Master Thesis

Secure verification of computational power

Author(s):
Wagner, Rolf

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005797306

Rights / License:
In Copyright - Non-Commercial Use Permitted
Secure Verification of Computational Power

Master thesis

Supervised by
Prof. Dr. Srdjan Capkun
Ghassan Karame
System Security Group, ETH Zurich

Rolf Wagner

2009
Abstract

This thesis addresses the problem of efficiently verifying computational power. Nowadays, existing proposals for benchmarks implicitly push trust towards the hosts involved in the verification process. A malicious host can therefore abuse the operation of current benchmarks and trick a measuring entity into accepting false computational power capability. In this work, we demonstrate attacks on current benchmarks and we motivate for the need of secure un-parallelizable benchmark codes. More specifically, we investigate the properties of the "repeated squaring" benchmark and we show that it limits the advantage of malicious provers in faking their performance claims. We, further, propose a secure benchmark, based on repeated squaring that pushes parts of the computational verification overhead towards the prover, itself. We demonstrate the security and the performance of our proposed benchmark both analytically and by implementation over PlanetLab nodes.
Acknowledgments

This thesis was written in the course of my master studies at Swiss Federal Institute of Technology (ETH) Zurich. I am grateful to Prof. Dr. Srdjan Capkun who gave me the opportunity to write my master thesis in an open research field. Special thanks go to Ghassan Karame who co-supervised me during the progress of this work.
5.1 Modular squaring as benchmark code ........................................ 51
5.1.1 Fast modular squaring algorithms ........................................ 54
5.1.2 Parallelization .............................................................. 55
5.1.3 Montgomery ............................................................... 55
5.1.4 Special groups ............................................................. 57
5.2 Reduce verification effort ...................................................... 58
5.2.1 Use precomputed parameters ............................................. 58
5.2.2 Direct verifier optimizations .............................................. 60
5.3 Sketch of SecRaCP ............................................................ 64
5.3.1 Verifier Setup .............................................................. 65
5.3.2 Protocol .................................................................... 65
5.4 Security analysis ............................................................... 68
5.4.1 Resulting protocol assumptions ......................................... 68
5.4.2 Attacker model ............................................................ 68
5.4.3 Benchmark phase ......................................................... 68
5.4.4 Verification phase ......................................................... 73
5.5 Runtime analysis .............................................................. 77
5.6 Implementation of SecRaCP .................................................. 78
5.6.1 Setup ...................................................................... 78
5.6.2 Results ................................................................. 79
5.6.3 Interpretation ............................................................. 83

6 Conclusion .............................................................. 84
Chapter 1

Introduction

Faster and more reliable (Internet) connections animate to use more than just common net-
work services like e-mail. A more fundamental approach as just to provide specific services, is to provide computational power.

In this paper we treat the following problem: A customer wants to outsource a computa-
tional problem to one or several computer systems (nodes). He often has to decide between several nodes and wants to rank the available systems by theirs computational power abilities. Either because he has a time critical computation to do, or because he is interested in the price-performance ratio.

A customer (verifier) wants to verify (rank) several computer systems (nodes) by theirs computational power abilities.

There are two main categories of computational power service usage:

1. The customer pays for the computational resources as much he use it.
2. The customer can use the computational resources for free as a member.

Hence, there are two main motivations why a service provider may want to cheat:

1. Service providers want to be selected as extern computational resources, because they get an utilization fee. Problem here: How to be sure that service providers do not claim false (higher) computational power? If this happens, your calculation could be finished to late and cost you a lot of money or reputation.

2. Service providers want to avoid selection as extern computational resources, because they already got the flat-rate fee or are not paid anyway. Problem here: How to ensure that service providers do not claim lower computational power than he has available.

This thesis considers the problem of claiming higher computational power than available.
It is the more frequent situation in real life. Therefore, we rank all benchmarked nodes by their computational power they at least have (lower bound). To find the upper bound of a system’s available computational power, a trusted entity on the nodes is needed, because every node can slow down his own system. This can be for example a trusted platform module (TPM). However this will not be treated here.

To determine the minimal available computational power, we can run a benchmark on the remote host (prover). Nowadays benchmarks are trust based. Either the verifier has to believe the prover computational power claims or the verifier has to setup a trusted entity on the prover’s side. That may be possible if you benchmark a few near located computer systems. But not, if you benchmark a large number of arbitrary computer systems.

The idea is to measure the time a node needs to correctly respond to the benchmark request. The correct answer indicates also the correct run of the whole benchmark. If the verifier benchmarks a node because he wants to outsource a computation, the node will obviously have more available computational power.

It is important, that the verifier’s verification effort is lower than the prover’s benchmark effort.

