DEGREE PROJECT, IN COMPUTER SCIENCE , FIRST LEVEL
STOCKHOLM, SWEDEN 2015

FACT- and SAT-solvers on different types of semiprimes

LUDWIG SIDENMARK & ERIK V. KJELLBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY

CSC SCHOOL

KTH Computer Science
and Communication

FACT- and SAT-solvers on different types of semiprimes

LUDWIG SIDENMARK, ERIK V. KJELLBERG

Degree Project in Computer Science, DD143X
Supervisor: Per Austrin
Examiner: Örjan Ekeberg

CSC, KTH, 08/05/2015

Abstract

This thesis is aimed to cover how boolean satisfiability solvers can be used on integer factorization problems and to compare them with already established integer factorization solvers. The integer factorization problem is believed to be hard and is used in modern day cryptoalgorithms such as RSA. This thesis is also aimed to explore how well different solvers can solve different types of semiprimes and if there is any notable difference between them. This report covers three different boolean satisfiability solvers as well as three of the integer factorization solvers. The results from the thesis show that boolean satisfiability solvers can not be used as a replacement of already established integer factorization solvers. The thesis also shows that the type of semiprime does affect how fast the solvers are able to factorize the semiprime. The boolean satisfiability solvers had favorable results toward asymmetrical semiprimes and disfavorable results toward prime powers.

Referat

Denna avhandlings mål är att undersöka hur lösare för booleska satisfierbarhetsproblemet kan användas för att lösa primtalsfaktoriseringsproblem och jämföra dem med redan etablerade lösare för primtalsfaktorisering. Primtalsfaktorisering tros vara svårt och används i flera moderna krypteringsalgoritmer som RSA. Denna avhandling undersöker även hur väl olika lösare kan lösa olika typer av semiprimtal och om det finns någon noterbar skillnad mellan dem. Rapporten täcker tre olika lösare för booleska satisfierbarhetsproblem och tre olika primtalsfaktoriseringslösare. Resultaten från denna avhandling visar att lösare för booleska satisfierbarhetsproblem inte kan används som ersättning för redan etablerade primtalsfaktoriseringslösare. Avhandlingen visar även att typen av semiprimtal påverkar hur snabbt lösarna faktoriserar semiprimtalet. Lösarna för boolesk satisfierbarhet visade fördelaktiga resultat mot asymmetriska semiprimtal och ofördelaktiga resultat mot primtalspotenser.

Contents

Chapter 1

Introduction

The Integer Factorization problem (FACT), is a well known problem in the field of number theory. The problem is solved by for any given integer, finding a set of prime integers, whose product is the given integer. The exact computational class of the FACT-problem remains unknown and a method has not been found that is able to solve the problem in polynomial time. The problem is assumed to be hard to solve and this assumption is at the heart of widely used algorithms in the field of cryptography, such as RSA. The type of the FACT-problem that RSA uses and also the type which this project will investigate works in such a way that the given integer is a product of exactly two prime integers, a semiprime. There is an interest in finding a method that can solve FACT in polynomial time due to its relevance in cryptography. If someone would find an efficient method, then it would jeopardize all systems based on that type of cryptography, meaning that a need for new cryptographic methods would arise.

The boolean satisfiability problem (SAT), is a well documented problem known to be NP-complete. Methods of solving the SAT-problem has been researched extensively with competitions between different SAT-solvers in order to stimulate further research. Even though there is a possibility that SAT is harder to solve than FACT, more work has been put into researching SAT. There is a possibility that advances in the area of SAT may be used to find a way to quickly factorize big integers if there is a suitable reduction from FACT to SAT. A common method of reducing a FACT into SAT is to construct a boolean circuit for calculating the product of two prime integers.

1.1 Thesis goal and motivation

This thesis aims to explore and find an answer to the following questions.

1. **How effectively do SAT-solvers solve the FACT-problem for big integers compared to specialized FACT-solvers and classical algorithms such as Trial division and Fermat's factorization method?**

If there is to be any future in solving FACT with SAT-solvers the computational time should at least be comparable to the other already existing methods, the aim of this thesis is to seek if there is such potential.

2. **How do the SAT-solvers perform depending on how the semiprimes are built?**

   The classical algorithms has well documented characteristics depending on how the semiprime is built. If SAT-solvers turn out to be faster for a specific type of semiprime then SAT-solvers could be used more effectively by knowing in which cases they will perform better.

1.2 Limitation of scope

Due to time constraints the following limitations have been chosen.

1. **No in-depth analysis of solvers.**

   Instead of delving into the advantages and difference of different SAT-solvers in respect to the FACT-problem we have based our choices of SAT-solver as explained in 3.1.1.

2. **No in-depth analysis of reductions.**

   The purpose of this thesis is to find noticeable differences between the different semiprimes solving times of SAT-solvers, and thus, in-depth analysis of the reductions are out of scope.

1.3 Hypothesis

How effectively does SAT-solvers solve the FACT-problem for big integers compared to specialized FACT-solvers and classical algorithms such as trial division and Fermat's factorization method?
We believe that the SAT-solvers are strictly computationally slower than the specialized FACT-solvers, they will however, be able perform similar results as the more simple and classical algorithms especially for bigger integers.

How do the SAT-solvers perform depending on how the semiprimes are built?
Our hypothesis is that for semiprimes built by close primes are equally hard to solve as semiprimes built by primes not as close to each other for SAT-solvers, given that the semiprimes are of close bitsize.

Chapter 2

Background

2.1 Integer factorization

This section provides the necessary information of the FACT-problem. For additional information refer to [1].

2.1.1 Fundamentals of the factorization problem

The fundamental theorem of arithmetic is a classic theorem that states that every integer greater than 1 is either a prime or the product of prime numbers that can be written in one and only one way. Even though the order of the primes is arbitrary the primes themselves are not.

$$\mathrm{n} = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k} = \prod_{i=1}^{k} p_i^k$$

Where $p_1 < p_2 < \dots < p_k$ and are prime numbers, and $a_i \in N$.

