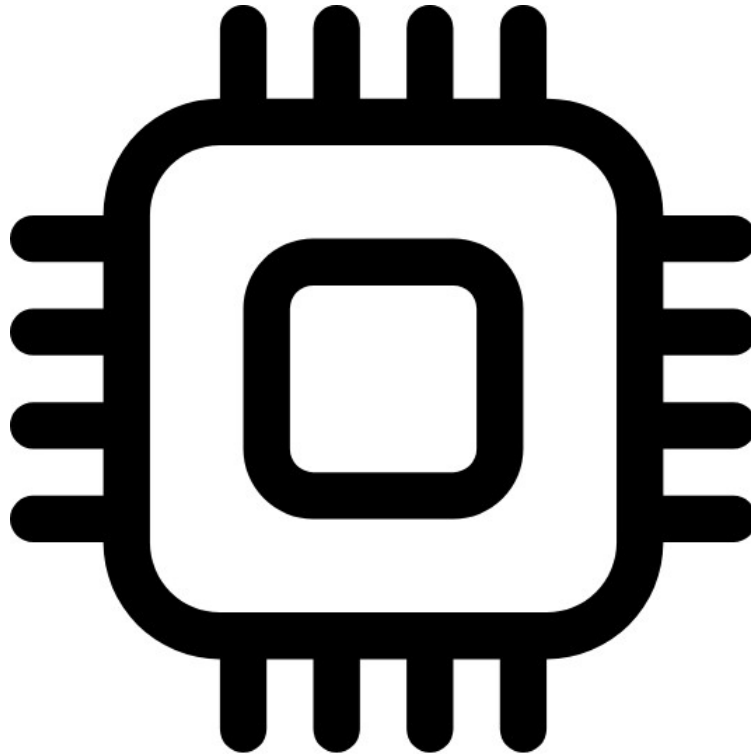


# RSACrackstation

Daniel Nettelfield og Gustav Nybro



En teknologisk og matematisk gennemgang af RSA kryptering  
og RSACrackstation projektet, udviklet i forbindelse med  
programmeringsfaget på HTX.

Teknisk Gymnasium Silkeborg

3x

31/10 – 2022

# Abstract

Formålet med projektet er at kunne demonstrere vores tillærte viden fra programmeringsfaget. Ved brug af vores ræsonnementskompetence har vi formået at sammesætte et projekt, der dækker alle de faglige mål i programmeringsfaget.

Vi har fremstillet en hjemmeside til at behandle RSA, på en brugervenlig og hurtig måde. Koden kan tilgås fra links i bilag og det færdige produkt kan tilgås på internettet, på [rsacrackstation.com](http://rsacrackstation.com).

# Indhold

<b>1</b>	<b>Problemformulering</b>	<b>3</b>
1.1	Beskrivelse . . . . .	3
1.2	Krav . . . . .	3
<b>2</b>	<b>Hvad er RSA?</b>	<b>3</b>
2.1	Hvordan virker RSA? . . . . .	3
<b>3</b>	<b>Hvad er en API?</b>	<b>4</b>
<b>4</b>	<b>Frontend</b>	<b>4</b>
4.1	Intro . . . . .	4
4.2	Tailwind CDN . . . . .	4
4.3	Darkmode . . . . .	5
4.4	Hex Auto Toggle . . . . .	5
4.5	AJAX Call . . . . .	6
<b>5</b>	<b>Backend</b>	<b>7</b>
5.1	Primtalsfaktorisering . . . . .	7
5.2	RSA Funktionalitet . . . . .	9
<b>6</b>	<b>Test af programmet</b>	<b>10</b>
6.1	Test af kryptering . . . . .	10
6.2	Test af faktorisering . . . . .	10
6.3	Test af RSA-brydning . . . . .	11
<b>7</b>	<b>Konklusion</b>	<b>11</b>
<b>8</b>	<b>Billag</b>	<b>11</b>
8.1	Kilde Kode - Zip Fil . . . . .	11
8.2	Kilde Kode - Github . . . . .	11
8.3	URL - RSACrackstation . . . . .	11

# 1 Problemformulering

## 1.1 Beskrivelse

Hjemmesiden er en side der kan faktorisere primtal ved hjælp af API call til en ekstern database. Derudover kan hjemmesiden bruges til at bryde krypteringsalgoritmen RSA, og dertil dekryptere indtastet tekst. Derudover kan hjemmesiden også bruges til at kryptere tekst med RSA. Hjemmesidens målgruppe er primært CTF spillere, cybersikkerheds entusiaster og andre der skal kryptere og dekryptere med RSA.