The rest of this thesis report is organized as follows: In the first three chapters, a motivation to this problem is given. A Java simulation shows the impact of claiming wrong computational power. We also do some fundamental considerations about the problems of measuring computational power and why parallelization is an important aspect. After a short discussion about different measure methods, we also state fundamental cheating approaches. The insecurity of the already proposed benchmarks and how they can be cheated is explained in chapter four. We analyze every proposed uncheatable benchmark in detail. The underlying techniques are explained and a short summary of the proposed benchmark is given. In the weakness analysis we point out the cheating potential and propose a way to attack the benchmark. For the Gaussian Elimination and the Matrix Multiplication benchmark proposal, we implemented the benchmark and the according attack in Java. The results are presented by showing the advantage malicious nodes can gain. Next to the measured results from our lab environment, we interpolate the results for theoretical fiber optic and internal cluster communication. In chapter five, the techniques for a secure benchmark are developed and we propose a secure benchmark protocol. The rough idea of our benchmark was already proposed before. However, we detail and improve the idea and focus hereby on the possible presence of malicious nodes. Additionally we propose a new faster verification method where the verifier outsources parts of the verification to the prover. An extended security analysis is also part of this chapter. At last we present the results of the SecRaCP protocol implementation, where we benchmark several nodes of the PlanetLab distributed server platform.
Chapter 2

Related work

To the best of our knowledge, very little work has been done, related to this topic. "Time-lock puzzles and time-released Crypto" considers the advantages of repeated modular squaring as a un-parallelizable, computational workload, code. Two papers about uncheatable benchmarks were proposed and therefore analyzed in this thesis.

Time-lock puzzles and time-release Crypto

Rivest, Shamir and Wagner propose in "Time-lock puzzles and time-release Crypto"[2] repeated squaring as none parallelizable method to keep computer systems busy for a defined time. They use a variant of the Blum Blum Shub[3] number generator as we will do too, to define an unpredictable value. They use it to encrypt a message that can not be decrypted by anyone, until a pre-determined amount of time has passed. To decrypt the message, a pre-defined amount of modular squaring rounds have to be done. To map rounds of modular squaring to run time, the exact modular squaring performance of the decrypter must be known. Actually, they introduce the idea of repeated modular squaring as a secure, un-parallelizable, computational workload code. In our work we don’t have to know the modular squaring performance exactly. It can be quite inexact since we use it to rank nodes in comparison to each other. We can assume an average performance based on regular published performance tests of modern computer systems. We just have to be sure, that the communication delays are small enough compared to the modular squaring time.

Uncheatable benchmarks

Cai and Nerurkar propose in "Making Benchmark Uncheatable"[6] several ‘uncheatable’ benchmarks. They propose to use Matrix Multiplication, Fast Fourier Transformation and
Gaussian Elimination as benchmark techniques which have to be representative, optimal, repeatable and uncheatable. Gaussian elimination as a benchmark technique, is also used in the widely known Linpack[20] benchmark. We show in Chapter 4 how to cheat them anyway. Mainly, they did not consider the property un-parallelizable which is important for uncheatable benchmarks as we will show.

After we worked out the idea of repeated squaring as a benchmark, we recognized that Cai, Lipton, Sedgewick and Yao mentioned in "Towards uncheatable benchmarks"[7] a benchmark also based on modular squaring. But they just roughly sketch the idea on one page. We derive and analyze a more specific proposal in detail and also state an improved verification method.
Chapter 3

Motivation

In this chapter, a motivation to this problem is given. A Java simulation shows the impact of claiming wrong computational power. We also do some fundamental considerations about the problems of measuring computational power and why parallelization is an important aspect. After a short discussion about different measure methods, we also state fundamental cheating approaches.

Contents

3.1 Cheating simulation ................................. 7
   3.1.1 Setup ........................................... 8
   3.1.2 Simulation: Grid computing .................. 10

3.2 Fundamental considerations ......................... 13
   3.2.1 Measure computational power .................. 13
   3.2.2 Parallelization problem ........................ 14
   3.2.3 Pretesting or dynamic .......................... 14
   3.2.4 Cheating approaches ........................... 15

3.1 Cheating simulation

The following simulation illustrates the impact of malicious nodes claiming false computational power on grid computing. Grid computing is a typical model for organizations like universities, where the grid computer resources are available to the member of these organizations if they need it. Examples are the Worldwide Large Hadron Collider Computing Grid (WLCG) or weather forecast calculations. Grid computing requires the use of software that can parallelize the computation and deploy these pieces on the grid system.
3.1.1 Setup

A self-written Java program simulates two scenarios where malicious nodes that claim wrong computational power can tamper the computation highly. This Java program can be parameterized through three main values:

- Number of nodes
- Percent of malicious nodes
- Upper and lower limit of computational power in GFLOPS every node can have

Every node is represented by a Java object with following properties:

- Computational power the node has randomly chosen, between the upper and lower limit parameter
- Computational power the node will claim, depending on the scenario
- Flag if the node is malicious

The simulation runs in three main steps:

- Choose the three main parameters
- Choose the scenario and the behavior of the malicious nodes
- The simulation will run the scenario two times:
  - using the real computational power of the nodes
  - using the computational power the nodes claim (if they are malicious)
- The output is the ratio between this two values in percent; So 100% represents the output value if there were no malicious nodes

**Computational power limit parameters**

Powerful computational resources often are used for scientific purposes and therefore the widely used metric is FLOPS (floating point operations per second). We use the Linpack benchmark package for comparison because it is widely used and performance numbers are available for almost all relevant systems including the most powerful scientific supercomputers around the world ([22]).

A Linpack benchmark test from May 2008 (see figure 3.1) is the reference for the upper and lower bound of FLOPS.

- Upper bound: 30 GFLOPS
- Lower bound: 9 GFLOPS
Figure 3.1: Nowadays processor performance in GFLOPS
3.1.2 Simulation: Grid computing

Scenario

In Grid computing the computation is distributed to loosely coupled nodes, the grid. The assumptions for the simulations are:

- There are one hundred nodes available which can be selected for the grid or not
- The selection is based on the computational power every node is claiming
- Depending on the cheating strategy of the malicious nodes, the nodes will behave differently

In this grid simulation there are five node cheating strategies implemented which everyone relates to real-world problems.