Proof for this theorem is not included but can be found in [1]. The FACT problem is based around this theorem and the goal is to find these primes for any integer.

The specific type of the FACT problem that we are investigating is finding the factors for semiprimes. A semiprime is a natural number that is the product of exactly 2 distinct prime numbers. These semiprimes are considered to be the hardest to factorize and this property is the foundation of the RSA cryptosystem. If a method for solving FACT for semiprimes is found it would lead to an exploitable weakness. Therefore it is vital for RSA to choose a semiprime that is hard enough to factorize so that it would take infeasible time for any potential attacker to try.[2]

2.1.2 How to solve integer factorization

There are several special-purpose algorithms for solving FACT and the running time of the algorithms depends on the properties of the number to be factored or one of the hidden properties such as size or special form.[3] This thesis will cover two of these special-purpose algorithms which has different characteristics as well as the most popular general-purpose algorithm with a running time that only depends on the size of the integer being factored.[3]

1. Trial division
2. Fermat's factorization method
3. General number field sieve

The purpose behind these choices is to show that not all integers are equally hard to solve and also to compare the results of the SAT-solvers with these well-documented algorithms to see if the SAT-solvers have any similar characteristics.

Trial division
Trial division is the most computationally intense but also the simplest of the factorization methods. The essential idea of the method is to see if the integer to be factored n, can be divided by numbers in turn that is less than n. The pseudocode looks as in *Function 2.1.*

```
function TrialDivision(N)
    for $i$ $from$ 2 $to$ $\sqrt{N}$ do
        if $N/i$ then
            return i, N/i
        end if
    end for
end function
```

Function 2.1 Trial division pseudocode

It is preferable to start from the smallest primes since a random n is more likely to be divisible by two rather than three etc.. The method also only tests primes that are not bigger than $\sqrt{n}$ since if n is divisible by some number p where $n = pq$ and q is smaller than p then the method would have found q earlier than p and the method only needs to find one to calculate the other prime factor in this case.

This method has the specific characteristic that it is able to find the factors quicker for smaller integers. This means that small n are very weak to trial division. The method is also much more likely to find a solution for bigger semiprimes n when one of p or q is much smaller than the other.[4]

Fermat's factorization method
The essence of this method in order to find the factors lies in representing the odd integer n as the difference of two squares.

$$n = a^2 - b^2$$

That difference is algebraically factorizable as $(a - b)(b + a)$ and if neither of these factors equals 1 this is a legitimate factorization of n. The pseudocode for this method is in *Function 2.2.*

```
function FermatFactor(N)
    for $x\ from\ cieling(\sqrt{N})\ to\ N$ do
        ySquared = x * x - N
        if $isSquared(ySquare)$ then
            y = $\sqrt{ySquared}$
            s = x - y
            t = x + y
            if $s \neq 1 \wedge s \neq N$ then
                return s, t
            end if
        end if
    end for
end function
```

Function 2.2 Fermat's factorization method pseudocode

This method has the characteristic to be able to find the factors of n very effectively if their values are close to each other.[5] Therefore it is important for cryptosystems such as RSA to not have their semiprimes have factors that are too close to each other or Fermat's would be able to break it easily.

General number field sieve
Explaining the General number field sieve (GNFS) method is beyond the scope of this thesis. The interested reader can find more about the algorithm in [6]. It is however the most popular method to solving the FACT problem for big integers without special properties which can be exploited by special-purpose algorithms. An implementation of this method currently holds the record for the factorization of the largest integer, after the successful factorization of the RSA-768.[7]

2.2 Boolean satisfiability problem

The boolean satisfiability problem (SAT) uses the values true (T) and false (F) along with logic operators such as OR ($\vee$), AND ($\wedge$) and the negation of expressions with NOT ($\neg$) creating literals that need to be satisfied in order to find a solution to the problem. The SAT problem is a known NP-Complete problem, meaning it is at least as hard to solve a SAT problem as any other NP-Complete problem. A very simple SAT-instance that which can be solved by SAT-solvers looks like:

$(x_1 \vee x_2) \wedge (\neg x_3)$ with a possible solution $x_1 = T, x_2 = F, x_3 = F$

More about SAT can be found in [8].

2.2.1 Conjunctive normal form

In boolean logic Conjunctive normal form (CNF) is a special way to write logic expressions in and consists of three main parts.

- A literal l that is either an atom l or the negation of that atom $\neg l$
- A clause c that contains literals separated by $\vee$ operators (disjunction). For example: $c = (l_1 \vee l_2 \vee l_3)$.
- A formula f that contains clauses separated by $\wedge$ operators (conjunction). For example: $f = c_1 \wedge c_2 \wedge c_3$

The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) has proposed a standard format for CNF which is widely used today including the SAT-solvers which will be examined in this thesis. [9]

2.2.2 How to solve SAT

This section will cover some of the more common methods that established SAT-solvers use in order to solve a SAT-problem. Additional information about the methods can be found in [10].

Backtracking
SAT-solvers use backtracking in order to jump back and try a new solution if a conflict has arisen. A conflict happens when decisions taken by the SAT-solver results in a clause to be false. Normal backtracking algorithms back up only one step, however the cause to the conflict may be higher up in the decision tree. Therefore modern SAT-solvers use more evolved backtracking algorithms that first examines the cause of the conflict, and then jump to the found cause and change it. This is called non-chronological jumping.

Conflict driven clause learning
If the SAT-solver would know if a decision would lead to a conflict it would never had taken it. Therefore when the solver finds the cause of the conflict with the help of the non-chronological jumping it will create another clause that will represent the conflict. As such it will not make the same mistake additional times.

Restart
The solver may be able to try to restart in order to be able to make better decisions. The clauses that are added from the Conflict driven clause learning are not

discarded on a restart but instead used so the SAT-solver will make better decisions.

Simplifier
Most solvers has the option of using preprocessing in order to simplify the problem before it starts try to solve it. The method for doing this is different depending on the solver.