## 1.2 Krav

- Hjemmesiden skal kunne faktorisere primtal.
- Hjemmesiden skal kunne bryde svag RSA kryptering.
- Hjemmesiden skal kunne kryptere med RSA.

# 2 Hvad er RSA?

RSA er en asymmetrisk kryptografialgoritme, som er udviklet af Ron Rivest, Adi Shamir og Leonard Adleman i 1977. En asymmetrisk kryptografialgoritme er en kryptografialgoritme der bruger to forskellige nøgler til at kryptere og dekryptere, og bruges ofte af blandt andet banker, til at sikre deres data. Idéen er at man kan offentliggøre krypteringsnøglen, men beholde dekrypteringsnøglen privat, så brugere kan kryptere data på deres egen maskine, og sende den krypterede tekst til banken, som dekrypterer den med sin private nøgle. RSA nøglerne er genereret ved at bruge to primtal, som er valgt tilfældigt. Den offentlige nøgle regnes ved at gange de to primtal sammen, og den private ved at bruge en kompliceret algoritme, der hedder den Udvidede Euklidiske Algoritme. De to tilfældige primtal kaldes  $p$  og  $q$ , deres produkt (den offentlige nøgle) kaldes  $n$ , og den private nøgle kaldes  $d$ . Udover at bruge primtal til at generere RSA nøglerne, bruger RSA også en eksponent, som har en række af krav den skal opfylde, alt efter værdierne af de to primtal. Dog findes der et tal, som altid kan bruges som eksponent, og det er 65537, denne eksponent kaldes  $e$ . Det sidste tal man bruger i RSA, hedder  $m$ , og er den værdi man gerne vil sende, ofte en tekst.

## 2.1 Hvordan virker RSA?

Når man krypterer med RSA, bruges følgende formel:

$$c = m^e \mod n \quad (1)$$

Her bruges modulus regning, som er måden man sikrer, at den krypterede tekst, ikke kan dekrypteres. Modulus virker nemlig ved at tage resten efter en heltalsdivision, det vil sige at flere udregninger godt kan give det samme resultat

$$2^{11} \mod 527 = 467 \quad (2)$$

$$106^{13} \mod 551 = 467 \quad (3)$$

I dette eksempel, er  $m = 2$ , i ligning (2) og  $m = 106$  i ligning (3). Så man kan man se at bare fordi man kender  $c$  betyder det ikke at man kender  $m$ , da  $c$  er ens i begge eksempler, men  $m$

er forskellige. For at dekryptere denne tekst skal man bruge værdien for  $\mathbf{d}$ , som findes ved at faktorisere de to  $\mathbf{n}$ 'er. I dette eksempel bruger vi meget små tal, og det er derfor meget nemt at faktorisere de to moduluer. Når man har faktoreret  $\mathbf{n}$ , kan man køre den udvidede Euklidske algoritme, som bruges til at finde  $\mathbf{d}$ . Hvis man kører denne algoritme på vores to eksempler, får man  $\mathbf{d} = 131$  og  $\mathbf{d} = 349$ .  $\mathbf{d}$  kaldes for den inverse af  $\mathbf{e}$  mod  $\mathbf{n}$ , og bruges til at gøre det modsatte af kryptering. Når man har regnet  $\mathbf{d}$ , er det meget enkelt at dekryptere, man skal nemlig gøre det samme som når man kryptere, men med  $\mathbf{d}$  i stedet for  $\mathbf{e}$ :

$$m = c^d \mod n \quad (4)$$

Hvis man gør dette for vores to eksempler får man dette:

$$467^{131} \mod 527 = 2 \quad (5)$$

$$467^{349} \mod 551 = 106 \quad (6)$$

Hvor de to resultater, er de samme to tal, vi krypterede til at starte med, og vi har altså dekrypteret vores tekst.

### 3 Hvad er en API?

API står for Application Programming Interface, hvilket er en måde for noget kode at interagere med noget andet kode. En API er modsætningen til en brugergrænseflade, som er en grænseflade rettet mod mennesker. API kald kan enten ændre data eller hente data fra en server. APIs er overalt på internettet og størstedelen af populære hjemmesider bruger APIs. Et godt eksempel på en API der bliver brugt af mange mennesker, er Rejseplanen. Rejseplanen fungerer ved at en bruger indtaster noget data, som bliver sendt til backenden. Backend sender kun rå JSON data tilbage, som frontenden behandler og viser på en måde, brugeren lettere kan forstå.