- Try to get selection: If the nodes get paid based on the usage of their computational power, the nodes try to get selected. If they are selected, they will provide their full computational power.
  - Claim higher: Node is claiming a random computational power between its real available power and the upper limit.
  - Claim maximum: Node is claiming the upper limit of computational power possible.

- Try to avoid selection: If the nodes are volunteers or are paid in a flat-rate model, the nodes probably are not interested in providing computational resources. However if they are selected, they will provide their full computational power.
  - Claim lower: Node is claiming a randomly lower computational power, between the lower limit and the actually available.
  - Claim minimum: Node is claiming the lower limit of computational power possible.

- Try to get selection and then strike: If nodes want to sabotage the computation they try to get selection and then do not provide any computational resources.
  - Strike: Node is claiming upper bound of computational power but will not provide any computational power to the grid.

Next to this there are also three node selection strategies defining how the nodes are selected for the grid.

- Select all: Select all 100 nodes for the grid computation. You want to compute as fast as possible.
Figure 3.2: Malicious nodes try to get selected

- Select nodes, until a particular limit of available computational power is reached. We defined 1/3 of the average available computational power over all 100 nodes as this limit. 1/3 is chosen to clearly border from the select all case and to get clear results.
  - *Select fastest*: Select the fastest nodes depending what they claim until the limit is reached
  - *Select random*: Select random nodes until the limit is reached

Results

Figure 3.2 shows the effect if malicious nodes try to get selected. Even if the malicious nodes do not claim the maximum possible computational power but just claim it randomly higher, the available computational power decreases up to 20%.

Figure 3.3 shows the effect if malicious nodes try to avoid their selection. 100 % at the y-coordinate represents the claimed computational power of the selected nodes. Remember that the malicious nodes will provide all their computational power once they are selected. For the malicious it is possible to suppress up to 100 % of computational power. And if fastest nodes are selected, malicious nodes can almost completely avoid selection until a malicious node percentage of 50 %!

Figure 3.4 shows the effect if malicious nodes try to get selected and then completely deny
Figure 3.3: Malicious nodes try to avoid their selection

Figure 3.4: Malicious nodes try to get selected and then strike
to provide computational power. Unlike in the other cheating strategy the malicious nodes will provide no computational power once they are selected. This is an unrealistic situation because we assume that after the malicious nodes strikes no additional nodes are selected instead of them. But it illustrates clearly what malicious nodes can cause. If fastest nodes are selected, already a malicious rate of 20% can bring the available computational power to under 10%.

With these simulations it is clearly shown that malicious nodes can highly influence the performance of distributed computation, if they can arbitrarily claim computational power. There are many examples and because distributed computing gets more and more popular, the problem of claiming wrong computational power will also increase.

3.2 Fundamental considerations

3.2.1 Measure computational power

To measure computational power (cp) of a computer system is a very tricky task. At first we have to define what computational power is. It is obvious that the clock rate in GHz is not sufficient enough to define the overall cp because even the processor architecture can make a huge difference. To know the theoretical specifications of the processor like instructions per second, may not reveal how fast a specific code will run. Some other related quantities are the algorithm to solve a problem, the human and processor effort to optimize the code, the operating system and the architecture of the computer. Some advanced architecture may faster load and store the operands of floating point operations.

To measure the computational power of a computer system for a specific program, abstract the program to representative fundamental operations that can run at their own, and then measure the run time of it in comparison to other computer systems. Then the really needed resources for this computation are benchmarked. In an abstract way, computational power can be defined as: How many times specific code can run per second.

This definition can be adapted to benchmarks.

- Real program benchmarks: Measure the time to e.g. start Office or to render a dummy picture. Here the specific code is complex software where different kinds of fundamental CPU operations are needed

- Kernel benchmarks: Measure how many fundamental operations per seconds are possible. This results are often in FLOPS (floating points operations per second) or for non-scientific applications in MIPS (million instructions per second)

Powerful computational resources are often used for scientific purpose and therefore the widely used metric is FLOPS. Top500.org[22] uses the Linpack[20] package for comparison
because it is widely used and performance numbers are available for almost all relevant systems. The comparison of the computer systems in Top500.org is not done by a trusted party. All manufactures of such a powerful computer systems tests their systems at their own because every computer system has their specific advantages. The often used HPL benchmark depends also on the algorithm, chosen by the manufacture and the amount of memory available on the computer being benchmarked. The comparison depends on the fact, that every manufacture can optimize the algorithm and the size of the problem used for solving the equations for their computer system.

Benchmarks can not measure computational power in a standalone unit but can rank different computer systems by the performance of running the benchmark code.

### 3.2.2 Parallelization problem

For our purpose a code is parallelizable if several connected computational units, can run the code faster than any of these computational units alone. The idea of un-parallelizable benchmark code is that no computer system can claim higher computational power with help of other computer systems (distributed computing).

But this limits the use of multiple core processors and multiple processor systems. At the first moment a weaker limitation seams to be better: It should not be possible to parallelize the code over the LAN/WAN. The computer system can then parallelize just internal what is allowed. The detection of the latency of a network connection is possible by choosing the benchmark properties in a way that the run time of the benchmark code is several times lower then the expected network delay. But this implies the too strong assumption that you can detect the exact network delay from the verifier to the benchmarked computer system in the moment of sending the benchmark code and receiving the answer. To measure the run time remotely on the benchmarked system, a trusted party on this system is needed. The implementation of such a trusted party like TPM[23] has also too strong assumptions. These limitations can limit the hardware which can be used or may cause software update problems[1].

