Ruhr-Universität Bochum
Lehrstuhl für Netz- und Datensicherheit
Master-Seminar Netz- und Datensicherheit
Wintersemester 2012/13

Mining Your P's and Q's: Detection of Widespread Weak Keys in Network Devices

Christian Schlipp
christian.schlipp@ruhr-uni-bochum.de
February 12, 2013

Contents

1 Introduction

It is well known fact that proper randomness is a very important criteria in modern cryptography. Most of the cryptographic systems currently in use need at least in one method random input, for example for key generation during the initial setup. If the source for this entropy is faulty, it may result in a catastrophic failure of the whole system. This could be recently observed with the discovery of the weak keys in Debian distributions [3].

This thesis will show and discuss the results of two independent papers published in 2012 concerning random number generators:

- **Mining your P's and Q's: Detection of Widespread Weak Keys in Network Devices:**
  This Paper has been published by Heninger (University of California, San Diego), Durumeric (University of Michigan) et al. in August 2012 in the proceedings of the 21st USENIX Security Symposium.
- **Ron was wrong, Whit is right:**
  The second paper has been published by Lenstra (EPFL Lausanne), Huges (Palo Alto, CA) et al. in February 2012.

We will start with a short overview over the used algorithms and the different collections of data used by the two teams. We then continue to examine the methodology of the different approaches and show the results. Afterwards, we will discuss if these studies have a severe impact on certificates or RSA/DSA cryptosystems in general. Chapter 6 will present some lessons to be learned in order to prevent these failures. Finally, we will draw a conclusion.

2 Background

In this chapter we would like to introduce the used algorithms and the collection of data used by the different teams.
It is important to know the inner proceedings of these cryptographic algorithms in order to understand the results of these papers. Furthermore we will point out the differences in the used data collection, as it may be of interest concerning the results explained in Chapter 4.

2.1 RSA

The RSA cryptosystem, published in 1977 by Rivest, Shamir, and Adleman, is one of the most commonly used asymmetric algorithms in cryptography.[5] It is used in the common TLS protocol. The *RSA problem* demands whether it is possible to obtain d without being able to factorize N. The security is depends on the fact that is is hard to factorize large numbers.[5] Currently there is no known algorithm for the factorization of large numbers is a practical amount of time. RSA can be used either for encryption and decryption of data or for signing and verifying of data. The keys depend on the mode of operation. This section focuses on RSA signatures. The initial setup of the RSA parameters is described in Figure 1.

1. Choose $N = p * q$, whereas $p, q \in_R P$ $(prime)$.
2. Choose e, whereas $1 < e < \phi(N)$ and $gcd(e, \phi(N)) = 1$ $(e, \phi(N)\, coprime)$.
3. Determine $d = e^{-1}$ $(mod\ \phi(N))$ $(multiplicative\ inverse)$.

This results in $d * e = 1$ $(mod\ \phi(N))$

Figure 1: RSA cryptosystem: Key Generation.

The public key k_{pub} consists of (N, e), the private key k_{priv} of (N, d).
A data block can be signed as follows:

$$s = m^d\ (mod\ N)$$

Figure 2: RSA cryptosystem: Signing a message.

The verification in RSA is computed as follows:

$$m_s = s^e\ (mod\ N)$$

Check if $m = m_s$, then valid.

Figure 3: RSA cryptosystem: Verifying a message.

A known vulnerability of RSA is the fact that even though it is hard to factorize the modulus, it is very easy to find the greatest common divisor of two moduli. If one is able to find two RSA moduli sharing at least one prime factor, the other one can be easily calculated. This finally allows an attacker to compute the private key as shown in Figure 1.

2.2 DSA

The *Digital Signature Algorithm* has been proposed by the NIST in August 1991. It is based on the *discrete logarithm problem*, saying that is is hard to calculate the correct logarithm of a number in a congruence class under multiplication modulo N.[5] The initial key setup is described in Figure 4.

