 42 million public keys discovered when scanning TCP port 443 for raw X.509 certificates, between June 6, 2012 and August 4, 2013. Another database used in the study was made by crawling Github and retrieving the keys used by users to authenticate themselves when pushing to repositories they contribute to. We show that the percentage of broken keys with 512 bits is 3.7 %, while the percentage of broken keys with 1024 bits is 0.05 %. The smaller value is due to the fact that factorization of large numbers includes new prime numbers, unused in the small keys.

Keywords: RSA · Public keys · Weakness · Vulnerabilities · GCD · Euclid · Internet · Common factor

1 Introduction

Generating proper random numbers is essential in nowadays cryptography. Random number generation has been long studied from both practical and theoretical perspectives [15,17] and vulnerabilities were found due to bad implementation (e.g.: using `srand(time(NULL))` in C for seeding). Also another important fact of the RSA key is it's length. In history we can denote the following milestones of RSA factorization:

– In 2000, a 512-bit RSA number, having 155 digits, was factored using the Number Field Sieve factoring method, same method that was used in the previous record, from 1999, to factor a 140 digit RSA modulus [14].

I. Bica and R. Reyhanitabar (Eds.): SECITC 2016, LNCS 10006, pp. 92–102, 2016.
DOI: 10.1007/978-3-319-47238-6_6

- Between 2006 and 2008, Linux distributions Debian and Ubuntu had a bug in which less than 220 possible keys for SSH, OpenVPN etc. were possible to generate. Instead of mixing in random data for the initial seed, the only "random" value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32768, resulting in a very small number of seed values being used for all pseudo-random number generation operations (see [8]).
- On the 12th December 2009 a study reports factorization of 768-bit RSA and claims that factorization of 1024-bit RSA key is considered 1000 times harder [19].

Multiple approaches were done in order to find out how severe and how often can a RSA vulnerability occur. For instance in [20] it was found only an order of 0.003 % of insecure public keys (which have a common factor) from data provided by EFF SSL [5] in November 2001 containing 6185372 distinct X.509 certificates having multiple RSA key lengths. The main goal of the project was testing the validity of the assumption that different random choices are made each time keys are generated.

Another approach [18] is a large-scale study of RSA and DSA keys, focusing on keys which are used in TLS (HTTPS) and SSH in which 5.57 % TLS hosts and 9.60 % SSH hosts shared keys in a vulnerable matter, from a total number of 5.8 million unique TLS certificates from 12.8 million hosts and 6.2 million unique SSH host keys from 10.2 million hosts.

The approach in our paper was focusing on consequences of RSA issues that someone might find with enough super-computing power and experiment with various GCD implementations, using existing databases of RSA keys such as continuous scan of HTTPS Ecosystem between 2012 and 2013 [16] or dataset done by EFF SSL Observatory [5] in 2010.

The first analysis in our study was a sanity check session on 512-bit and 1024-bit RSA public keys from amongst 43 million unique certificates dumped from a regular and continuous scan of HTTPS Ecosystem between 2012 and 2013 Sects. 2 and 3 will describe this problems and how a simple `nmap` on port 443 can be done to obtain a certificate. This shows a simple Linux userspace approach to extract X.509 certificates that was used in [16]. These keys are considered the most vulnerable, that even ransomware viruses choose 2048-bit RSA length for their keys. Also, the default length used in OpenSSH for RSA key generation is 2048-bit. Section 5 will describe more of our results, using multiple common divisor approaches.

The next focus in our study was to find if there are vulnerable Github public keys or not. Many Github users usually use OpenSSH in Linux (command `ssh-keygen`) or Putty to generate their pair of public/private keys and upload the public key on Github. By using a simple HTTP Request to Github API one can extremely easily retrieve the SSH public keys of an user by using either a link like https://github.com/torvalds.keys or https://api.github.com/users/torvalds/keys.

