



NTNU

Norwegian University of
Science and Technology

RANDOMNESS 3: RANDOMIZATION

TTM4205 – Lecture 4

Tjerand Silde

29.08.2025

Contents

Announcements

Primality Testing

Factorization

De-Randomization

Takeaways

Contents

Announcements

Primality Testing

Factorization

De-Randomization

Takeaways

Reference Group

I am looking for 3-4 students to join the reference group. We will meet three times during the semester, and your feedback is extremely valuable.

Send me an email and/or talk to me in the break :)

Reference Material

These slides are based on:

- ▶ The referred papers in the slides
- ▶ JPA: parts of chapter 9

Contents

Announcements

Primality Testing

Factorization

De-Randomization

Takeaways

How do we check if
a number is prime?

Deterministic Methods

- ▶ Brute Force
- ▶ Sieving methods
- ▶ Wilson's Theorem?

Brute Force Testing

It is always possible to check all possibilities. But how long time does it take?
Assume that p is of 2048 bits.

Brute Force Testing

It is always possible to check all possibilities. But how long time does it take?
Assume that p is of 2048 bits.

- ▶ divisible by any number between 1 and p ? $\sim 2^{2048}$

Brute Force Testing

It is always possible to check all possibilities. But how long time does it take?
Assume that p is of 2048 bits.

- ▶ divisible by any number between 1 and p ? $\sim 2^{2048}$
- ▶ by 2 or any odd number between 1 and p ? $\sim 2^{2047}$

Brute Force Testing

It is always possible to check all possibilities. But how long time does it take?
Assume that p is of 2048 bits.

- ▶ divisible by any number between 1 and p ? $\sim 2^{2048}$
- ▶ by 2 or any odd number between 1 and p ? $\sim 2^{2047}$
- ▶ by 2 or any odd number between 1 and \sqrt{p} ? $\sim 2^{1023}$

Brute Force Testing

It is always possible to check all possibilities. But how long time does it take?
Assume that p is of 2048 bits.

- ▶ divisible by any number between 1 and p ? $\sim 2^{2048}$
- ▶ by 2 or any odd number between 1 and p ? $\sim 2^{2047}$
- ▶ by 2 or any odd number between 1 and \sqrt{p} ? $\sim 2^{1023}$

This is infeasible to compute! Even 2^{128} is considered out of reach.

Sieving Methods

It is possible to pre-compute many small prime numbers to speed up the process, e.g., the sieve of Eratosthenes:

Sieving Methods

It is possible to pre-compute many small prime numbers to speed up the process, e.g., the sieve of Eratosthenes:

First, keep 2 and remove all even numbers. Then, keep 3 and remove all multiples of three. 4 is already removed. Keep 5 and remove all multiples of five. 6 has already been removed. ...

Sieving Methods

It is possible to pre-compute many small prime numbers to speed up the process, e.g., the sieve of Eratosthenes:

First, keep 2 and remove all even numbers. Then, keep 3 and remove all multiples of three. 4 is already removed. Keep 5 and remove all multiples of five. 6 has already been removed. ...

It still requires exponential work to check all possibilities!

Wilson's Theorem

Wilson's Theorem: A natural number $p > 1$ is a prime number if and only if the product of all the positive integers less than p is one less than a multiple of p .

Wilson's Theorem

Wilson's Theorem: *A natural number $p > 1$ is a prime number if and only if the product of all the positive integers less than p is one less than a multiple of p .*

This means: $(p - 1)! \equiv -1 \pmod{p} \iff p$ is a prime number.

Wilson's Theorem

Wilson's Theorem: *A natural number $p > 1$ is a prime number if and only if the product of all the positive integers less than p is one less than a multiple of p .*

This means: $(p - 1)! \equiv -1 \pmod{p} \iff p$ is a prime number.