1. Choose length L and N, whereas $L > N$.
2. Choose $q,\ |q| = N, q \in_R P\ (prime)$.
3. Choose $p,\ |p| = L, p \in_R P\ (prime), p-1$ is a multiple of q.
4. Determine $g = h^{\frac{p-1}{q}}$.
5. Choose random $x,\ 1 < x < q$.
6. Determine $y = g^x\ mod\ p$.

Figure 4: DSA cryptosystem: Key Generation.

The public key k_{pub} consists of (p, q, g, y), (x) is the private key k_{priv}.
Using DSA, data can be signed using the following algorithm:

1. Choose random $s,\ 1 < s < q$ as ephemeral key.
2. Determine $s_1 = ((g^s\ (mod\ p)\ mod\ q)$. If $s_1 = 0$ restart at step 1.
3. Determine $s_2 = s^{-1} * (m + s_1 * x)\ (mod\ q)$. If $s_2 = 0$ restart at step 1.

Figure 5: DSA cryptosystem: Signing.

The signature for m is (s_1, s_2). The ephemeral key s has to be kept secret and only used once.
A DSA signature can be verified by calculating the term shown in Figure 6.

1. Determine $w = s_2^{-1}\ (mod\ q)$.
2. Determine $v = (g^{m*w\ mod\ q} * y^{s_1*w\ mod\ q}\ mod\ p)\ mod\ q$.
3. Check if $v = s_1$, then valid.

Figure 6: DSA cryptosystem: Verifying.

According to the previously mentioned steps, the private key can be computed if the ephemeral key s is used in more than one message. Therefore the steps shown in Figure 7 have to be performed.

1. $s = (m_1 - m_2)(s_{2,1} - s_{2,2})^{-1} \ (mod\ q)$
2. $x = s_1^{-1}(s * s_2 - m) \ (mod\ q)$

Figure 7: DSA cryptosystem: Retrieving the private key using two messages with the same ephemeral key.

As we can see, it is essential for both named cryptosystems to use proper randomness. Otherwise both systems are prone to failure and should be considered as insecure.

2.3 Data Collection

Both teams performed their research on different databases. This chapter explains the underlying data set and the differences between the two teams approach.

2.3.1 Heninger et al.

This team performed an internet-wide scan of all SSH and TLS hosts reachable. Therefore they developed a three-stages approach:

1. During the first stage the team wanted to discover all SSH (port 22) and TLS (port 443) hosts currently reachable in the whole IPv4 address range. Using Nmap 5 [1], this search completed on 25 Amazon EC2 micro instances of 5 regions in 25 hours.

2. In the second stage, all hosts discovered in the first step were contacted in order to retrieve a certificate. It resulted in 5.8 million TLS and 6.2 million SSH unique certificates collected by Python tools and a simple SSH client written in C on EC2 large instance servers. All hosts offering a DSA key during SSH scan were later on contacted in two runs by hosts at UCSD and the university of Michigan.

3. For TLS, the X.509 fields of the certificate were parsed into a database by a Python and C script.

After these steps, the team had a database of all available certificates at this time. In Figure 8 the results of their own scan is compared to the scan of the SSL Observatory of December 2010 in response to the Debian weak keys.

[1] `http://nmap.org/5/`

	SSL Observatory (12/2010)	Heninger's TLS scan (10/2011)	Heninger's SSH scan (2-4/2012)
Hosts with open port 443 or 22	16,200,000	28,923,800	23,237,081
Completed protocol handshakes	7,704,837	12,828,613	10,216,363
Distinct RSA public keys	3,933,366	5,656,519	3,821,639
Distinct DSA public keys	1,906	6,241	2,789,662
Distinct TLS certificates	4,021,766	5,847,957	-
Trusted by major browsers	1,455,391	1,956,267	-

Figure 8: Comparison of the results of the SSL Observatory scans and Heininger's et al. scans. Taken from [1].