## 4 Frontend

### 4.1 Intro

For at kunne bruge hjemmesiden ordenligt er det vigtigt med et brugervenligt interface. For at kunne lave et brugervenligt interface, skal der bruges HTML, CSS og JavaScript. Det er muligt at lave et flot og meget brugervenligt interface med blot normal CSS og JavaScript, men for at hurtigere kunne udvikle vores produkt, valgte vi at bruge Tailwind CSS og JQuery i stedet. Tailwind CSS er et CSS framework, der har en masse prædefinerede klasser, som kun har én bestemt funktion. Man kan derfor hurtigt lave et flot interface, uden at skulle skrive CSS selv, men bare bruge de prædefinerede klasser. Tailwind gør det også nemmere at lave et responsivt UI, da hele frameworket er bygget op omkring det.

### 4.2 Tailwind CDN

For at kunne bruge Tailwind frameworket, har man flere forskellige muligheder, dog er der to oplagte:

1. Installere Tailwind via NPM
2. Lade Tailwind med en CDN (Content Delivery Network)

Eftersom vi ikke ville gøre det her projekt mere avanceret end nødvendig, og fordi projektet ikke skal hostes i et produktionsmiljø, kan vi godt nøjes med CDN versionen. Hvis hjemmesiden skulle bruges i produktion, ville man ikke bruge Tailwind via CDN, fordi det er ressourcer der skal hentes fra en 3. part, hvilket kan give forskellige problemer. Hvis Tailwinds servere går ned, går alle hjemmesider der bruger Tailwinds CDN ned, fordi CSS'en ikke kan loades. CDN'en indeholder også alle klasser og skal derfor loades mere end nødvendigt, hvilket kan gøre hjemmesiden langsommere.

Dette projekt var også småt nok, til at det ville tage en betydelig del af tiden, at sætte Tailwind op ordenligt. Tailwind via NPM virker kun, hvis der kører en process og scanner HTML-filerne for opdateringer og laver de klasser der mangler. Det betyder så også at Tailwind kun indeholder hvad der bliver brugt og intet andet, i modsætning til mange andre CSS frameworks, så som Bootstrap.

### 4.3 Darkmode

Alle vores HTML elementer som er blevet ændret med Tailwind, har to forskellige klasser - en til darkmode og en til lightmode. Et eksempel kan ses nedenfor

---

```
<div class="w-fit dark:text-white text-gray-800">
...

```

---

I eksemplet ovenfor bliver der oprettet et `<div>` tag som indeholder nogle elementer. Tagget har to klasser der ændrer tekstens farve, som burde overskrive hinanden, men fordi den ene har *dark*: foran, bliver den kun brugt når darkmode er slået til.

### 4.4 Hex Auto Toggle

Hjemmesiden kan både arbejde med decimaltal og hexadecimaltal, og der er en knap der skifter mellem dem. Hvis man kopierer tallet et sted fra, kan man nemt glemme at skifte mellem de to, hvilket vi løser, ved at automatisk skifte mellem dem, afhængig af hvad der er indsat i inputfeltet. Koden nedenfor skifter automatisk mellem de to.

---

```
1 // Checks if number is pasted and automatically changes state to hex if hex is pasted
2 $("#num").bind('paste', function(e) {
3     let data = e.originalEvent.clipboardData.getData('text'); // Gets data from clipboard
4     if (/^[A-Za-z]/g.test(data)) { // Regex to check if hex is pasted
5         if (!isHex){
6             updateHexToggle();
7         }
8     }
9     else if (isHex){
10        updateHexToggle()
11    }
12 });
```

---

Listing 1: Views/Home/Index.cshtml, Linje 49 – 60

I linje 2 refererer vi til elementet *num*, som er tekstfeltet, hvor man kan indtaste sit tal. Efter det tjekker vi om der bliver indsat et tal ind i inputfeltet og laver en funktion, som kører hvis der bliver indsat noget.

I linje 3 initialiserer vi en variabel i local scope, der indeholder dataen i upklipsholderen, som læses direkte fra udklipsholderen i stedet for tekstfeltet. Dette gør vi fordi værdien af inputfeltet ikke har ændret sig, når funktionen kører.