It still requires exponential work to compute $(p - 1)!$

Wilson's Theorem

Wilson's Theorem: *A natural number $p > 1$ is a prime number if and only if the product of all the positive integers less than p is one less than a multiple of p .*

This means: $(p - 1)! \equiv -1 \pmod{p} \iff p$ is a prime number.

It still requires exponential work to compute $(p - 1)!$

But it is possible to use similar techniques to speed it up.

Randomized Methods

- ▶ Monte Carlo algorithms
- ▶ The Miller-Rabin method

Monte Carlo Algorithms

A Monte Carlo algorithm is a randomized algorithm whose output may be incorrect with a given probability.

Monte Carlo Algorithms

A Monte Carlo algorithm is a randomized algorithm whose output may be incorrect with a given probability.

A **false**-biased Monte Carlo algorithm is always correct when it returns **false**, but it might be wrong if it outputs **True**. Similar for a **true**-biased algorithm.

Monte Carlo Algorithms

A Monte Carlo algorithm is a randomized algorithm whose output may be incorrect with a given probability.

A **false**-biased Monte Carlo algorithm is always correct when it returns **false**, but it might be wrong if it outputs **True**. Similar for a **true**-biased algorithm.

If the probability of being wrong is $1/2$, then it can be amplified by parallel repetition: λ rounds gives the probability $(\frac{1}{2})^\lambda \rightarrow 0$ of being wrong.

Monte Carlo Algorithms

A Monte Carlo algorithm is a randomized algorithm whose output may be incorrect with a given probability.

A **false**-biased Monte Carlo algorithm is always correct when it returns **false**, but it might be wrong if it outputs **True**. Similar for a **true**-biased algorithm.

If the probability of being wrong is $1/2$, then it can be amplified by parallel repetition: λ rounds gives the probability $(\frac{1}{2})^\lambda \rightarrow 0$ of being wrong.

Some commonly used algorithms: Soloway-Strassen, Fermat testing (**warning**: Carmichael numbers) and Miller-Rabin.

Miller-Rabin Primality Testing

Fermat's little theorem: if p is prime, then we have that $a^{p-1} \equiv 1 \pmod{p}$.

Miller-Rabin Primality Testing

Fermat's little theorem: if p is prime, then we have that $a^{p-1} \equiv 1 \pmod{p}$.

Let p be an odd integer and write $p - 1$ as $2^s \cdot d$ where d is an odd number.

Miller-Rabin Primality Testing

Fermat's little theorem: if p is prime, then we have that $a^{p-1} \equiv 1 \pmod{p}$.

Let p be an odd integer and write $p - 1$ as $2^s \cdot d$ where d is an odd number.

Let $1 < a < p$ be a randomly sampled integer. Then $a^d \equiv 1 \pmod{p}$ or $a^{2^r d} \equiv -1 \pmod{p}$ for some $0 \leq r < s$. Only $\log_2 p$ options to check.

Miller-Rabin Primality Testing

Fermat's little theorem: if p is prime, then we have that $a^{p-1} \equiv 1 \pmod{p}$.

Let p be an odd integer and write $p - 1$ as $2^s \cdot d$ where d is an odd number.

Let $1 < a < p$ be a randomly sampled integer. Then $a^d \equiv 1 \pmod{p}$ or $a^{2^r d} \equiv -1 \pmod{p}$ for some $0 \leq r < s$. Only $\log_2 p$ options to check.

If this is false, then p is composite. However, the above fact is true for roughly $\frac{1}{4}$ of all composite numbers for a randomly sampled value a .



Miller-Rabin Primality Testing

Fermat's little theorem: if p is prime, then we have that $a^{p-1} \equiv 1 \pmod{p}$.

Let p be an odd integer and write $p - 1$ as $2^s \cdot d$ where d is an odd number.

Let $1 < a < p$ be a randomly sampled integer. Then $a^d \equiv 1 \pmod{p}$ or $a^{2^r d} \equiv -1 \pmod{p}$ for some $0 \leq r < s$. Only $\log_2 p$ options to check.