2.3 Reduction from FACT to SAT

One popular way of reducing the FACT-problem into a SAT-problem is by creating a digital circuit that has the form of a CNF input. Digital circuits have the values of 0 and 1 corresponding to the false and true values in propositional logic. They also have gates comparable to the propositional logic operators. First step is creating a circuit $C_n = (a_1 \dots a_l b_1 \dots b_m)$ where a and b are potential factors of n. l and m are the bit-length of a and b respectively. The circuit is built in such a way that $C_n = (a_1 \dots a_l b_1 \dots b_m) = 1$ if and only if $a \times b = n$. The remaining task is to find a and b, whose product is n. After creating the digital circuit it must be reduced into CNF with the help of Tseitin transformation so that SAT-solvers can use it and solve the problem. More information about the Tseitin transformation can be found in [11].

Chapter 3

Method

The goal of this thesis is to find out how well SAT-solvers find the factors of different semiprimes. The questions that need to be answered is how well they find the factors for different sizes of the semiprimes, also how well they find the factors for different types of semiprimes. Because of the fact that it is hard to predict how the SAT-solver will behave an empirical approach was decided. The limiting factor for the data collection was decided to be CPU time.

3.1 Solvers

3.1.1 SAT

The SAT Competition provides a ranking from modern SAT-solvers and as a result we chose some of the sequential solvers that had achieved steady high ranks in the competition. The number of SAT-solvers was decided to be tested in such a way it would take a reasonable time for the scope of this thesis to run all the tests. Therefore the following SAT-solvers were chosen.

- MiniSat 2.2.0 with simplifier
- LingeLing - ayv - 86bf266-140429
- Glucose 3.0 with simplifier

All solvers can be found in [12].

3.1.2 FACT

Implementing our own version of the General number field sieve is out of scope for this thesis. An already existing implementation that is well regarded and suitable in size for our needs was thus chosen. Therefore the "MSieve FACT-solver" was chosen which can be found in [13].

3.2 Reduction

The reduction used in this thesis was implemented by [14]. The type of reduction that was chosen for this thesis was the *n-bit adder* and the *carry-save multiplier*. The choice stems from the results of [15].

3.3 Instances

As well as different bit-sizes of the semiprimes we also have to take into consideration how the different semi primes are built. We have therefore chosen the following type of semiprimes.

- Safe same size semiprimes, semiprimes with its factors in similar size but not too close for efficient Fermat.
- Prime power, semiprimes who are a power of a single prime number, *e.g.* $7^2 = 49$.
- Unsafe same size semiprimes, semiprimes with its factors in similar size and also close enough for efficient Fermat.
- Asymmetrical size semiprimes, semiprimes which has one factor that is considerably smaller than the other.

For each bit-size and type a total of 100 semiprimes were randomly generated. In total 2000 semiprimes were generated and tested. The same semiprimes were used for all the different solvers. The code for the generator written in java can be found in appendix D. All generated instances together with code handling them can be found in appendix C.

3.4 Measurables

3.4.1 CPU-time

To measure the usage of the time it takes the Central Processing Unit (CPU) to solve any of the given will give a good measurement for comparing the different solvers. As long as the tests are not interrupted when running it will give a result that is accurate enough for the purpose of this thesis.

For the SAT-solvers as well as the General Number Field Sieve implementation their built in time measurement tool was used in order to measure the CPU-time. For Trial Division and Fermat's factorization method the java library's *system.nanotime* was used.

3.5 Environment

The tests were run on an ubuntu system of version 12.04 LTS with a 64-bit OS type. More information about the hardware specifications can be found in appendix B.

Chapter 4

Results

This chapter consists of the results from the tests performed as stated in chapter 3.

4.1 Overview

This section contains an overview of the results gathered from the tests. Some key points.

- Trial Division is the fastest method concerning CPU-Time.
- Lingeling is the overall fastest SAT-solver while the slowest was Glucose.
- Tested semiprimes with bit-sizes larger than 40 would lead to processes longer than 3600 seconds for the SAT-solvers which led to testers aborting the process.

Bit-Size	Glucose	MiniSat	Lingeling	Trial Division	Fermat
25	0,25202	0,17401	0,56600	0,00038	0,00629
30	1,38809	1,13207	1,54200	0,00169	0,16064
35	11,71270	9,80661	9,10500	0,01024	*0,00047
40	68,43430	53,98540	50,26300	0,05700	*0,00041
Total	81,78711	65,09809	61,47600	0,06930	*0,16782

Table 4.1. Solvers total median time in seconds and split in bit-size of semiprimes.
* method was not able to solve all semiprimes of this size within reasonable time.

4.2 SAT-solvers

The following section contains the results from the experiment for all of the SAT-solvers.

4.2.1 Glucose with simplifier

Glucose					
Semiprime Size	Safe Same Size (s)	Unsafe Same Size (s)	Prime Power (s)	Asymmetrical Size (s)	Total (s)
25	0,2860	0,2160	0,2300	0,2600	0,2520
30	2,1481	1,5341	1,9501	0,6040	1,3881
35	16,6371	10,4847	17,1711	6,9064	11,7127
40	95,6460	78,9990	118,9650	13,3549	68,4343

Table 4.2. Glucose's median time in seconds split by size and by type, an exponential growth in solving time can be appreciated in most semiprimes.