About 97.7 % public keys on Github are `ssh-rsa`, while the rest of them are `ssh-dsa`. A similar effort was done by Cryptosense company. Their focus was on 2048-bit RSA keys (the most common amongst Github users), as these are 93.3 % from all the keys and only 4.2 % are 1024-bit length. In June 2015 from all Github keys there were also public keys with major vulnerability due to length: 2 keys with 256 bits to them and 7 that have 512 bit [1]. While crawling on Github, we did not manage to find these keys so the users might have got the warning and managed to retract them in time. Section 4 will detail our procedure to scan Github keys. The study from Cryptosense used an implementation of GMP-ECM (Elliptic Curve Method for Integer Factorization) [12] but there is no clear disclosure of their results [10].

In 2013, it was reported that an attacker can efficiently factor 184 distinct RSA keys out of more than two million 1024-bit RSA keys downloaded from Taiwan's national *Citizen Digital Certificate* database. The Ministry of Interior Certificate Authority (MOICA) from Taiwan confirmed that these keys were generated, using a low-quality hardware random number generator, by Renesas HD65145C1 chips inside Chunghwa Telecom HICOS PKI Smart Card and also no run-time sanity check was performed. [13] That is why, in Sect. 3 we describe briefly how we took a look at Estonia Electronic ID.

Lastly, another focus in our research was the ransomware virus. Ransomware represents the mechanism through which a hacker locks a resource owned by a user and demands a ransom in return for unlocking that resource. The resource locking is usually done through encryption. A cryptographic ransomware is capable of encrypting an entire filesystem using AES and then encrypt the AES password using RSA. Usually these viruses do not store the RSA public key on the victim's computer due to the known facts about RSA problems that they might have.

2 Background

2.1 Scanning for X.509 Certificates

A potential methodology for scanning HTTPS TCP port 443 in Linux can be described as follows:

– discover hosts with HTTPS (443) port activated. One can easily achieve this by using `nmap` command, similar to the following execution:

```
u@linux: ~ $ nmap --script=ssl-cert.nse -p 443 www.google.com
```

– completing a TLS handshake with responsive addresses and collecting the presented certificate chains. This can be achieved in Linux command line by using the `openssl` suite:

```
u@linux: ~ $ openssl s_client -crlf -connect www.example.net:443
```

– parse and validate certificate. A full C example of how this can be done using OpenSSL library is described in [11].

2.2 RSA Background

RSA is one of the most well known and most used asymmetric cryptographic algorithm which uses two keys for the encryption and decryption process: a public key and a private key. The public key is represented by an exponent e and by a modulus N. The modulus is computed as the product of two randomly private generated prime numbers p and q. The private key d can be computed using the following formula:

$$d = e^{-}1 \mod (p-1)(q-1)$$

Since p and q are unknown the best way to calculate the private key is to factor the modulus N and obtain the two prime numbers. However, this kind of attack can be unfeasible given a certain RSA key length. A better approach is to try to find if the moduli from multiple RSA public keys have a common factor.

2.3 GCD Algorithms

For running the initial sanity check session on 512 and 1024 bit length RSA keys we used the C language with the OpenMP support for easy multi-threading enablement in order to use at maximum an AMD multi-core architecture we had. Because C does not have built-in support for big numbers, which was a requirement for our application, we used an arbitrary precision (bignum) library.

We decided to use GMP (GNU Multiple Precision Arithmetic Library) [6], as it has support for integer and rational numbers, can do computations in finite fields, aiming at speed and supporting numerical algorithms such as greatest common divisor, extended euclidean algorithm for inverse modulo n and other useful cryptographic computations.

The brute-force approach to find the prime factors of a number n is to check against all the prime numbers in the interval $[2, \sqrt{n}]$. Because this is not feasible for big numbers (larger than 2^{100}), another approach has to be chosen, such as batch GCD.