If this is false, then p is composite. However, the above fact is true for roughly $\frac{1}{4}$ of all composite numbers for a randomly sampled value a .

If we sample λ random values a , the Miller-Rabin primality testing algorithm has $(\frac{1}{4})^\lambda$ chance of being wrong every time, which becomes negligible.

Primality Testing APIs

The most common way of checking the primality of a candidate p is a combination of the above as follows:

Primality Testing APIs

The most common way of checking the primality of a candidate p is a combination of the above as follows:

1. Pre-compute a list of the first thousand prime numbers.

Primality Testing APIs

The most common way of checking the primality of a candidate p is a combination of the above as follows:

1. Pre-compute a list of the first thousand prime numbers.
2. Check p is divisible by any prime number in the list.

Primality Testing APIs

The most common way of checking the primality of a candidate p is a combination of the above as follows:

1. Pre-compute a list of the first thousand prime numbers.
2. Check p is divisible by any prime number in the list.
3. Run the Miller-Rabin algorithm, say, ~ 40 times.

Primality Testing APIs

The most common way of checking the primality of a candidate p is a combination of the above as follows:

1. Pre-compute a list of the first thousand prime numbers.
2. Check p is divisible by any prime number in the list.
3. Run the Miller-Rabin algorithm, say, ~ 40 times.
4. If all checks succeeds, then output: *probably prime*.

Primality Testing Failures

It might be highly rewarding for an attacker to convince you that a composite number is prime. For example:

Primality Testing Failures

It might be highly rewarding for an attacker to convince you that a composite number is prime. For example:

RSA is not secure if n is not a bi-prime, and it is easy to compute discrete logarithms in composite order groups (not containing large prime subgroups).

Primality Testing Failures

It might be highly rewarding for an attacker to convince you that a composite number is prime. For example:

RSA is not secure if n is not a bi-prime, and it is easy to compute discrete logarithms in composite order groups (not containing large prime subgroups).

A classic mistake in Miller-Rabin: Integers a are sampled randomly but pre-fixed before testing. This gives an attacker the chance to find composite numbers (pseudo-primes) that pass the test after all.

Primality Testing Failures

It might be highly rewarding for an attacker to convince you that a composite number is prime. For example:

RSA is not secure if n is not a bi-prime, and it is easy to compute discrete logarithms in composite order groups (not containing large prime subgroups).

A classic mistake in Miller-Rabin: Integers a are sampled randomly but pre-fixed before testing. This gives an attacker the chance to find composite numbers (pseudo-primes) that pass the test after all.

Sometimes it is a mix between fixed a 's and freshly sampled a 's, still giving the adversary a good chance to fool the test.

Prime and Prejudice: Primality Testing Under Adversarial Conditions

Martin R. Albrecht¹, Jake Massimo¹, Kenneth G. Paterson¹, and Juraj Somorovsky²

¹ Royal Holloway, University of London

² Ruhr University Bochum, Germany

`martin.albrecht@rhul.ac.uk`, `jake.massimo.2015@rhul.ac.uk`, `kenny.paterson@rhul.ac.uk`,
`juraj.somorovsky@rub.de`

Figure: <https://eprint.iacr.org/2018/749.pdf>

The Need for Secure Primality Testing

Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation

Steven Galbraith¹, Jake Massimo², and Kenneth G. Paterson²

¹ University of Auckland

² Royal Holloway, University of London

s.galbraith@auckland.ac.nz, jake.massimo.2015@rhul.ac.uk,

kenny.paterson@rhul.ac.uk

Figure: <https://eprint.iacr.org/2019/032.pdf>

Secure Primality Testing API

A Performant, Misuse-Resistant API for Primality Testing

Jake Massimo¹ and Kenneth G. Paterson²

¹ Information Security Group,
Royal Holloway, University of London
`jake.massimo.2015@rhul.ac.uk`

² Department of Computer Science,
ETH Zurich
`kenny.paterson@inf.ethz.ch`

Figure: <https://eprint.iacr.org/2020/065.pdf>

Contents

Announcements

Primality Testing

Factorization

De-Randomization

Takeaways

How do we factor
large bi-primes?