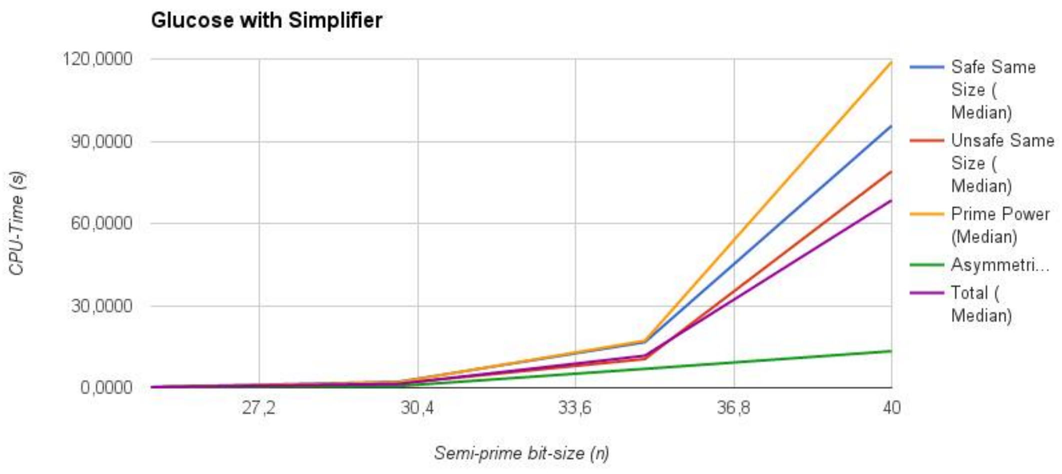

Figure 4.1. Values from **Table 4.2** presented in a line chart comparing different types of semiprimes. Results showing an exponential growth in CPU-time in all types of semiprimes except for asymmetrical size semiprimes, which shows an almost linear growth.

Glucose Spread				
Bit-Size	25	30	35	40
Type	s	s	s	s
Total	0,1890	1,2617	10,3873	72,3933
Safe Same Size	0,1895	1,2454	11,0341	72,5399
Unsafe Same Size	0,1687	1,2138	9,3184	63,7401
Prime Power	0,1961	1,3526	11,2437	77,5616
Asymmetrical Size	0,1925	0,7818	6,3207	31,1230

Table 4.3. Standard deviation split by semiprime type and bit-size, with 'total' showing the standard deviation for all semiprimes of given size.

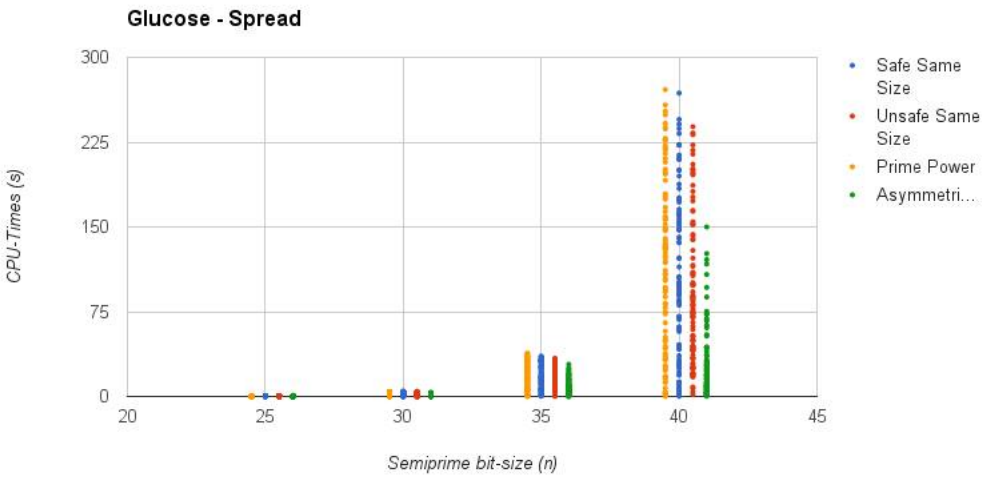

Figure 4.2. Showing the spread of the solver by type and bit-size, showing every single instance's required CPU-Time, lowest growth and solving time for asymmetrical size semiprimes.

4.2.2 MiniSat with simplifier

MiniSat					
Semiprime Size	Safe Same Size (s)	Unsafe Same Size (s)	Prime Power (s)	Asymmetrical Size (s)	Total (s)
25	0,2060	0,1600	0,2240	0,1260	0,1740
30	1,1781	1,1761	1,2881	0,7480	1,1321
35	15,1490	10,1866	13,7049	3,7262	9,8066
40	72,6625	48,8511	76,6108	14,2349	53,9854

Table 4.4. MiniSat's median time in seconds split by size and by type. Exponential growth can be appreciated, with the lowest for asymmetrical size semiprimes.

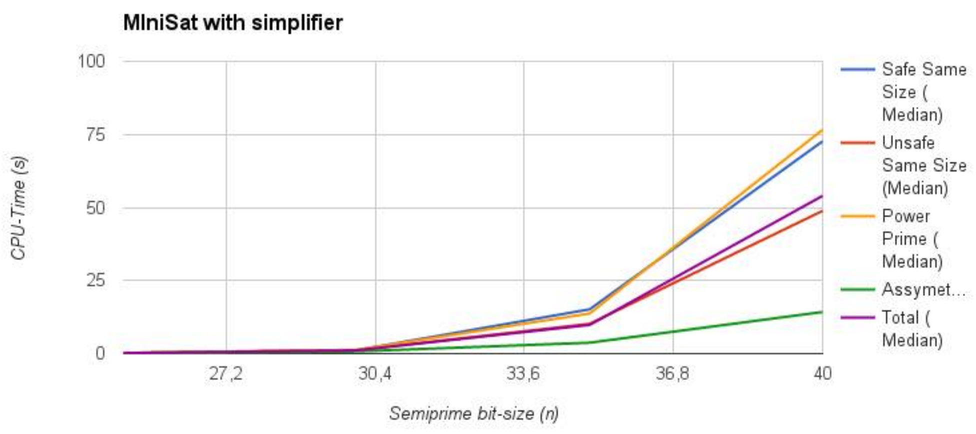

Figure 4.3. Values from **Table 4.4** presented in a line chart comparing different types of semiprimes. While all types are of exponential size the Assymetrical is the one with the least growth.