Our approach is to make sure, that our benchmark code can not be efficiently parallelized. This excludes also the parallel execution on several computational units within a cluster or a supercomputer. If a computer system has several processors (or cores), it has to run our benchmark in parallel on each processor.

### 3.2.3 Pretesting or dynamic

Benchmarks measure the available computational power which is just accurate at the moment of testing. The computational power available can change very fast. It depends on the goals
what kind of testing is accurate.

Pretesting

The node is once benchmarked before he is selected or his services are used. This is the case by common benchmarks like Linpack[20].

Advantages:

- More detailed tests are possible because it can take more time and power to benchmark
- Simpler to implement because it has to be real time and resources spared
- Good for maximum computational power tests

Disadvantages:

- Available computational power can change very fast due workload changes etc.

Dynamic

The node is continuously benchmarked also during the time he provides his services.

Advantages:

- Real time information about available computational power
- Claimed computational power can be verified over full service period

Disadvantages:

- Efficient and real time test functions needed which do not have a big overhead
- Efficient data collection needed
- Probably measures available cp and not provided cp for the client because the provides services is running

Dynamic testing implies to strong assumptions. We focus on the pretesting approach.

3.2.4 Cheating approaches

Alice benchmarks Bob with a problem \( f(x) \) he has to solve and report the result. The variable \( x \) must not change the run time complexity but just the result as the results have to be comparable. Let \( A (DTIME(n^a)) \) and \( B (DTIME(n^b)) \) be two complexity classes which represent the computational power of Alice and Bob respectively. It is necessary that checking the result through Alice is faster than computing the result through Bob.
Benchmark cheating approaches all try to let Alice believe of a faster computation of the benchmark through Bob as he actually is able to.

1. Solve the problem with a faster algorithm. Use $g(x') = f(x)$ where $g(x')$ can be computed in $DTIME(n^c), c < b$.

2. Do not solve problem but just compute a result which will pass checking. This is an attack on the verification strategy of Alice.

3. Parallelize execution with $m$ allies. Use $g(g_1(x_1), ..., g_m(x_m)) = f(x)$ where $g(x)$ and with that every $g_i(x_i)$ can be computed in $DTIME(n^c), c < b$.

The first and second approach seems to be similar in their definition. A difference is that the second approach is just possible if Alice uses an optimized checking method. For example if Alice does not check all digits of the results but just the cross sum. The first cheating approach is harder to detect since it is actually not cheating but just optimization.
Chapter 4

Analysis of existing benchmarks

The insecurity of the already proposed benchmarks and how they can be cheated is explained in this chapter. We analyze every proposed uncheatable benchmark in detail. The underlying techniques are explained and a short summary of the proposed benchmark is given. In the weakness analysis we point out the cheating potential and propose a way to attack the benchmark. For the Gaussian Elimination and the Matrix Multiplication benchmark proposal, we implemented the benchmark and the according attack in Java. The results are presented by showing the advantage malicious nodes can gain. Next to the measured results from our lab environment, we interpolate the results for theoretical fiber optic and internal cluster communication.

Contents

4.1 Common Benchmarks ........................................... 18
  4.1.1 Linpack[20] ................................................. 18
  4.1.2 Whetstone[4] .............................................. 18
  4.1.3 Generally ................................................... 19

4.2 Proposed uncheatable benchmarks ................................. 19
  4.2.1 Gaussian elimination benchmark .......................... 20
  4.2.2 Fast Fourier transformation benchmark ................. 26
  4.2.3 Matrix powering and multiplication benchmark ........ 29
  4.2.4 Sorting benchmark ....................................... 32
  4.2.5 The Power Benchmark ..................................... 34
  4.2.6 The One-way Benchmark .................................. 35

4.3 Implemented attacks ........................................... 36
  4.3.1 Introduction ............................................... 36
  4.3.2 Gaussian elimination parallelization attack ............ 38
4.1 Common Benchmarks

Many common benchmarks measure the computational power of computer systems in FLOPS. MFLOPS is a rate of execution, millions of floating point operations per second. When this term is used it will refer mostly to 64 bit floating point operations and the operations will be either addition or multiplication. The difference between benchmark packages are the code they run to compute the FLOPS metric. The theoretical possible FLOPS deduced out of the processor specification are not reliable because this theoretical value will not be reached in 'real life'. FLOPS is not an exact value like CPU clock rate, because the value differs depending on the benchmark code that is executed. If your Benchmark code is just one addition, the FLOPS output will be completely different to a benchmark who tries to run a more representative benchmark code (synthetic benchmarks).

4.1.1 Linpack[20]

The Linpack benchmark measures a computers floating point rate of execution (64bit). It is determined by running a computer program that solves a random system of linear equations \( Ax = b \). The used code to solve the equations is not the fastest possible to compute the solution. There are three levels of problem size and optimization opportunity: 100 by 100 problem (inner loop optimization), 1000 by 1000 problem (three loop optimization - the whole program), and a scalable parallel problem (HPL) used also for Top500.org. Solving a system of equations with Linpack requires \( \frac{2}{3}n^3 + 2n^2 + O(n) \) floating point additions and multiplications. They propose that if you need exact values you can use PAPI[21] to be more accurate in the measured floating point operations. PAPI allows to access low level floating point operation counters.