Deterministic Methods

Some trivial ways to attack an RSA moduli n :

Deterministic Methods

Some trivial ways to attack an RSA moduli n :

- ▶ Brute force by checking if n is divisible by 2 or any odd numbers less than \sqrt{n} . This requires exponential work...

Deterministic Methods

Some trivial ways to attack an RSA moduli n :

- ▶ Brute force by checking if n is divisible by 2 or any odd numbers less than \sqrt{n} . This requires exponential work...
- ▶ Even only checking divisibility against primes between 2^{1023} and 2^{1024} for 2048 bit n requires exponential work...

Deterministic Methods

Some trivial ways to attack an RSA moduli n :

- ▶ Brute force by checking if n is divisible by 2 or any odd numbers less than \sqrt{n} . This requires exponential work...
- ▶ Even only checking divisibility against primes between 2^{1023} and 2^{1024} for 2048 bit n requires exponential work...
- ▶ Fermat Factorization find prime factors close to \sqrt{n} .

Deterministic Methods

Some trivial ways to attack an RSA moduli n :

- ▶ Brute force by checking if n is divisible by 2 or any odd numbers less than \sqrt{n} . This requires exponential work...
- ▶ Even only checking divisibility against primes between 2^{1023} and 2^{1024} for 2048 bit n requires exponential work...
- ▶ Fermat Factorization find prime factors close to \sqrt{n} .
- ▶ Pollard's Rho algorithm find largest prime factor in $\sqrt[4]{n}$

Deterministic Methods

Some trivial ways to attack an RSA moduli n :

- ▶ Brute force by checking if n is divisible by 2 or any odd numbers less than \sqrt{n} . This requires exponential work...
- ▶ Even only checking divisibility against primes between 2^{1023} and 2^{1024} for 2048 bit n requires exponential work...
- ▶ Fermat Factorization find prime factors close to \sqrt{n} .
- ▶ Pollard's Rho algorithm find largest prime factor in $\sqrt[4]{n}$

Randomness comes to the rescue in this situation as well!

Randomized Methods

The Number Field Sieve is the most efficient algorithm for factorization.

Randomized Methods

The Number Field Sieve is the most efficient algorithm for factorization.

Collect many random pairs (a_i, b_i) where $a_i^x \equiv b_i^y \pmod n$, for integers x, y .

Randomized Methods

The Number Field Sieve is the most efficient algorithm for factorization.

Collect many random pairs (a_i, b_i) where $a_i^x \equiv b_i^y \pmod n$, for integers x, y .

Then these equations can be combined in such a way that we can find a and b satisfying $a^2 \equiv b^2 \pmod n$, which means that $a^2 - b^2 \equiv (a - b)(a + b) \equiv 0 \pmod n$.

Randomized Methods

The Number Field Sieve is the most efficient algorithm for factorization.

Collect many random pairs (a_i, b_i) where $a_i^x \equiv b_i^y \pmod n$, for integers x, y .

Then these equations can be combined in such a way that we can find a and b satisfying $a^2 \equiv b^2 \pmod n$, which means that $a^2 - b^2 \equiv (a - b)(a + b) \equiv 0 \pmod n$.

Then we *might* find a factor of n by computing the greatest common divisor between n and $a - b$ and $a + b$.

Number Field Sieve

The running time of the Number Field Sieve is

$$\exp\left(\left(64/9\right)^{1/3}(\log n)^{1/3}(\log \log n)^{2/3}(1 + o(1))\right)$$

Number Field Sieve

The running time of the Number Field Sieve is

$$\exp\left(\left(\frac{64}{9}\right)^{1/3}(\log n)^{1/3}(\log \log n)^{2/3}(1 + o(1))\right)$$

This algorithm is *sub-exponential*. The largest number we have ever factored (in public) is of size 829 bits.