The approach we used was to compute the GCD using Euclid's algorithm on all the possible pairs in a set of numbers. This way, instead of storing a large database of prime numbers, we only store the set of numbers to be checked.

```
for i = 0 to m-1
    for j = i to m
        t = gcd(A[i], A[j])$
        if t != 1 and t !=  A[i]$
            print i:A[i]:t
            print j:A[j]:t
```

The idea behind batch GCD is very simple: Given a sequence X of positive integers, the algorithm computes the sequence

- $gcd(X_0,\ X_1 \cdot X_2 \cdot X_3 \ldots)$
- $gcd(X_1,\ X_0 \cdot X_2 \cdot X_3 \cdot \ldots)$
- $gcd(X_2,\ X_0 \cdot X_1 \cdot X_3 \cdot \ldots)$
- etc. ...

It shows which integers share primes with other integers in the sequence. Because one only wants to know if a key is compromised, not with which key has a common divisor. The initial development of algorithm was done in [3]. The algorithm can be described using the following steps

- Input: $N_1, \ldots, N_m$ RSA public keys
- Compute: $P = \prod_{i=1}^{m} N_i$ (use product tree)
- Compute $z_i = (P \mod N_i^2), \forall i = 1, \ldots, m$ (use remainder tree)
- Output: $gcd(N_i, z_i/N_i), \forall i = 1, \ldots, m$

The final output is the GCD of each modulus N_i with the product of all the other N. Interest is in those for which this GCD is not 1.

2.4 Ransomware

The ransomware techniques can be classified into two categories: locker ransomware and crypto ransomware.

Locker ransomware denies access to computing resources by usually locking the device's user interface. It then asks the user for a ransom in order to restore access. In general the user interface will contain only the ransomware interface through which he will make the payment. Access to the mouse is disabled and access is granted only to the numerical keys on the keyboard. Locker ransomware just locks the access to a system, it does not modify anything in the system (filesystem data). This type of ransomware is among the least destructive types since it can be removed cleanly without affecting the system, by using various tools provided by security vendors.

Crypto ransomware is the most destructive type of ransomware. It is capable of encrypting data on a device through an encryption process. It usually runs under the radar, it tries to search and encrypt as much as files as possible notifying the user and demanding a ransom in return afterwards. The user can regain access to his data only if he pays the ransom or if the user is capable of computing the decryption key necessary to decrypt the ransomed data.

The modern cryptographic ransomware techniques usually use both symmetric and asymmetric cryptographic algorithms. A symmetric algorithm uses the same key for the encryption and the decryption process. This key can be either generated locally (on the infected device) and sent back to the attacker or it can be generated by the attacker (C&C server). An important observation is that after the files were encrypted this key needs to be erased from the user's system since it can be tracked and used to decrypt the files. The advantage in using a

symmetric encryption algorithm is that it is faster than an asymmetric encryption algorithm. Depending on how many files the ransomware tool encrypts, the encryption process can take a significant amount of time. Using a symmetric key can boost the speed of the encryption process and prevent the user from detecting on time that files are being encrypted.

An asymmetric algorithm uses a pair of keys: a public one for data encryption and a private one for data decryption. In ransomware techniques the public key is used to encrypt the files whereas the private key is held by the C&C server and will be used once the ransom is paid by the infected user. Having the public key stored on the infected device does not generally affect the security of the key pair used for ransom. A significant drawback of this algorithm is that it is slow and it can expose the encryption process to the user.

Depending on where the cryptographic keys are stored there are multiple ransomware families:

- downloaded public key - the files are encrypted with an AES symmetric key that is generated on the infected device. The symmetric key is encrypted with a public key that is downloaded from the C&C server. The encrypted symmetric key is stored in each encrypted file and cannot be decrypted since the private key is held by the server. A significant drawback for this method is that if the C&C server cannot be accessed because of a firewall or because of having no internet connection then this ransomware attack will fail. An example of a ransomware virus that behaves this way is Trojan.Cryptodefense.
- embedded public key - the ransomware virus includes an embedded RSA public key which will be used to encrypt a locally generated AES symmetric key. The advantage of this method is that there is no need to contact the C&C server. The drawback is that the ransomware virus needs to have a different public key every time it infects a device. If it is not different then once the private key has been determined the ransomware virus will become obsolete. An example of such a virus is CTBlocker.
- embedded symmetric key - the ransomware virus includes an embedded AES symmetric key which will be used to directly encrypt the files. There are no asymmetric keys used in this technique. The advantage is that the virus does not have to contact the C&C server, but the weakness is that once the secret key has been determined all the files can be decrypted. An example of a virus from this family is represented by Android.Simplelocker, a virus for Android mobile devices.