I linje 4–8 tjekker vi om dataen fra variabelen i linje 3, indeholder store eller små bogstaver og kører funktionen *updateHexToggle()* hvis variabelen *isHex* er falsk. Vi kører funktionen for at opdatere knappen man kan se på hjemmesiden, samt variabelen *isHex*. If-statementet tjekker om der er bogstaver i teksten, da hexadecimale tal ofte er en blanding mellem bogstaver og tal. Dette er ikke en helt sikker måde at skifte mellem tallene, fordi mindre tal nogle gange ikke indeholder bogstaver. Det er dog umuligt at tjekke om et tal er hex eller ej, uden at bruge den her metode. Eksempelvis kan tallet 33 forstås som både hex og decimal.

Hvis tallet ikke indeholder bogstaver kører koden mellem linje 9 og 12. Denne kode tjekker igen om variabelen *isHex* er sand, og kører funktionen *updateHexToggle()* hvis den er. Det tjekker blot om knappen allerede er sat til hex og skifter den tilbage til decimal, hvis teksten der bliver indsat er en decimal værdi.

## 4.5 AJAX Call

For at frontenden kan kommunikere med backenden, bruger vi et AJAX kald.

---

```
1  function factorize() {
2      let num = $("#num").val();
3      ...
4      $.ajax({
5          url: "/api/GetFactors",
6          type: "POST",
7          data: {
8              "inputNum": num,
9          },
10         success: function(data) {
11             if (data[0] == -2 || data[1] == -2) {
12                 showSnackbar("FactorDB is currently offline", 0)
13                 $("#p").val("");
14                 $("#q").val("");
15                 return;
16             }
17             if (data[0] == -1 || data[1] == -1) {
18                 showSnackbar("Could not factorize", 1)
19                 $("#p").val("");
20                 $("#q").val("");
21                 return;
22             }
23             $("#p").val(data[0]);
24             $("#q").val(data[1]);
25         }
26     });
27 }
```

---

Listing 2: Views/Home/Index.cshtml, Linje 82 – 127

Vi laver en funktion som indeholder AJAX kaldet, for nemmere at kunne sende en forespørgsel til backenden. I linje 2, laver vi en variabel, som indeholder dataen fra det første tekstfelt. Under linje 2, håndterer vi dataen og tjekker om den overholder forskellige kriterier, men for simplicitet, har vi valgt at undlade denne del her. Vi tjekker blot om dataen er noget serveren kan håndtere, og hvis den ikke kan, modtager brugeren en fejlbesked. Det betyder også at dataen er i et gyldigt format efter dette punkt og når den skal sendes via AJAX kaldet.

Fra linje 4 begynder AJAX kaldet. Mellem linje 5 og 9 specificerer vi hvilken data, der skal sendes hvorhen og hvordan den skal sendes. Dataen sendes til `/api/GetFactors`, som er endpointet for backenden, hvilket betyder at det er stedet hvor backenden og frontend kan kommunikere. Den sendes med et POST forespørgsel, hvilket er en type forespørgsel, man kan sende data med. Dataen sendes som et JavaScript objekt, hvilket også er derfor det ikke nødvendigt at sende en 'content-type' med. Der findes også andre måder at sende dataen på, hvor man ikke bruger et JavaScript objekt, men det her er klart den bedste måde.

Koden mellem linje 10–24, kører hvis funktionen ikke fejler og får en respons fra serveren. Dataen der bliver sendt tilbage har formatet `[x, y]`. Hvor `x` og `y` begge er strings, der indeholder tal. Vi er nødt til at bruge strings, da tallene er for store til at være i en JavaScript int. Første if-sætning tjekker om dataen er '-2', hvilket betyder at serveren ikke kunde forbinde til databasen. Hvis dette er tilfældet viser den en lille fejlbesked og fjerner en eventuel værdi fra `p` og `q` felterne, samt returner null, for at stoppe koden fra at køre. Anden if-sætning tjekker om dataen er '-1', hvilket betyder at der er sket en generel fejl på serveren. Hvis der er sket en generel fejl, viser den igen en lille fejlbesked og tømmer felterne, samt returnerer null for at stoppe koden. Hvis der ikke er sket nogle fejl, indeholder data-arrayet to tal, som indsættes i felterne `p` og `q`, på linje 23–24.