Number Field Sieve

The running time of the Number Field Sieve is

$$\exp\left(\left(\frac{64}{9}\right)^{1/3}(\log n)^{1/3}(\log \log n)^{2/3}(1 + o(1))\right)$$

This algorithm is *sub-exponential*. The largest number we have ever factored (in public) is of size 829 bits.

Factoring as a service: In 2015, it was possible to factor 512 bit RSA keys in less than four hours.



Factoring as a Service

Luke Valenta, Shaanan Cohney, Alex Liao,
Joshua Fried, Satya Bodduluri, Nadia Heninger

University of Pennsylvania

Figure: <https://eprint.iacr.org/2015/1000.pdf>

Breaking into mailboxes today...

Knakk svak nøkkel

– Vi fant mest mulig info, og samla det sammen for å lage en antakelse om hvordan dette fungerte, forklarer UnblvR.

Ett av de viktigste funnene i prosessen var at sikkerheten stod og falt på en offentlig 512-bits RSA-nøkkel – én for PostNord-boksene, én for Posten-boksene.

RSA-nøkler på bare 512 bits ble knekt allerede i 1999, og blir ikke ansett å være sikre å bruke i 2025. Det fikk utviklerne raskt se at de heller ikke er:

De leide seg en server på nett, brukte offentlig tilgjengelig programvare – og på under ett døgn, for under femtilappen, var pakkeboks-nøklerne knekt.

– Da vi trodde vi hadde skjønt alt, betalte vi Posten for å sende en pakke, og gikk til pakkeboksen for å prøve å åpne luka vi hadde fått tildelt, fra vår egen løsning, forteller UnblvR.

– Og det fikk vi til.

Figure: <https://www.kode24.no/artikkel/her-hacker-han-opp-pakkeboksen/243107>

State of the Art

The state of the art in integer factoring and breaking public key cryptography

Fabrice Boudot¹, Pierrick Gaudry², Aurore Guillevic², Nadia Heninger³, Emmanuel Thomé²,
and Paul Zimmermann²

¹Université de Limoges, XLIM, UMR 7252, F-87000 Limoges, France

²Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

³University of California, San Diego, USA

Figure: <https://hal.science/hal-03691141/document>

RSA, DH and DSA in the Wild*

Nadia Heninger

University of California, San Diego, USA

Figure: <https://eprint.iacr.org/2022/048.pdf>

Fermat Factorization in the Wild

Hanno Böck

January 8, 2023

Figure: <https://eprint.iacr.org/2023/026.pdf>

Ron was wrong, Whit is right

Arjen K. Lenstra¹, James P. Hughes²,
Maxime Augier¹, Joppe W. Bos¹, Thorsten Kleinjung¹, and Christophe Wachter¹

¹ EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland

² Self, Palo Alto, CA, USA

Figure: <https://eprint.iacr.org/2012/064.pdf>

Shared Prime Factors

Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices

Nadia Heninger^{†*} Zakir Durumeric^{‡*} Eric Wustrow[‡] J. Alex Halderman[‡]

[†] *University of California, San Diego*
nadiah@cs.ucsd.edu

[‡] *The University of Michigan*
{zakir, ewust, jhalderm}@umich.edu

Figure: Check out the blog post, paper and slides: 1)

<https://freedom-to-tinker.com/2012/02/15/new-research-theres-no-need-panic-over-factorable-keys-just-mind-your-ps-and-qs>, 2)

<https://factorable.net/weakkeys12.extended.pdf>, 3)

<https://crypto.stanford.edu/RealWorldCrypto/slides/nadia.pdf>

Contents

Announcements

Primality Testing

Factorization

De-Randomization

Takeaways

De-Randomized Crypto

We need randomness for CPA secure encryption, right!?