MiniSat Spread				
Bit-Size	25	30	35	40
Type	s	s	s	s
Total	0,1341	0,8807	9,4556	62,1590
Safe Same Size	0,1458	0,7601	10,0801	68,0404
Unsafe Same Size	0,1151	0,9417	8,4717	51,9426
Prime Power	0,1407	0,9654	9,5821	61,6294
Asymmetrical Size	0,1230	0,7893	7,2080	52,2828

Table 4.5. Standard deviation split by semiprime type and bit-size

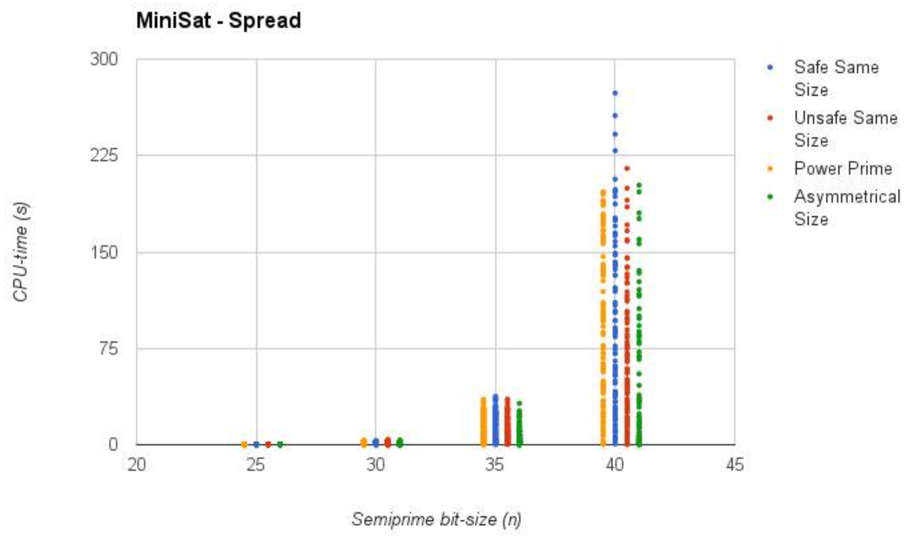

Figure 4.4. Showing the spread of the solver by type and bit-size, showing every single instance's required CPU-Time. All four similar intervals in its spread with prime power with the largest amount over 200 seconds and safe same size with the biggest outliers.

4.2.3 Lingeling

Lingeling					
Semiprime Size	Safe Same Size (s)	Unsafe Same Size (s)	Prime Power (s)	Asymmetrical Size (s)	Total (s)
25	0,6120	0,3560	0,6100	0,5060	0,5660
30	1,7620	1,3360	1,7340	1,5000	1,5420
35	10,1270	6,6800	10,4170	8,4030	9,1050
40	57,0095	43,9950	60,3620	40,4590	50,2630

Table 4.6. Lingeling's median time in seconds split by size and by type.

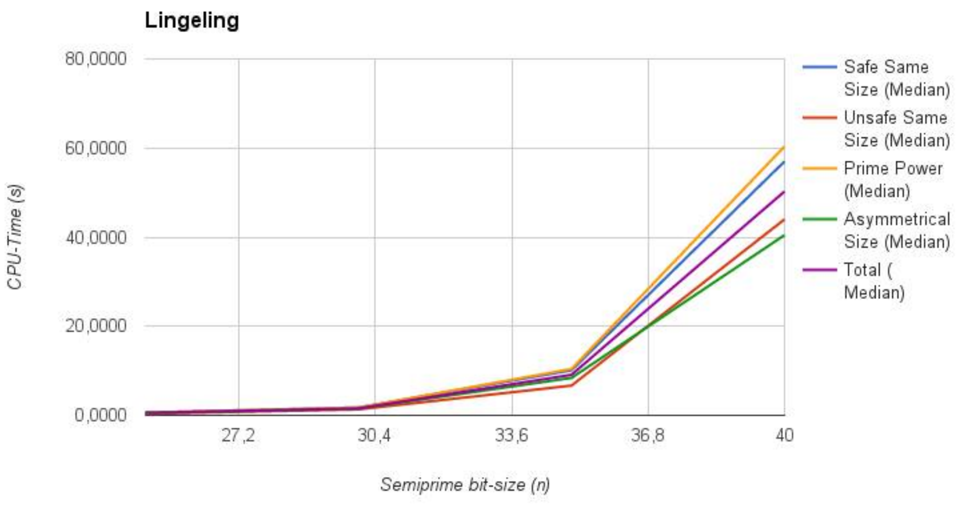

Figure 4.5. Values from **Table 4.6** presented in a line chart comparing different types of semiprimes. All types are of exponential and similar growth.

Lingeling Spread				
Bit-Size	25	30	35	40
Type	s	s	s	s
Total	0,2485	0,6957	5,2953	36,0882
Safe Same Size	0,2336	0,6897	5,6592	34,7984
Unsafe Same Size	0,2179	0,5983	4,7056	28,1889
Prime Power	0,2814	0,7037	5,3190	40,3248
Asymmetrical Size	0,2450	0,7346	4,8517	37,0476

Table 4.7. Standard deviation split by semiprime type and bit-size.

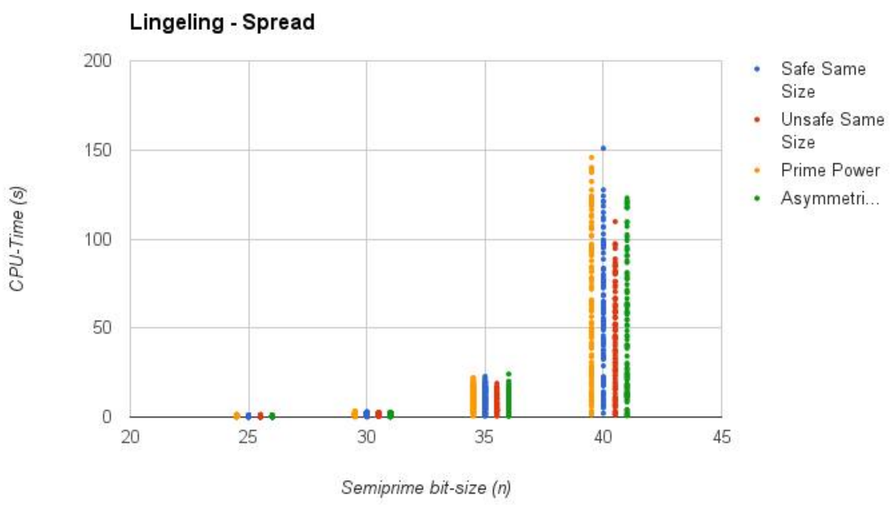

Figure 4.6. Showing the spread of the solver by type and bit-size, showing every single instance's required CPU-Time. All four have similar spread throughout the different bit-sizes but power primes with the most instances with over 100 seconds in CPU-Time.