2.3.2 Lenstra et al.

Lenstra's research relies mainly on the data collected by the SSL Observatory mentioned in the previous section. Before this data was publicly available, they started collecting keys "from a wide variety of sources, assisted by colleagues and students" [2]. In Appendix A of [2], the newly generated data set of the SSL Observatory as of February 2012 is added to the previously made research. Both data sets were not merged, but observed independently. Other than Heninger's team, they did not engage in crawling the web for certificates themselves. Furthermore, they used public available PGP keys.

The list of certificates has been reduced by their chosen criteria, namely an expiration date later than 2011 and the use of SHA 1 or better as a hash algorithm. 33.4% of the provided certificates fulfilled both criteria and have been analyzed in the following process. We can guess that certificates with an expiration date before 2011 and MD5 has algorithms were considered as expired and/or insecure.

3 Methodology

This section will explain the different methods used to analyze the data.
Due to the vast amount of certificates gathered by Heninger's team, a manual inspection was impossible. As a normal approach of calculating the gcd of every 2 moduli would take 30 years, they implemented Bernstein's algorithm for fast gcd computation [4] in almost linear time. Therefore, all distinct moduli are arranged in a binary product tree to calculate $\prod(N_i), N_i \in \{N_1, ..., N_m\}, \forall i \in 1, ..., m$, whereas $N_i = p_i * q_i, p_i, q_i prime$. Using a binary remainder tree, the gcd of a modulus with

the product of all other moduli is calculated. One exemplary tree is shown in Figure 9 and 10.

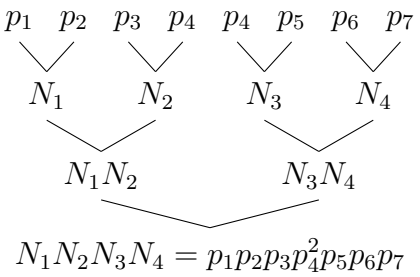

Figure 9: Bernstein Algorithm: Multiplication Tree.

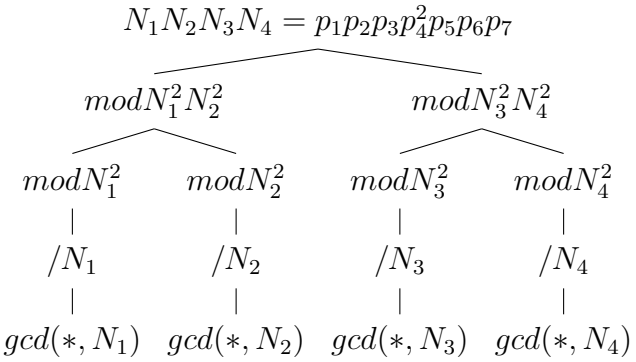

Figure 10: Bernstein Algorithm: Remainder Tree.

As a result, the gcd for each known Modulus $[N_i]$ with the product of all other moduli (and their primes) $[*]$ is being calculated. See Appendix A [9] for an example.

In case a modulus shares both primes, it has to be factorized manually as the algorithm would result in the modulus itself. This computation finished on a Core i5 with 32GB of RAM in 5.5 hours or on a EC2 Compute Eight Extra Large Instance in about 1.3 hours at a total cost of 5$ [1].
The classification was mainly possible due to the X.509 fields of a TLS certificate. Nmap host detection or an analysis of the other IP services were used in case the X.509 data was insufficient. For SSH hosts, IP fingerprinting and an analysis of the information available through HTTP and HTTPS has been used, as these certificates do not include any descriptive data.

These information is used to divide the whole data set into groups according to their vulnerability. The main groups used are:

- Repeated Keys.
- Default Keys.
- Repeated Keys due to low Entropy.
- Factorable RSA Keys.
- DSA Signature Weakness.

Lenstra's team, in contrast, mainly focuses on the correctness of keys instead of the causes. They do not divide their data set into different groups, but instead analyze them under different aspects. For each cryptosystem RSA, DSA, and ElGamal, they examine:

- ... the number of correct keys/moduli according to the definition of the cryptosystem.
- ... the number of shared keys/moduli among all other keys of the same cryptosystem.
- ... the key/moduli sizes.
- ... the shared primes and generators.