We DO need randomness for key generation. However:

De-Randomized Crypto

We need randomness for CPA secure encryption, right!?

We DO need randomness for key generation. However:

- ▶ Use the secret key to generate nonces

De-Randomized Crypto

We need randomness for CPA secure encryption, right!?

We DO need randomness for key generation. However:

- ▶ Use the secret key to generate nonces
- ▶ Schnorr example with $r = H(sk, m)$

De-Randomized Crypto

We need randomness for CPA secure encryption, right!?

We DO need randomness for key generation. However:

- ▶ Use the secret key to generate nonces
- ▶ Schnorr example with $r = H(sk, m)$
- ▶ Counters + master seed + hashing

De-Randomized Crypto

We need randomness for CPA secure encryption, right!?

We DO need randomness for key generation. However:

- ▶ Use the secret key to generate nonces
- ▶ Schnorr example with $r = H(sk, m)$
- ▶ Counters + master seed + hashing
- ▶ HMAC with key for deterministic MAC

De-Randomized Crypto

We need randomness for CPA secure encryption, right!?

We DO need randomness for key generation. However:

- ▶ Use the secret key to generate nonces
- ▶ Schnorr example with $r = H(sk, m)$
- ▶ Counters + master seed + hashing
- ▶ HMAC with key for deterministic MAC
- ▶ Hedging techniques (paper on next slide)

Hedged Public-Key Encryption: How to Protect Against Bad Randomness

Mihir Bellare* Zvika Brakerski† Moni Naor‡ Thomas Ristenpart§
Gil Segev¶ Hovav Shacham|| Scott Yilek**

April 21, 2012

Figure: <https://www.cs.utexas.edu/~hovav/dist/hedge.pdf>

Contents

Announcements

Primality Testing

Factorization

De-Randomization

Takeaways

Random Number Generation

Check the quality of the built-in RNG that you rely on:

- ▶ How does it collect randomness?
- ▶ Is the RNG seeded / pre-seeded?
- ▶ How much entropy does it provide?
- ▶ Does it warn you about issues?
- ▶ Is it cryptographically secure?
- ▶ (Linux's */dev/random* vs */dev/urandom*)

Faulty Voting Randomness

A faulty PRNG in a voting system – a real-world cryptographic disaster

Kristian Gjøsteen

Department of Mathematical Sciences
Norwegian University of Science and Technology
Real World Crypto, January 2018

Figure: https://youtu.be/xq_6ey2JGAE?feature=shared

Pseudo-Random Number Generation

Check the quality of the built-in PRNG that you rely on:

- ▶ Does it rely on a proper RNG as seed? Is it pre-seeded?
- ▶ Is the PRNG cryptographically secure? NIST-approved?
- ▶ Verify the output: Do values repeat? Correct bit-size?
- ▶ Which library/version is used? Known vulnerabilities?

Some good resources are available at

<https://github.com/veorq/cryptocoding#use-strong-randomness>.

A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications

Figure: <https://csrc.nist.gov/pubs/sp/800/22/r1/upd1/final>

Choice of Primitives

Check the cryptographic primitive that you rely on:

- ▶ Does it rely on a proper PRNG? Is it pre-seeded?
- ▶ Is it the newest/most secure primitive? NIST-approved?
- ▶ Verify the output: Do values repeat? Correct bit-size?
- ▶ Which library/version is used? Known vulnerabilities?
- ▶ Are there de-randomized algorithms available instead?

Rolling Your Own Crypto

Security Cryptography Whatever

The Great "Roll Your Own Crypto" Debate with Filippo Valsorda

JULY 31, 2021 SECURITY, CRYPTOGRAPHY, WHATEVER



Figure: <https://securitycryptographywhatever.buzzsprout.com/1822302/895384-2-the-great-roll-your-own-crypto-debate-with-filippo-valsorda>

Questions?