User devices usually end up being infected with ransomware viruses through unscanned downloads from spam e-mails, from exploit kits, bot infections and even from social engineering attacks. [7]

3 Mining After Public Keys

3.1 Extracting Github Keys

Previous attempts, such as the one performed by Cryptosense company [10] used OCaml to implement batch GCD, but no disclosure of how Github API

was used to extract the public keys. It is important to note that Github API only shows information of users that exist and does not include the users whose accounts have been deleted or IDs of private organizations. Listed users obtained after a HTTP request to Github API can be of type *User* or type *Organization*. Organizations are also regular Github users with some particularities.

In our approach, we developed a method to extract keys using Python and HTTP requests to Github API. The first issue we ran into was that Github has rate limiting for API queries, allowing only 60 HTTP requests per hour for unregistered scripts. We have generated a token so that we were able to make 5000 calls per hour.

Another lesson learned while crawling the keys was that instead of using Github API to extract a user's public keys, using a HTTP request to
https://api.github.com/users/torvalds/keys
we found that we could do a simple HTTP request to
https://github.com/torvalds.keys
which did not cost us any API calls, and in 1 h we were able to process more users and make timeouts smaller.

For extracting the public keys we just estimated the total number of Github users (a statistic done by Prajan Mittal determined 10492402 valid accounts in 11 January 2015 [2]) and at each iteration retained the last valid ID of user and get the next 30 registered users, as there is no way to list all the Github users using only one HTTP request. The only accepted method is listing a chunk of users by querying https://api.github.com/users?since=111. Using this method we can list all the users, in the order that they signed up on Github and pagination is powered exclusively by the `since` parameter - this parameter expects a valid ID number.

Because of the timeouts after 5000 hits due to Github API rate limiting and because of the low computing powers required (all we needed was a hard drive and a computer connected to Internet), we did this key mining on a Raspberry PI platform connected via USB to a hard- drive with external 5.1 V DC input voltage.

3.2 Extracting Estonia Certificates

Estonia uses a nation-wide database to store the citizen's identification data and cryptographic certificates, which can be queried using LDAP. The certificates store 1024-bit long RSA public keys. To protect against crawlers, they limit the number of queries a host can do in a certain time-frame, and limit the possible LDAP queries to two types: general queries (returning a maximum of 50 identities at a time) and targeted queries (assuming the personal ID number is known).

To crawl this database beyond the 50 initial identities, we had to generate queries with valid ID numbers. The Estonian ID numbers can be easily brute-forced, as they contain seven digits for the date of birth and gender information, three digits as serial numbers and one checksum digit. To get the certificates of every citizen born in the same day, only 2000 queries are needed.

To do such a query, the following command is used:

```
ldapsearch -x -h ldap.sk.ee -b c=EE "(serialNumber=$ID_NUMBER)"
```

Unfortunately, after the first hundred of requests, time-based restrictions kick in, blocking further requests until a timeout expires. Among the gathered certificates, no weak keys were found.

3.3 Ransomware

In early 2015 a ransomware virus named SleeperLocker has silently infected the workstations of thousands of employees, but it hadn't triggered at all until the midnight of 25th of May 2015. According to [9] a possible source for the ransomware spread was a corrupted installer of the game Minecraft.

The locker uses Windows services to encrypt using an RSA key files with different extensions (.doc, .docx, .jpg, etc.). It does not change the file extensions since the operating system would notify the user of the appearance of corrupted files. Apparently, the locker will terminate if it detects that the system it was installed on is a virtual machine. Also, it deletes the volume copies from `C:\shadow` which contains snapshots of the C drive at certain moments of time. In order to have its files decrypted, the user had to pay 0.1 bitcoins.