Further details on their research have not been made public.

4 Results

This chapter gives an overview of the results of the two papers. As both teams draw different conclusions out of their work, a discussion about this will follow in Chapter 5.

4.1 Heninger et al.

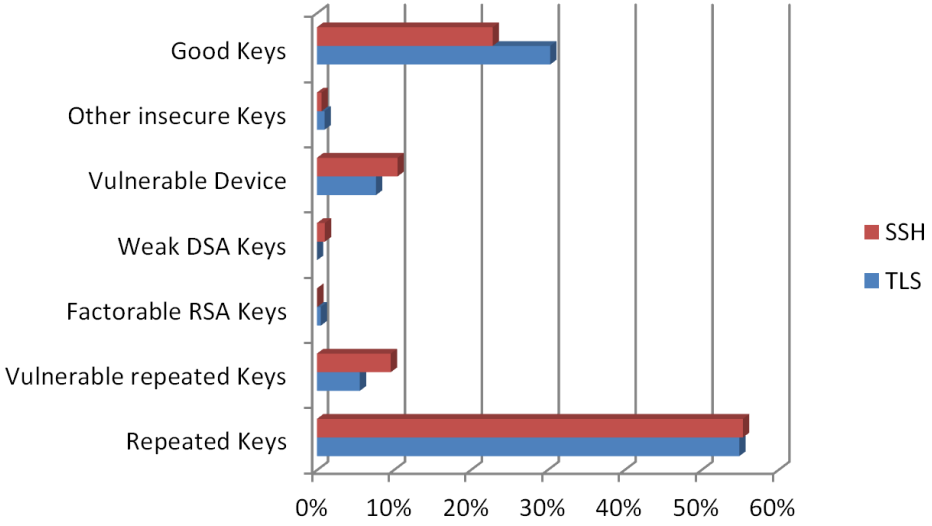

Figure 11: Results of Heninger et.al.

Repeated Keys Lenstra's team found out that 61% (7,770,232) of all TLS hosts and 65% (6,642,222) of all SSH host delivered the same public key as at least one other host. Most of these duplicated keys occurred due to shared hosting or different hosts within one organization. These results were excluded from further inspection, as this behavior is not critical and commonly used. Nevertheless, 5,57% of all TLS hosts had vulnerable repeated keys. SSH host are even worse, 9.6% (981,166) of all scanned SSH hosts used a repeated key.

Default Keys Most of the remaining duplicated keys could be classified as default keys set by the manufacturer. 5.23% or 714,243 of all TLS hosts served a default certificate. The according private keys to these default certificates can be found out either by reverse engineering these models or simply by using databases like littleblackbox.

Repeated Keys due to low Entropy 0.34% of all TLS hosts served repeated keys due to low entropy, a few of of them were even certified by a Certification Authority. Heninger's team was unable to distinguish SSH default keys from repeated keys due to low entropy. As devices offering a repeated key once should not be trusted, they extrapolated these device models to all known hosts, resulting in 2.45% of all hosts serving a potentially repeated key. The Debian weak keys were excluded of this analysis.

Factorable RSA Keys It was possible to factorize 0,5% (64,081) of all gathered RSA keys used in TLS and 0,03% (2,459) of all RSA keys used in SSH using the method described in this section. That means that in all these cases the private key could be calculated. At least, none of these keys were signed by a trusted CA. 99% of these factorisable keys could be assigned to 41 manufacturers, whereas an "IBM Remote Supervisor Adapter" has to be mentioned especially. This device generated RSA keys out of 9 fixed primes distributed uniformly across all 36 possible moduli.

DSA Signature Weakness 4,365 (0.05%) of the DSA signatures could be compromised using the the weakness described in Figure 7 on page 6. Most of these repeated uses of the ephemeral key were caused by two devices.