4.1.2 Whetstone[4]

Whetstone was state of the art until \( \sim 1990 \). This Benchmark is intended to characterize computers for use in scientific processing and hence has a very high number of floating point operations. The actual "C/C++ Whetstone Benchmark Double Precision" benchmark has the following eight testing sections.

- Section 1, Array elements
- Section 2, Array as parameter
• Section 3, Conditional jumps
• Section 4, Integer arithmetic
• Section 5, Trig functions
• Section 6, Procedure calls
• Section 7, Array references
• Section 8, Standard functions

The counted value is million whetstone instructions per second (MWIPS). This will be converted in FLOPS with static factors. It was also designed to defeat automatic compiler optimization what can have a significant influence on the benchmark results. The mixture of different code function is not typical for modern computer system anymore. The Whetstone code is often parallelizable and human optimizations can be done as well.

4.1.3 Generally

Common benchmark suites are for trusted use only. Either the owner of the computer system wants to find out the performance for him self, or the tester has full access to the tested systems. There is no verification whether the benchmark code was not modified to speedup the computation. The main goal of these benchmarks is not to compute a distinct result, but just to produce a computational dummy load to measure the time over the known FLOP effort. The calculation of the result value, if there is one, can be done faster. Some benchmark variants like HPL are made for parallel execution what would not be useful for our purpose.

4.2 Proposed uncheatable benchmarks

In this section we analyze benchmarks that are designed not to have the discussed disadvantages, i.e. they should not allow to forge wrong computational power. The uncheatable benchmarks proposed by Jin-Yi Cai and Ajay Nerurkar[6] were built along the following properties. A benchmark had to be representative, optimal, repeatable and uncheatable.

The first three properties were already considered in earlier common benchmark suites like Whetstone[4] and Linpack[20]. The property ‘optimal’ covers the problem to avoid non-malicious automated optimizations through e.g. the compiler. Uncheatable benchmarks however had also to be resistant against automated and manual manipulations through malicious parties.

We will analyze the security of the proposed benchmark in detail. However they did not consider the property un-parallelizable which is important for uncheatable benchmarks as we showed. All proposed benchmarks in their paper are enough parallelizable to influence
the result significantly. Most benchmarks rely on a problem, a benchmarked system had to
solve. On the one side, the problem has to be large enough to produce measurable work load.
But the more complex a problem is, the more difficult it is to be sure that there is no faster
algorithm or even a shortcut to compute the problem. Even if there exists an asymptotically
lower bound, a speed up with e.g. factor two compromises a benchmark heavily.
Unfortunately Cai and Nerurkar described their benchmarks just roughly, so they do not
state what specific algorithm they want to use for the calculation. So they assume that the
benchmarkd node just use a kind of Gaussian elimination algorithm or a kind of FFT. But
the exact algorithm is crucial in the security analysis.
This analysis will therefore just analyze what is actually described, i.e. mostly the benchmark
preparation phase and verification idea.

4.2.1 Gaussian elimination benchmark

Proposed in ”Making Benchmarks Uncheatable”[6].

Basics

The asymptotic Runtime is generally $O(n^3)$ but optimized methods reach $O(n^{2.36})$.
The Gaussian elimination method is a simple method to solve linear systems $Ax = b$,

\[
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
    a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
    \vdots \\
    a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m
\end{align*}
\]

where

\[
A = \begin{pmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m1} & a_{m2} & \cdots & a_{mn}
\end{pmatrix}
\quad \text{and} \quad
x = \begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}
\quad \text{and} \quad
b = \begin{pmatrix}
    b_1 \\
    b_2 \\
    \vdots \\
    b_m
\end{pmatrix}.
\]

Three types of elementary row operations are needed: Multiplying a row by a non-zero con-
tant, adding a multiple of one row to another and exchanging two rows.

There are two parts in this calculation: The forward elimination and the backwards sub-
stitution. In the elimination process, the matrix entries below the main diagonal will be
eliminated with help of the row operations.
After that you will be able to back substitute to find the solution. There exist also other methods or improvements to solve linear systems like Gauss-Jordan, Gaussian Elimination with partial pivoting or the iterative Gauss-Siedel method. Gaussian Elimination with partial pivoting applies row switching to normal Gaussian Elimination. At the beginning of the k-th step of forward elimination, find the maximum of $a_{k,k}...a_{n,k}$. If $a_{p,k}$ is the maximum, switch row with them.

The benchmark algorithm

They loosely propose a way to generate and verify the system $Ax = b$. Mainly it is a multiplicative congruential generator with random seed $s$ and a multiplier $m$. A multiplicative congruential generator looks like this:

$$x_1 = s; \ x_{k+1} = m \times x_k \ (mod \ B)$$

Matrix $A$ will be generated entry by entry and $b$ is set to be the sum of the columns of $A$, to be treated as a column vector. To get a dense matrix with double precision floating point number, they choose the modulus as a power of two $B = 2^{48}$. If $B$ is chosen in that way, they can generate $x_k$ and shift this value by 48 bit with $x_k/B$. The verification with a given $x$ within $O(n^2)$ is faster then solving for $x$.

Weaknesses

This problem is parallelizable with the parallel successive Gaussian Elimination (SGE) algorithm[8] and the parallel GE algorithm[9]. We propose also the second parallel attack because the implementation results of the first one were bad due the high communication delays in our lab.

The benchmarked system will be asked to compute the solution $x$ of $Ax = b$. The method to solve the system can be freely chosen as there is no verification of the used algorithm or intermediate steps. There are many ways to solve a linear system what makes it difficult to get reliable result on the available computational power.