The unthinkable happened on 30 May 2015. Apparently the author of the locker ransomware apologized for what his tool has caused and uploaded a database containing bitcoin addresses, public keys and private keys. Afterwards, on the 2nd of June the author issued a command to have the locker ransomware decrypt all files.

We managed to find the database dump on [4]. This dump was written in an XML format used in .NET applications. As a matter of fact, according to a post belonging to the author of the ransomware all the RSA key-pairs were generated using the `RSACryptoServiceProvider` class from the `.NET` framework and all the AES keys were generated using the `RijndaelManaged` class.

The database has 62703 rows and each row of the database contains its data encoded in the `base64` format. The data contains the following information:

- the public key - represented by the moduli N and the public exponent e
- the private key - represented by the prime numbers p and q whose product gives N. It also contains the values of dP, dQ and $Q^{-}1$. These keys contain the necessary elements that can be used in Chinese Remainder's Theorem for decrypting the private key. Lastly the row also contains the private exponent d.

All the generated keys have a 2048 length. An interesting observation is that all the keys have the same public exponent $AQAB$ in BASE64 format or 65537 in decimal format. This exponent is the standard one used because it is a compromise between being a high enough number in order for the key to be secure and the computational cost of performing an exponentiation. Another reason is due to it being a Fermat prime number which makes exponentiation a lot faster (Table 1).

4 Scenarios and Results

Table 2 shows the results extracted from database provided by [16] which contained a total number of 44474713 keys. The results from the 512-bit length keys was done using the naive approach (to demonstrate how weak 512-bit RSA is) by computing all-pairs GCD using Euclid's algorithm. Using an AMD quad-core x86_64 CPU, running at 3.9 GHz, with 6 GB RAM we were able to perform $720k$ GCD computations per second for 512-bit length RSA. We also used this approach for some of the 1024-bit length RSA keys using two approaches: exhaustive search for matches on a set of $100k$ keys (phase I) and trying to match the 2 divisors from the previous set against the full dataset (phase II). The two phases from naive approach took 48 h for 1024-bit RSA and about 8 h for 512-bit RSA.

Table 1. Results of RSA keys from 2012–2013 scan of X.509 certificates

Len/Ph	Total keys	Pairs	GCDs	Broken
512-bit	323338	52273246116	4717	12209 (3.7 %)
1024 (ph I)	100000	4999850001	2	6 (0.0006 %)
1024 (ph II)	26177420	53738048	6806	13617 (0.05 %)

The third approach (phase III) on 1024-bit RSA was to use the fast GCD implementation done by [18]. Because of the limited amount of RAM of our systems we broke the 26177420 (which is 60 %) total number of 1024-bit keys from the dataset in chunks of 800000, thus comparing a key with the product of the other 799999 keys, and used 8 threads. Using this approach computation took only 18944.7 s. In this third approach there was no pairs approach. Out of 26177420 keys tested, about 0.25 % (meaning 63502) keys were found to be broken.

During two weeks of Github crawling between 22 December 2015 and 7 January 2016 we managed to discover that only 26 % of the users we processed (approximatively 3 million Github users) had public SSH-RSA keys configured. 1 key was 512-bit length and only 12 keys were 16384-bit length. 0.51 % were 1024-bit length,

The single 512-bit RSA key discovered through Github crawling was ran against our set of databases and was found to be broken. For the other lengths, by comparing keys between them, no vulnerability was found. It is needed now a smarter method to compare the 1024 and 2048 bit lengths with databases available.

Regarding Estonia LDAP with RSA IDs a big limitation was the restrictions on the number of queries. Thus we were not able to extract a relevant number of keys to find vulnerabilities.

Overall, the generated public keys for ransomware virus from [4] seemed to be secure due to their length (2048-bit). Comparing the keys between them, the

Table 2. Results of Github scanned keys

Len	Percent keys
512	1 key
1024	0.51 %
2048	55,5 %
4096	3 %
8192	0.01 %
Other	41 %

entropy did not raise any concern, as no vulnerability was discovered by any of our GCD approaches.