4.2 Lenstra et al.

Lenstra's team first focused on the different moduli observed in their data set, of which 4.3% (266,729) of the X.509 certificates are shared with at least one other. They point out that many of these repeated moduli are reused variables by the same owner, but that there are also repeated moduli of obviously unrelated owners. Two parties were considered as 'unrelated' if the additional data of a certificate does not lead to other assumptions. The gathered PGP certificates have been examined independently, showing only 28 of 397,621 moduli occurring more than once. 21,479 distinct moduli of X.509 and PGP certificates are affected by shared factors. Most of the RSA moduli are ≥ 1024 bits, as expected, only a few of the available certificated were incorrect (e.g. non-prime p and/or q used).

As with RSA, most of the given ElGamal keys are correct and $\geq$ 1024bits. There are only 876,202 (34.40%) distinct values of p in 2,546,628 ElGamal keys, which is quite surprisingly. Nevertheless, the fact that only 93 distinct values for p occur more than once is even more surprisingly. They suspect these values to be generated by the same software using an identical random seed or no randomness at all.

Most of the DSA keys passed the tests for correctness as well. The most frequent size for q is 160 bits and almost every key is unique. The primes used in the DSA signatures are mostly unique, too.

As Heninger found out as well, the amount of ECDSA signatures is very small. (In fact, Lenstra had only one ECDSA in their database.) Therefore these have not been further analyzed by Lentra either.

The difference in the shared factors of PGP signatures can be explained by the time these were generated. As these signatures are not created on boot but later on during the execution of the operating system, the entropy pool has already been filled and therefore created fresh and proper values. PGP signatures have not been examined by Heninger's team.

5 Is Online Banking still safe?

Although both teams have roughly the same results, they both draw a different conclusion out of these facts. For Lentra's team, the amount of sharing in RSA is a cause for concern. They conclude that the proper seeding of a random number generator still seems to be a problematic issue due to the amount of X.509 certificates offering no security at all. They doubt that their results are new to parties having more sophisticated methods at one's disposal.

Heninger's team, as focused more on the security of random number generators, is less concerned about the amount of insecure certificates. They argue that only very few of these have been certified by CA, and even none of them belonged to a larger company.

All in all one can agree that the results of these research have not led to a significant reduction of the security of RSA, DSA or ElGamal.

That is why Online Banking is as (in)secure as it was before.

6 Lessons (to be) learned

Heninger's team went one step further, by examining the causes of the bad randomness observed. They inspected the random number generator of Linux and the OpenSSL package more in depth. Randomness can be taken from Linux's */dev/random* or */dev/urandom* pool, although *urandom* is non-blocking in case there is not enough entropy available. It is advised to use only *random* for cryptographic

systems in order to receive proper random values, although many applications use urandom instead. This way, the application remains responding even if there is not enough entropy available.

6.1 The Linux RNG

The random number generator has been stripped down to disable all sources of entropy unavailable at headless or embedded systems. The sources of entropy observed are:

- The uninitialized contents of the pool buffer at kernel start-up.
- The start-up clock time in nanoseconds.
- User input events and disk access timings.
- Entropy saved across boots.
- Entropy generated by changing the entropy pool during the calculation of random numbers when accessed by more than one thread.

They were surprised that entropy is no longer gathered through interrupts, as this may be the only source available at headless devices' first boot, having no real-time clock, no user input, and no disk. Interrupts are generated for example upon arrival of an ethernet package or a timer overflow.
In an experiment with a kernel where all these previously mentioned sources have been deactivated, they showed that the random number generator had a predictable and repeatable deterministic output. It takes more than one minute before entropy from the entropy pool is fed to */dev/urandom*, the so-called *"boot-time entropy hole"*. They showed that for example sshd does not only read from *urandom* instead of *random*, but also accesses the pool at about 3-4 seconds after boot, when urandom is still in a deterministic state. At this point in time, not even the stored entropy of previous boots has been restored. This vulnerability maintains if an application controls its own entropy pool fed by the deterministic RNG at boot. Nevertheless, on servers or desktop machines, entropy is gathered just fast enough to generate a well-chosen key.