4.3 FACT-solvers

The following sections contains the results of both the special-purpose and general-purpose factorization methods.

4.3.1 Trial division

Trial Division					
Semiprime Size	Safe Same Size (s)	Unsafe Same Size (s)	Prime Power (s)	Asymmetrical Size (s)	Total (s)
25	0,00037	0,00068	0,00045	0,00012	0,00038
30	0,00044	0,00209	0,00209	0,00006	0,00169
35	0,00238	0,01297	0,01286	0,00024	0,01024
40	0,00468	0,07093	0,07043	0,00007	0,05700

Table 4.8. Trial division's median time in seconds split by size and by type. Prime powers were clearly harder to solve for Trial division while asymmetrical size semiprimes were quickly decomposed.

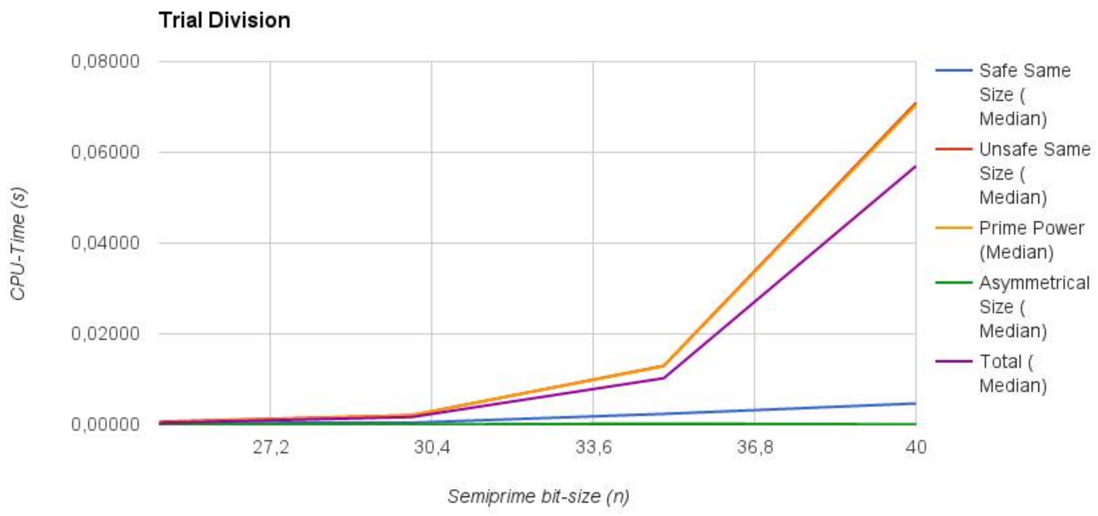

Figure 4.7. Values from **Table 4.8** presented in a line chart comparing different types of semiprimes.

Trial Division Spread				
Bit-Size	25	30	35	40
Type	s	s	s	s
Total	0,0025	0,0053	0,0110	0,0363
Safe Same Size	0,0018	0,0027	0,0077	0,0092
Unsafe Same Size	0,0029	0,0066	0,0113	0,0145
Prime Power	0,0034	0,0073	0,0108	0,0157
Asymmetrical Size	0,0003	0,0004	0,0003	0,0003

Table 4.9. Standard deviation split by semiprime type and bit-size.

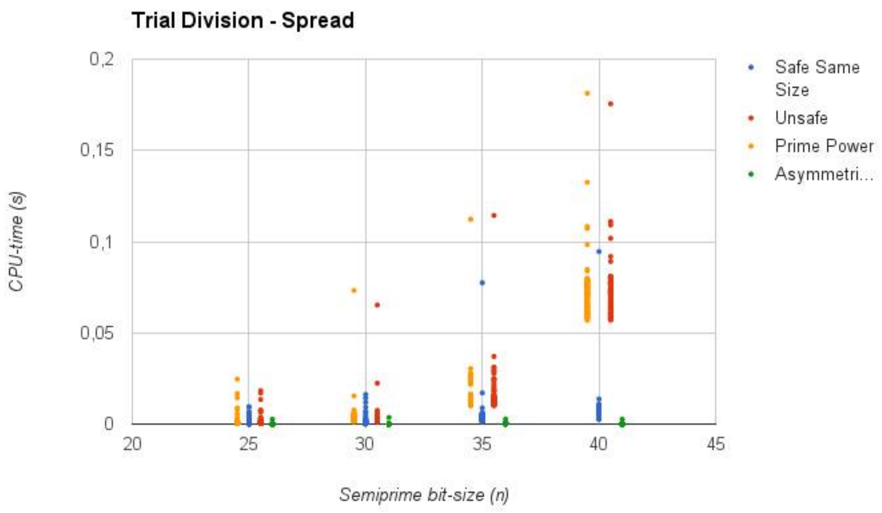

Figure 4.8. Showing the spread of the solver by type and bit-size, showing every single instance's required CPU-Time. The spread is dependent of how big the smallest prime-number is.

4.3.2 Fermat's factorization method

Fermat					
Semiprime Size	Safe Same Size (s)	Unsafe Same Size (s)	Prime Power (s)	Asymmetrical Size (s)	Total (s)
25	0,01655	0,00087	0,00020	3,07059	0,00629
30	0,58481	0,00054	0,00022	539,01524	0,16064
35	2,47466	0,00047	0,00024	N/A	0,00047
40	63,05384	0,00041	0,00027	N/A	0,00041

Table 4.10. Fermat's mean time in seconds split by size and by type. N/A stands for processes that took longer than 3600 seconds and were killed.