The verification is to check if $Ax = b$ holds and can not be attacked as Alice is just convinced if Bob can deliver a matching solutions $x$. The CP measurement part is not described in detail.

The remaining part which we can attack is the generation of the linear system through Alice. The proposed modulo is dividable by two. Therefore the generator is less random. We show
that this generator generates sequences that are not as random as they should be. With that the probability to have the linear system precomputed, rises.

Think of a generator with the parameters $a = 13$, $c = 1$, $m = 16$ and $x_0 = 0$ which should produce random number in $0 \cdots 15$. It produces the sequence

\[
\begin{align*}
0000, 0001, 1110, 0111, 1100, 1101, 1010, 0011, 1000, \ldots
\end{align*}
\]

You can see that the least significant bits (LSB) changes every time, since $m = 16$ is a multiple of 2. Further the least significant two bits are cyclic with period four (multiple of 4) and the least significant three bits are cycle with period eight (multiple of 8). The existence of such patterns makes the sequence less random.

Even when we do not factorize the modulus you can see the factor two and therefore double the probability to guess following numbers.

**Parallel attack 1**

We use the existing parallel SGE algorithm [8] to build our parallel attack protocol. With $n$ system we can theoretical reach a run time of $2n^2 + O(n)$ and therefore an advantage of factor $n$. The advantage of a real implementation depends on the size of the equation matrix, the computational power of the nodes and the network latency. The worst case in practical implementation would be if we just can use one additional node for parallelization. But even then we have a speedup of approximately factor two compared to an $O(n^3)$ algorithm.

The solution of $Ax = b$ can be computed using the classic Gaussian Elimination algorithm. This method has a triangulation phase followed by a back-substitution phase. In our protocol we will parallelize an adapted triangulation phase in a way, that at last the back-substitution is one division. We know that the elimination of $\frac{n}{2}$ variables results in $\frac{n}{2}$ linear independent equations of these variables.

The parallel algorithm to solve the linear equations has three steps which will be used iteratively. We assume that $w$ computer systems (nodes) are available for the parallelization where ideally any node knows the computational power of the others. The parallelization works as follows:

1. Send matrix $A_i$ of order $n$ to the two fastest idle nodes $P_u$ and $P_v$.

2. $P_u$ and $P_v$ uses GE method to triangulate $A_i$ in forward (left to right), respectively backward (right to left) direction. $P_u$ eliminates the variables $x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}$ and $P_v$ the variables $x_{n-\lfloor \frac{n}{2} \rfloor}, \ldots, x_{\lceil \frac{n}{2} \rceil}$. The matrix is now reduced to the half of their size in parallel execution.

3. The matrix $A_{i0}$ and $A_{i1}$ can now iteratively be sent to step one again. If the parallelization level (p-level) is reached (number of remaining variables smaller than predefined), the next available fast node can compute the sub solution with the classic GE method.
The communication delays between the nodes can have a significant influence on the computational time needed, if the planned benchmark time is not much larger than the communication delays in the network. Therefore, synchronization of the list with the fastest available nodes can be impracticable. We propose two different approaches to avoid additional communication.

- If just the ranking of the nodes by their computational power is known, this list is fix partitioned for computing. First we need two nodes to compute the forward and backward GE. Then we need four nodes, what means two additional for computing and then four additional etc. Out of the matrix numbering, every node can compute to which node he have to send his result matrix for backward elimination (he will do the forward elimination by himself).

- If the exact computational power of every node would be known, Bob#0 can precompute a customized computing plan for optimal performance. We will focus on the first approach, since knowing the exact available computational power is a quite strong assumption as we showed before.

Now we state the detailed protocol among the $w$ nodes.

Remark: If the benchmarked node Bob#0 has enough computational resources, he can additional act as a right hand Bob#i.
Parallel attack 2

We use the existing parallel GE algorithm[9] to build our parallel attack protocol. With $n$ system we can theoretical reach a run time of $O(n^2)$ and therefore an advantage of factor $n$ compared to the original GE algorithm. The advantage of a real implementation depends on the size of the equation matrix, the computational power of the nodes and the network latency. The worst case in practical implementation would be if we just can use one additional node for parallelization. But even then we have a speedup of approximately factor two compared to an $O(n^3)$ algorithm.

The solution of $Ax = b$ can be computed using the classic Gaussian Elimination algorithm. This method has a triangulation phase followed by a back-substitution phase. Here we parallelize the time consuming triangulation phase. We assume $w$ computer systems (nodes) available for the parallelization where ideally any node knows the computational power of the others. The classic GE triangulation phase implementation for $Ax = b$ with $n$ entries in $A$ can be abstract like this:

1. For all columns in $A$ except the last one do:

2. Search for the maximal element in the actual $i$-th column and swap lines if needed (partial pivoting)

3. Subtract a multiplier of the actual $i$-th row to any lower ($i, \cdots, n-1$) row, to eliminate the $i$-th variable in this rows.

We parallelize the third step. Instead one node alone subtracts the actual row from all others, we can split the remaining rows into $w$ pieces and send them together with the actual row to each node. This includes a part of the $A$ matrix and the related part of $b$. Every node receives also the actual row it have to use for the subtraction. Compared to step 3, step 2 is not very time consuming and we can see that we can run the triangulation phase $w$ times.
faster in theory. The drawback is the large number of connections to send and receive the pieces to and from the nodes ($\approx 2n$).