6.2 Why there are still different keys

But why do keys with only one common factor exist? One could think that all keys should be identical when generated from a deterministic RNG. Instead, the first prime factor is mostly the same, whereas the second one does not divide any other moduli in most cases. Heninger's team explains this by the add of entropy to the pool once a value has been extracted and further examines the clock of the device. As an example, the source of entropy for the OpenSSL implementation has been reviewed. They found out that the internal entropy pool of OpenSSL is fed by the deterministic output of Linux's RNG and therefore is deterministic itself. However, if a random number is generated, for example by *bnrand()*, the current time in seconds, the process ID, and the uninitialized contents of the destination buffer if available are added to the entropy pool. They made three different assumptions:

- If during the generation of the key no second elapses, both primes are deterministic and identical to other instances.
- If a change in seconds occurs while generating the second prime, only the first prime is deterministic and repeatable. The second one will be unique.
- If a second elapses during the generation of the first prime, both primes are most likely to be distinct among all other primes of different instances.

They proved these assumptions by modifying the source code of OpenSSL, adding a dilation multiplier to the clock. All three assumptions could be verified as the multiplier has been increased. Some devices may not even have a (functional) built-in clock, resulting is the fact that no entropy is added to the pool at all.

The generation of the ephemeral keys for DSA has been investigated using Dropbear as an example. As for OpenSSL, the entropy pool of Dropbear is fed with vulnerable values from the operating system after boot. Dropbear uses an internal counter, which is added to the entropy pool once data has been extracted. They conclude that two (or more) Dropbear servers fed with the same deterministic output of urandom will stay in sync as long as the 32bit counter will not overflow. This has been proved by contacting previously found Dropbear servers. Even after a few hours, both servers were still in sync and used the same ephemeral key independently.

To fix these issues, Heninger et.al. propose several solutions:
Implement a cryptographically safe pseudo random number generator (CSPRNG) The operating system should provide a random number generator which is cryptographically safe, in order to satisfy the needs of secure applications.

Alter the PRNG interface The PRNG interface should report to the user or developer if enough entropy is available to process the request. This way, a developer can react to the state instead of having his application blocked by */dev/random*.

Do a better testing of RNG The output of the RNG on embedded or headless devices should be tested to confirm that enough entropy can be generated.

Use the most secure configuration as default Both, OpenSSL and Dropbear, had disabled options to increase the security offered. These option should be enabled as default.

Defensive use of RSA and DSA The applications should prevent weak entropy from revealing the private key by adding more entropy themselves.

Generate keys on first use The needed keys should be generated once they are needed. This is typically in a later stage of execution and not during the *"boot-time entropy hole"*.

Show warnings from lower layers In case the operating system reports that not enough entropy is available, for example by blocking requests to */dev/random*, applications should not use */dev/urandom* instead, but wait for the operating system to recover from this state.

Use no default certificates Manufacturers should not set default certificates in their product if possible.

True random seed during production Whenever it is possible to implement a unique true random seed during production, this should be done.

Offer effective entropy sources As mentioned before, most headless systems offer only a few sources for entropy. These rare sources should be effective and used before generating random numbers.

Use hardware random number generators If possible, a hardware random number generator should be implemented on the device.

Check keys prior to certifying them Although this is already done yet, the CAs should process their databases in the same manner as Heninger's team did.

Recreate weak keys The end-user should create fresh keys instead of using the insecure, default or duplicate keys.

Check your key The end-user can check his own public key on the website of Heninger's team for known weaknesses.

Improve RNG The built-in RNG in operating systems have to improved in order to provide no more entropy problems.

Develop cryptographic primitives not as prone to weak entropy as the current ones Current cryptographic systems fail when used with weak entropy. Future systems should be developed with this factor in mind.