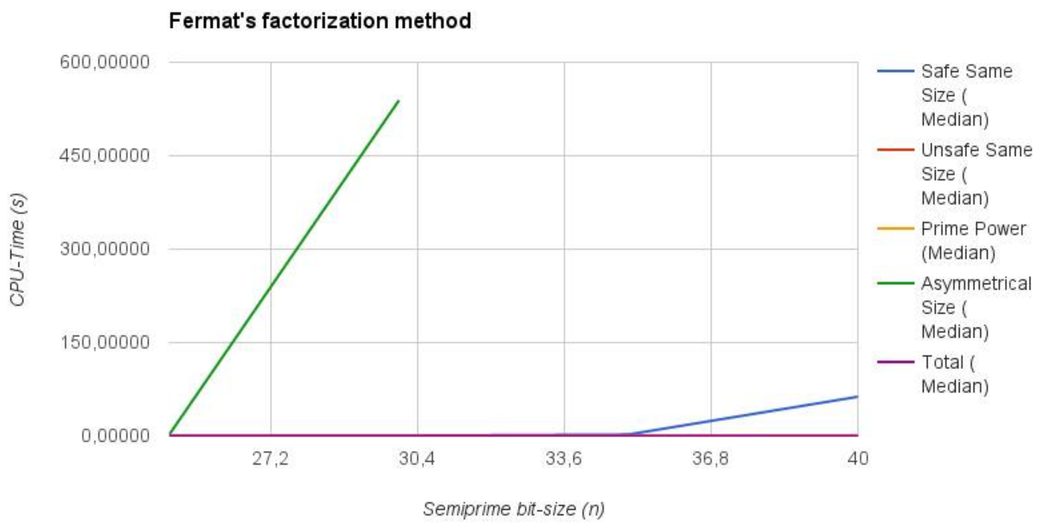

Figure 4.9. Values from **Table 4.10** presented in a line chart comparing different types of semiprimes. The unsafe and prime power are of almost constant CPU-Time, while for the asymmetrical, its worst case type it is not able to solve semiprimes over the bit-size of 30 in a reasonable amount of time.

Fermat Spread				
Bit-Size	25	30	35	40
Type	s	s	s	s
Total	1,36414	231,21183	1,14628	29,59586
Safe Same Size	0,01260	0,13732	0,46928	9,08551
Unsafe Same Size	0,00126	0,00059	0,00032	0,00028
Prime Power	0,00039	0,00039	0,00027	0,00027
Asymmetrical Size	0,54629	83,83570	N/A	N/A

Table 4.11. Standard deviation split by semiprime type and bit-size. N/A stands for processes that took longer than 3600 seconds and were killed.

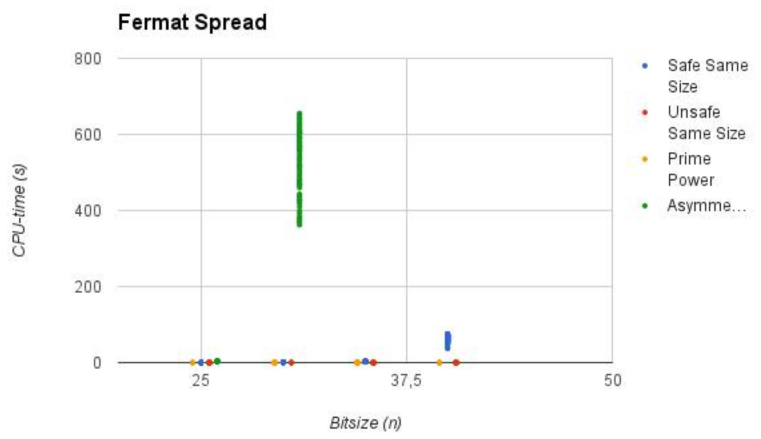

Figure 4.10. Showing the spread of the solver by type and bit-size, showing every single instance's required CPU-Time. The spread is negligible for Unsafe same size and Prime Power while apparent for the other two types, and a big increase for asymmetrical size semiprimes.

4.3.3 General number field sieve

After performing several test for semiprimes of bit-sizes up to 100 with CPU-times that were too low to give an accurate reading it was deemed that the method was not comparable to the rest as it had too low process times.

Chapter 5

Discussion

5.1 Environment

Efforts were put in order to make the environment for the tests as stable and reliable as possible. No additional programs that could have been controlled were running during the tests and the computer was not used for any other purposes. However due to the fact that the tests was performed on KTH's computers for students, the test could not be performed without an internet connection and some background processes that needed sudo access to kill and therefore were running during measurements. This may have impacted the results, however as all the different SAT-solvers and FACT-solvers had the same conditions it should only have a minor impact.

5.2 The experiment

In order to avoid problems relating to having a non-uniform environment for the tests, the semiprimes to be factorized should be of both random type and size. This however was not the case as the semiprimes running in each session was in both the same type and size. The results of this may be hidden within the results acquired.

Other factors that may impact the validity of the results are the choices of SAT-solvers. Since they were chosen by having a high ranking in the different SAT-competitions, and it may be that they are specifically designed to be able to solve a specific type of SAT. Therefore they may be ill suited for solving reduced FACT-problems. Another factor that may be of relevance is the different settings when running SAT-solvers that could be beneficial for solving problems of that specific kind faster. Due to our stated limitations of not being able to make an in-depth analysis of the solvers these settings were not taken into account and the solvers were used in their default settings.

5.3 Results

Results were consistent between FACT-solvers and SAT-solvers, showing that FACT-solvers are still more suitable to solve, when it comes to computational time, a factorization problem. The results of Trial division were convincingly the best computationally and also had the least spread of its results. One of the main problems that SAT-solvers face is their inconsistency for solving different semiprimes of the same size and type, as can be seen by the high standard deviation. The computationally fastest of the SAT-solvers, Lingeling, was the one with the smallest standard deviation leading it to not have as many or as big outliers as the other two solvers.

When comparing the results for different types of semiprimes some interesting points can be made. Our results for both Trial Division and Fermat's factorization algorithm conform with the previous established characteristics discussed in section 2.1.2. There were however some results concerning Trial Division that did not conform with the characteristics. One that stands out is shown in figure 4.8 were higher bit-sizes did not lead to more CPU-Time required for the asymmetrical size type. One of the reasons for such results is because the processes were so short it could not be guaranteed that they would be accurate enough.