1. Alice: Send to Bob $A, b$.

2. (Actual row $i=1$)

3. Bob: Search the maximal element in $A[i, \cdots, n][i]$ and swap lines if needed.

4. Bob: Split matrix $A[i+1, i+1 \ldots n, n]$ and $b[i+1 \ldots n]$ in $w$ pieces and send them with row $A_p[i, i \ldots n], b_p[i]$ to the nodes.

5. Nodes: Eliminate first row of $A$ with GE operations. Send the resulting matrix $A, b$ back.

6. $i = i + 1$ restart with step 3 if $i < n$.

The communication model between the nodes:
4.2.2 Fast Fourier transformation benchmark

Proposed in ”Making Benchmarks Uncheatable”[6].

Basics

The naive $n$-point Fourier transform involves calculating the scalar product of the $n$-sized sample with $n$ separate basis vectors. Each scalar product involves $n$ multiplications and $n$ additions and therefore the naive approach is an $O(n^2)$ algorithm. However, with the
optimized fast Fourier transformation algorithm which is a clever re-arranging of these operations, one can optimize it to a $O(n \log n)$ algorithm. The FFT transforms a signal from the time domain to the frequency domain and vice versa. Generally the transformation looks like this:

$$F_i = \sum_{k=0}^{n-1} \omega^{jk} f_k \text{ where } \omega = e^{-2\pi i/n}$$

$$f_i = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{jk} F_k \text{ where } \omega = e^{2\pi i/n}$$

The code for FFT is widely known. A simple benchmark can work as follows. The verifier (Alice) generates two sequences of integers. The benchmarked node (Bob) then transforms both sequences into the frequency domain with the widely known Cooley and Tukey algorithm. Bob can now multiply the two sequences component wise and then apply the inverse FFT to the result. Alice checks then for correctness of the computation. This example is easy to cheat, so let us check the proposed uncheatable FFT benchmark.

**The benchmark algorithm**

1. Alice sends Bob $n = 2^l$ values $x_i$
2. Bob performs FFT on all values and sends the results $y_i$ back
3. Alice generates a secret number $r$
4. Alice can check Bob’s computation by checking in $O(n)$ if

$$\sum_{j=0}^{n-1} y_j r^j = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \omega^{jk} x_k r^j = \sum_{j=0}^{n-1} x_k \sum_{k=0}^{n-1} (\omega^k r)^j$$

holds

Why is point 4 the case?

If the secret number $r$ is a root of unity of order higher than $n$, for all $i < n$ we know that $r^i \neq 1$. The closed form of the geometric sum is $\sum_{k=0}^{n-1} a r^k = a \frac{1 - r^n}{1 - r}$ ($r \neq 1$) and therefore $\sum_{k=0}^{n-1} (\omega^k r)^j = (r^n - 1)/(\omega^k r - 1)$ since $\omega^k r \neq 1$. Both sums $(r^n - 1) \sum_{k=0}^{n-1} x_k/(\omega^k r - 1)$ and $\sum_{j=0}^{n-1} y_j r^j$ can be computed and compared in $O(n)$ to verify the calculation of Bob.

**Weaknesses**

This algorithm is parallelizable with the Cooley-Tukey algorithm which is a divide-and-conquer approach. Let $w$ be the number of available computer systems for parallelization, $O(\frac{n \log n}{w})$ is reached[10]. The Cooley-Tukey algorithm recursively partitions a DFT into two half length DFTs and then applies Fourier transformation to each of them. Soundness is therefore not guaranteed for the computation part.
Again the concrete method to solve the problem (FFT) is not specified. However this is crucial because for some $n$ the faster radix-4 algorithm can be applied if enough memory is available. Due to the loose description it is just possible to do the security analysis of the preparation and verification idea. This idea relies on the fact that Bob can not compute the sum in the stated fast way, because he does not know the secret number $r$. An $n$-th root of unity can be computed with $e^{(2\pi i)/n}$ where $i$ is the imaginary unit ($\sqrt{-1}$). Therefore the domain of root of unity with order bigger than $n$, is too big to allow prediction of $r$.

**Parallel attack**

We take an existing parallel algorithm[10]. The theoretical parallelization advantage of $n$ nodes is $n$. The advantage of a real implementation depends on the size of the matrix, the computational power of the nodes and the network latency. The worst case in practical implementation would be if we just can use one additional node for parallelization. But even then we have a speedup of approximately factor two while we can also reduce the network latency by configuring the network in a way that Alice sends the initial matrix directly to Bob and the other node (say Bob#2).

Let $w$ be the number of computer systems available for parallelization and $n$ the number of points to transform ($n \gg w$). There are three steps:

1. **Preparation:** We divide the $n$ points into $p$ consecutive sub sequences of size $n/p$. The $i$-th sub sequence has the following indices: $i\frac{n}{p}, \cdots, (i+1)\frac{n}{p} - 1 (i = 0, \cdots, p-1)$. The FFT algorithm consists of $\log n$ steps, these steps do butterfly computation with index distance $1, \cdots, 2^{\log n-1}$ respectively. Parallel FFT is divided into two phases.

2. **Computation without communication:** From step 0 to step $\log(n/p) - 1$ there is no communication between any pair of computer systems.

3. **Computation with communication:** From step $\log(n/p)$ to step $\log n - 1$, because the index distance is larger then $n/p$, each system has to communicate with another system.