7 Conclusion

The development and improvement of random number generators has been a field of research for many years and is not to be neglectable in the present. Both teams presented the results of bad randomness although they both draw different conclusions out of this. At least the work of Heninger et al. has led to a significant improvement of current systems. The several manufacturers identified during this research have been contacted and partly reacted on these issues. Not only the fact that this paper was awarded the *Best Paper Award* makes it worth reading.
As Heninger et al. proposed in their work, the security settings of applications should be set to the highest available settings as default, instead of using a lower, more comfortable value. The Linux RNG has been patched to reduce the boot-time entropy hole.

Sadly, Lenstra et al. make no suggestions on how to fix this obvious problem, instead they only refer to other already known weaknesses in the analyzed algorithms.

Although both teams come to a different conclusion about the impact of their studies, one common factor remains: cryptographic algorithms have to relay on cryptographically secure random number generators.

8 Bibliography

1. **Mining your P's and Q's: Detection of Widespread Weak Keys in Network Devices**, Nadia Heninger (University of California, San Diego), Zakri Durumeric, Eric Wustrow, and J. Alex Halderman (all University of Michigan), in proceedings of the 21st USENIC Security Symposium, August 2012

2. **Ron was wrong, Whit is right**, Arjen K. Lenstra (EPFL Lausanne), James P. Hughes (Self, Palo Alto, CA), Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, Christophe Wachter (all EPFL Lausanne), Feburary 2012

3. **DSA-1571-1 openssl – predictable random number generator**, Debian Security Advisory, Luciano Bello, May 2008, `http://www.debian.org/security/2008/dsa-1571` as of February 12, 2013

4. **Fast multiplication and its applications**, D. J. Bernstein, Algorithmic Number Theory, May 2008, p.325 - 284

5. **Einführung in die Kryptographie**, Buchmann, 3rd Edition, Springer

9 Appendix A

The following trees give an example of the Bernstein Algorithm:

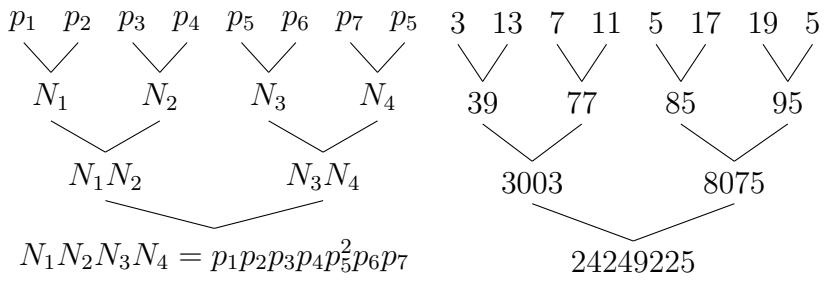

Figure 12: Example of Bernstein Algorithm: Multiplication Tree.

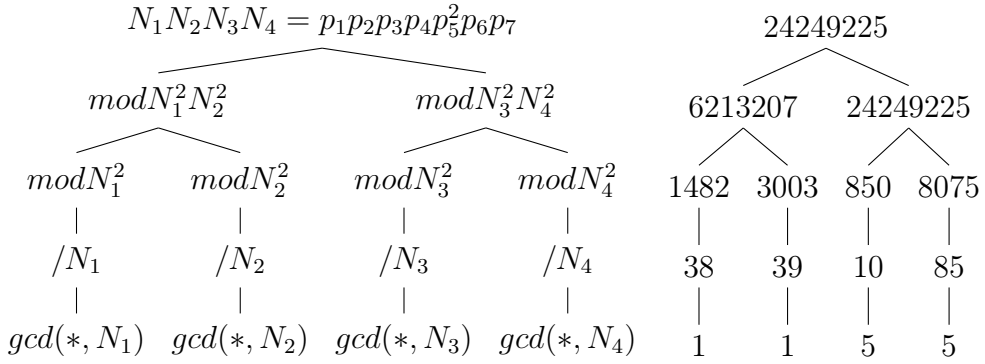

Figure 13: Example of Bernstein Algorithm: Remainder Tree.

Thus, N_3 and N_4 are sharing the same prime factor $p_5 = 5$.