When it comes to the SAT-solvers all three showed an inclination for being better suited at solving certain types of semiprimes. All three had the asymmetrical size as its fastest type which can be seen in *Figure 4.1*, *4.3* and *4.5* respectively. The difference was much more apparent for Minisat and Glucose than for Lingeling as Lingeling was more consistent overall than the other solvers for different types of semiprimes. The three solvers also had the prime power as its computationally hardest semiprime to factorize. The results show that the SAT-solvers generally require more CPU-Time to factorize semiprimes when the factors are close to each other. There are many possible reasons for this that requires an in-depth look of both solver and reduction to find, however, it is not dependant on the amount of clauses or literals as that amount is only dependant on the bitsize of the semiprime reduced.

5.4 Possible further studies

This thesis only explores one of the fundamentally different reductions from FACT to SAT and therefore it would be beneficial for future studies to expand to different kind of reductions. For example, does another reduction give more beneficial results for a different kind of semiprime compared to what this thesis found. Another possible angle is to test more SAT-Solvers in order to see if all of them abide to the results our tests have shown. In this thesis only the most polarizing of semiprimes were tested and it would be interesting to try and widen the spectrum and test different types of semiprimes. This includes testing more semiprimes of the types we have tested as this thesis only covered 2000 different semiprimes.

Chapter 6

Conclusion

6.1 Comparing FACT-solvers and SAT-solvers

As mentioned in the hypothesis in section 1.3 the GNFS algorithm was the computationally fastest. Trial division however, was computationally the second best and was the algorithm that would be able to factorize the biggest semiprimes of all types within reasonable time with the exception of GNFS. Lingeling was the best overall performing SAT-solver with also the smallest standard deviation. All of the SAT-solvers had problems with semiprimes with bit-size larger than 40.

6.2 Comparing different types of semiprimes

GNFS did not have any noticeable computational difference between the different types while the results for the two special-purpose algorithms conformed to already established characteristics. All of the SAT-solvers had favourable results for the asymmetrical-size semiprimes and disfavourable results for the prime power semiprimes.

Appendix A

Software

A.1 Libraries and compilers

A.1.1 Java and javac

java 1.6.0_34
OpenJDK Runtime Environment (IcedTea6 1.13.6) (6b34-1.13.6-1ubuntu0.12.04.1)
OpenJDK 64-Bit Server VM (build 23.25-bo1, mixed mode)

javac 1.6.0_34

A.1.2 GHC

GHCi version 7.4.1

A.1.3 GCC

gcc version 4.6.3
(Ubuntu/Linaro 4.6.3-1ubuntu5)

Appendix B

Hardware

CPU	Intel Core 2 Quad CPU Q9550 @ 2.83GHZ x 4
RAM	4 GB

Appendix C

Instances

The instances used as well as the code for handling the reductions written in java can be found in https://github.com/ludwan/SemiPrimeInstances. It also holds a modification of the code for reduction originally created by [14] written in haskell.

Appendix D

Source code

The applications for Trial Division, Fermat's Factorization method and the semiprime generator written in java can be found at: https://github.com/Elderkousom/FermatTrial-Generator.

Bibliography

[1] Long, Calvin T. Elementary introduction to number theory. 1987 3rd Ed. pp. 52-57

[2] Boneh, Dan. Twenty Years of attacks on the RSA Cryptosystem. Notices of the American Mathematical Society 46 (2) 1999: pp. 2-3.

[3] Bressoud, David M., and Wagon, S. A Course in Computational Number Theory. Key College Pub., in Cooperation with Springer, 2000. pp. 168-169.

[4] Crandall, Richard, and Pomerance, Carl B. Prime Numbers: A Computational Perspective. Springer New York. Web. pp. 116-120.

[5] Sherman Lehman, R. "Factoring Large Integers." Mathematics of Computation 28.126 1974. pp. 637-46. Web.

[6] Briggs, Matthew E. An introduction to the General Number Field Sieve. http://scholar.lib.vt.edu/theses/public/etd-32298-93111/materials/etd.pdf 2010. Accessed 2015-03-29.

[7] Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thome, E., Bos, J. W., Gaudry, P., Kruppa, A., Montgomery, P. L., Osvik, D. A., te Riele, H., Timofeev, A., and Zimmermann, P. Factorization of a 768-bit RSA modulus. 2010 version 1.4.

[8] Garey, Michael R. ,and Johnson, David S. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman. ISBN 0-7167-1045-5. A9.1: LO1 - LO7, 1979, pp. 259 - 260.

[9] SATComp Organizing committee. Submission format. http://www.satcompetition.org/2009/format-benchmarks2009.html 2009-01-13. Accessed 2015-05-07.

[10] Zhang, Lintao., Madigan, Conor F., Moskewicz, Matthew H., and Malik, Sharad "Efficient conflict driven learning in a boolean satisfiability solver". http://www.mimuw.edu.pl/ tsznuk/tmp/dpll.pdf 2001. Accessed 2015-04-18.

[11] Tseitin, G. S. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors. Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970, pp. 466–483. Springer, Berlin, Heidelberg, 1983.

[12] SATComp Organizing committee. Sat competitions. http://www.satcompetition.org/ Accessed 2015-04-16.

[13] Msieve source code. http://sourceforge.net/projects/msieve/ Accessed 2015-04-14.

[14] Purdom, Paul., and Sabry, Amr. Cnf generator for factoring problems. http://www.cs.indiana.edu/cgi-pub/sabry/cnf.html Accessed 2015-04-08.

[15] Asketorp, Johan. Attacking RSA moduli with SAT solvers. http://kth.diva-portal.org/smash/record.jsf?c=1searchType=SIMPLEquery=Attacking+RSA+moduli+with+SAT+solverslanguage=enpid=diva2%3A769846dswid=-4835 2014-12-09. Accessed 2014-02-22.