## 5 Backend

Backend er den del af hjemmesiden, som ikke bliver vist til brugeren. Det er her vi behandler dataen, som bliver sendt til os fra frontend. Vores backend er skrevet i C#, og bruger objektorienteret programmering, hvor vi laver klasser til at håndtere forskellige ting.

### 5.1 Primtalsfaktorisering

For at bryde RSA-algoritmen, skal man faktorisere det store tal `n`, som er produktet af to primtal. At faktorisere store tal er en meget svær opgave, som kræver meget tid og mange ressourcer. Det er derfor ikke muligt at faktorisere tallet, når brugeren anmoder om det. For at løse dette, findes der store databaser af faktorerede tal. Når en bruger indtaster et tal på hjemmesiden, sender vi tallet videre til en database, der hedder factorDB. FactorDB er den største database af faktorerede tal, og den er gratis at bruge. Hvis tallet findes i databasen, sender vi det tilbage til brugeren, og hvis det ikke findes, sender vi en fejlmeddelelse. Vores implementation af dette kan ses nedenfor.



---

```

1 public string[] GetFactors(){
2     string[] factors = { "-1", "-1" };
3     var url = $"http://factordb.com/api?query={_n}";
4     var request = WebRequest.Create(url);
5     request.Method = "GET";
6     var data = "";
7     ...
8
9     dynamic jsonData = JsonObject.Parse(data);
10    ...
11
12    for (var i = 0; i < 2; i++){
13        factors[i] = jsonData["factors"][i][0].ToString();
14    }
15
16    return factors;
17 }

```

---

Listing 3: Backend/RSACracker.cs, Linje 42

På linje 1 defineres en metode, der hedder *GetFactors()*, som returnerer et array af strings. Et array er en datastruktur der kan indeholde flere værdier af samme type. I dette tilfælde indeholder arrayet to strings, som er de to faktorer der kommer af at faktorisere *n*. På linje 2 initialiseres arrayet med to strings, som er sat til -1. {-1, -1} er vores default værdi, som vi bruger til at tjekke om vi har fået en faktorisering fra factorDB. På linje 3 defineres en variabel, der hedder *url*, som er en streng af tekst, der indeholder den url vi skal sende vores API forespørgsel til. På linje 4 defineres endnu en variabel, der hedder *request*, som er af typen *WebRequest*.

*WebRequest* er en klasse, der indeholder metoder til at sende forespørgsler til en server. I dette tilfælde, bruger vi typen af forespørgslen, der hedder 'GET', som er den mest almindelige type af forespørgsler. Med en 'GET' forespørgsel, kan vi hente data fra en server, uden at ændre noget på den, hvilket er det vi ønsker i dette tilfælde. På linje 5 definerer vi en ny variabel, af typen string, som hedder *data*. Der er her vores respons fra factorDB, skal ligge. Derefter kommer der en masse kode, som ikke er vigtig for at forstå hvordan vi bruger API'en. På linje 11 definerer vi endnu en variabel, der hedder *jsonData*, som er af typen *JsonObject*, hvilket lader os håndtere JSON data, som vi får fra factorDB.

Derefter kommer der lidt kode, som undersøger om vi får en gyldig respons, dette er ikke nødvendigt at gennemgå. På linje 12 laver vi en for-løkke, som kører to gange. Første gang den kører, sætter vi variabelen *i* til 0, og den anden gang sætter vi den til 1. Det er her vi henter ændrer værdierne i arrayet, som vi definerede på linje 2. Det gør vi ved at læse JSON dataen, og sætte værdierne ind i arrayet til de faktorer vi får fra factorDB. Til sidst returnerer vi arrayet, som indeholder de faktorer vi har fået fra factorDB.

## 5.2 RSA Funktionalitet

For at kryptere og dekryptere RSA, skal vi bruge lidt kode. Da vi bruger objektorienteret programmering, har vi lavet nogle klasser, en af disse hedder *RSADecrypter*, som indeholder metoder til at dekryptere. Hele klassen kan ses her:

---

```
1 public class RSADecrypter{
2     private BigInteger _n;
3     private BigInteger _d;
4     private BigInteger _c;
5     private BigInteger _m;
6
7     public RSADecrypter(string n, string d){
8         _n = BigInteger.Parse(n);
9         _d = BigInteger.Parse(d);
10    }
11
12    public string Decrypt(string c){
13        _c = BigInteger.Parse(c);
14        _m = BigInteger.ModPow(_c, _d, _n);
15        return Convert.ToString(_m);
16    }
17 }
```

---

Listing 4: Backend/RSADecrypter.cs

I denne klasse, har vi fire private variable, som er af typen *BigInteger*. *BigInteger* er en klasse, der kan håndtere meget store tal, hvilket vi har brug for, når vi arbejder med RSA. På linje 7 definerer vi en constructor-metode, som er den metode der kaldes, når man initialiserer et objekt af typen *RSADecrypter*. Denne metode tager to parametre, som begge er tal, men er af typen string, da vi ikke kan håndtere tal af typen *BigInteger* i frontenden.