5 Conclusion and Further Work

The results and facts presented in this paper should discourage the use of RSA keys having lengths less or equal to 1024 bits and force readers to use at least 2048-bit long keys, pay more attention to random number generators in their system (if they used Debian or derivates in 2008–2009 to generate RSA keys, they should re-generate a new pair and revocate the keys that might be compromised). Multiple online tools such as the ones by [10] have been developed for fast, local sanity checks, of freshly-generated RSA keys, but this is not enough. Users should be aware that, when using RSA, there is always a hacker with enough computing power and patience crawling for public keys and searching for vulnerabilities.

Acknowledgments. This work partially supported by the Romanian National Authority for Scientific Research (CNCSUEFISCDI) under the project PN-II-PT-PCCA-2013-4-1651.

References

1. Cox, B.: Auditing Github Users' SSH Key Quality. https://blog.benjojo.co.uk/post/auditing-github-users-keys
2. Cryptosense - Batch-GCDing Github SSH Keys. https://cryptosense.com/batch-gcding-github-ssh-keys/
3. Bernstein, D.J., Heninger, N., Lange, T.: FACTHACKS - RSA Factorization in the Real World. http://facthacks.cr.yp.to/batchgcd.html
4. Database with Ransomware Public Keys (from the author of the virus). https://archive.org/download/locker-ransomware-database-dump
5. Eckersley, P., Burns, J.: An observatory for the SSLiverse. Talk at Defcon 18 (2010). https://www.eff.org/files/DefconSSLiverse.pdf
6. GMP Library. https://gmplib.org/. Accessed 11 June 2015
7. Savage, K., Coogan, P., Lau, H.: The evolution of ransomware. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-evolution-of-ransomware.pdf

8. Bello, L.: Bug DSA-1571-1 OpenSSL Predictable Random Number Generator. http://www.debian.org/security/2008/dsa-1571
9. Sjouwerman, S.: Is Your Network Infected with Sleeper Ransomware? https://blog.knowbe4.com/is-your-network-infected-with-sleeper-ransomware
10. Total Number of Github User Accounts. http://tech.pranjalmittal.in/blog/2015/01/10/github-api-calculating-total-users-on-github/
11. Durumeric, Z.: Certificate Parsing with OpenSSL and C. https://zakird.com/2013/10/13/certificate-parsing-with-openssl/
12. Zimmerman, P., et al.: GMP-ECM (Elliptic Curve Method for Integer Factorization). https://gforge.inria.fr/projects/ecm/
13. Bernstein, D.J., Chang, Y.-A., Cheng, C.-M., Chou, L.-P., Heninger, N., Lange, T., van Someren, N.: Factoring RSA keys from certified smart cards: Coppersmith in the wild. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 341–360. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-42045-0_18
14. Cavallar, S., et al.: Factorization of a 512-bit RSA modulus. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 1–18. Springer, Heidelberg (2000). http://dl.acm.org/citation.cfm?id=1756169.1756171
15. Dorrendorf, L., Gutterman, Z., Pinkas, B.: Cryptanalysis of the windows random number generator. In: Proceedings of 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 476–485. ACM, New York (2007). http://doi.acm.org/10.1145/1315245.1315304
16. Durumeric, Z., Kasten, J., Bailey, M., Halderman, J.A.: Analysis of the HTTPS certificate ecosystem. In: Proceedings of 13th Internet Measurement Conference, October 2013
17. Gutmann, P.: Software generation of practically strong random numbers. In: Proceedings of 7th USENIX Security Symposium, San Antonio, Texas, 26–29 January 1998. USENIX, New York (1998)
18. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: detection of widespread weak keys in network devices. In: Proceedings of 21st USENIX Security Symposium, August 2012
19. Kleinjung, T., et al.: Factorization of a 768-Bit RSA modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer, Heidelberg (2010). http://dl.acm.org/citation.cfm?id=1881412.1881436
20. Lenstra, A.K., Hughes, J.P., Augier, M., Kleinjung, T., Wachter, C.: Ron was wrong, Whit is right. Technical report (2012)