På linje 8 og 9, gemmer vi de to parametre som private variable, som vi kan bruge i andre metoder i klassen. På linje 12, defineres metoden *Decrypt()*, som tager et tal af typen string, og returnerer en string. Her tager vi også tal ind som string, for at kunne håndtere det i frontenden. På linje 13, gemmer vi det indkommende tal som en privat variabel, og derefter laver vi en udregning på linje 14. Denne udregning blev gennemgået i under-sektion 2.1, og er sådan man dekrypterer RSA. Til sidst returnerer vi resultatet af udregningen som typen string, for at undgå problemer i frontenden.

## 6 Test af programmet

### 6.1 Test af kryptering

For at teste om vores program virker, har vi lavet en test, som krypterer begge vores navne. Vi starter med at generere en nøgle, som vi kan bruge til at kryptere vores navne. Derefter indsætter vi nøglen på krypteringssiden, og krypterer vores besked. Dette giver et stort tal, som enten kan vises som normalt heltal, eller som et hexadecimalt tal.

N

1023558936867327152922630785295438114647757107454885805680245910879035465319553615109

10 16

e

65537

m

Daniel Nettelfield og Gustav Nybro

ASCII

Decimal

HEX

Encrypt

1ecc934745c85dad770a3d001d8958ae7b6203735af03f5267090c67660f6163313d

10 16

### 6.2 Test af faktorisering

For at undersøge om vi kan faktorisere et tal, har vi valgt at lave en svag nøgle, så det er muligt at faktorisere den. Hvis man indsætter det tal vi brugte til at kryptere, på vores faktoriseringsside, får man følgende resultat:

1023558936867327152922630785295438114647757107454885805680245910879035465319553615109

10 16

Factorize

p 738710808158043234677781847846211780920019

q 1385601680066852604422563689357373717447111

Send to RSA Cracker

Det betyder at vi har fundet de to tal, der kan ganges sammen og give det tal vi startede med. Med den information kan man bryde RSA krypteringen, og få beskeden ud.

### 6.3 Test af RSA-brydning

For at teste om vi kan bryde RSA krypteringen, har vi brugt de to faktorer vi fik fra faktoriseringen, sammen med den nøgle vi brugte til at kryptere. Hvis man sætter dem ind på vores hjemmeside, sammen med den krypterede tekst, får man dette resultat:

C

1ecc934745c85dadcd770a3d001d8958ae7b6203735af03f5267090c67660f6163313d

1016

Crack

Input

Output

p

738710808158043234677781847846211780920019

N

10235589368673271529226307852954381146477571074548858f

q

1385601680066852604422563689357373717447111

d

94087707758515631892698851638849929567610210185769042f


e

65537

m

44616e69656c204e657474656c6669656c64206f67204775737461;

Daniel Nettelfield og Gustav Nybro



Her får vi den originale besked ud, hvilket betyder at vi har brudt RSA krypteringen, og bestået alle vores tests.

## 7 Konklusion

Vi har i dette projekt arbejdet med at bryde RSA kryptering ved hjælp af faktorisering. Vi har lavet en hjemmeside, som kan bruges til at kryptere og dekryptere RSA, og til at bryde svag RSA kryptering. Vi har været i stand til faktorisere store tal, ved hjælp af kald til en database, som vi kan tilgå ved at bruge en API, udviklet af factorDB, som er den største offentlige faktoriseringsdatabase. Vi har også udført en række tests, som viser at vores hjemmeside virker, og at vi kan leve op til vores krav om at bryde RSA krypteringen.

## 8 Billag

## 8.1 Kilde Kode - Zip Fil

RSA Crackstation.zip

## 8.2 Kilde Kode - Github

RSACrackstation - Github

### 8.3 URL - RSACrackstation

rsackrackstation.com