Detailed information can be found in [10].
Bob#0 can also be one of the working allies or just do the synchronization.

4.2.3 Matrix powering and multiplication benchmark

Proposed in "Making Benchmarks Uncheatable" [6].

The benchmark algorithm

The idea is to compute for a matrix $M$, $k$ rounds for squaring ($M^{2^k}$). The run time for squaring two $n$-sized matrices is $O(n^{2.376})$ with Coppersmith’s Winograd algorithm. For $k$ rounds of squaring it would be $O(k \times n^{2.376})$.

The only question remaining is how the verifier (Alice) can check the computation of the benchmarked system (Bob). This benchmark uses the following decomposition of a matrix $M$.

$$M = T^{-1}JT.$$  

where $J$ is a Jordan matrix

$$J = \begin{bmatrix} J_1 & \cdots \\ \cdots \\ J_p \end{bmatrix} \quad ; \quad J_i = \begin{bmatrix} \lambda_i & 1 & \cdots \\ \lambda_i & \cdots & 1 \\ \cdots & \cdots & \cdots \\ 1 & \cdots & \lambda_i \end{bmatrix} (\lambda_i: \text{eigenvalue of } M)$$

and $T$ is some non-singular matrix.

Note that, if Alice generates randomly $J$ and $T$ and out of it $M$, Bob will have to do the
squaring with $M$ but Alice just with $J$ ($M^i = T^{-1}J^iT$). This because finding the eigenvalue and the Jordan normal form decomposition is computationally more costly than matrix multiplication. To minimize the effort of Alice to compute the inverse of $T$, she chooses a Householder matrix. The advantage of using a Householder matrix is that the matrix is extremely simple to store and manipulate. It is its own inverse.

**Weaknesses**

One round of squaring (one multiplication) can be computed in $O\left(\frac{n^\alpha}{w}\right)$ with a parallel algorithm\cite{11} where $\alpha$ is the run time complexity exponent of the adapted sequential algorithm and $w$ the number of available computer systems for parallelization. But steps of more than one multiplication (multiple squaring) are not independent and therefore not parallelizable. Surely the parallelility has his limits due the needed communication time of the involved systems. However, this benchmark idea is similar to the modular squaring benchmark that we all discuss in chapter 5. But the Matrix Multiplication benchmark has the additional disadvantage that the computational gain of a parallel algorithm for one round of squaring is much higher. Therefore the cheating potential is also higher. Additionally the dimension of result values in the matrix can be predicted because they growing constantly and therefore the values are quite restricted in their randomness.

**Parallel attack**

The normal matrix multiplication $C = A \cdot B$ can be parallelized by broadcasting the sub matrices $A$ and $B$ every time to all allied computer systems where they compute a predefined field of $C$. But clearly it would perform better if we parallelize a fast matrix multiplication algorithm. The theoretical parallelization advantage is the number of available system for parallelization $w$. The advantage of a real implementation depends on the size of the matrix, the computational power of the nodes and the network latency. The worst case in practical implementation is if we just can use one additional node for parallelization. But even then we have a speedup of approximately factor two. We will describe the parallel attack for multiple squaring of a $n$-sized matrix using a parallel Strassen’s algorithm\cite{13}\cite{14}. Whether or not, the parallelization of multiple rounds of matrix squaring is efficient, depends on the size of the matrix. The larger the matrix is, the more advantage a parallel attack can give. The smaller the matrix is, the less advantage you have. The extreme case is to choose a matrix of size 1 and that is actually our repeated modular squaring approach. We use the Winograd variation of Strassen’s algorithm. It uses 7 matrix multiplications and 15 matrix additions:

$$C = AB \text{ where } A, B, C \text{ are } n\text{-sized matrices.}$$
Let matrices $A$, $B$ and $C$ be divided into half-sized blocks as follows (fill up with zeros if necessary).

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix} \cdot \begin{bmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{bmatrix} = \begin{bmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{bmatrix}.
\]

Now we can compute

\[
C_{1,1} = P_3 + P_2, \\
C_{1,2} = T_1 + T_3, \\
C_{2,1} = T_2 - P_7, \\
C_{2,2} = T_2 + P_5,
\]

with

\[
\begin{align*}
S_1 &= A_{2,1} + A_{2,2} \\
S_2 &= S_1 - A_{1,1} \\
S_3 &= A_{1,1} - A_{2,1} \\
S_4 &= A_{1,2} - S_2 \\
S_5 &= B_{2,1} - B_{1,1} \\
S_6 &= B_{2,2} - S_5 \\
S_7 &= B_{2,2} - B_{1,2} \\
S_8 &= S_6 - B_{2,1}
\end{align*}
\]

\[
\begin{align*}
P_1 &= S_2 S_6 \\
P_2 &= A_{1,1} B_{1,1} \\
P_3 &= A_{1,2} B_{2,1} \\
P_4 &= S_3 S_7 \\
P_5 &= S_1 S_5 \\
P_6 &= S_4 B_{2,2} \\
P_7 &= A_{2,2} S_8
\end{align*}
\]

The parallel algorithm for matrix multiplication then works as follows:

Consider a mesh of $w = p^2$ computer systems. Matrices $A$, $B$ and $C$ are also partitioned into $p$-sized blocks that each block is an $(\frac{n}{p}) \times (\frac{n}{p})$ matrix. Suppose system $P_{i,j}$ stores $A_{i,j}$ and $B_{i,j}$ and computes $C_{i,j} = \sum_{k=0}^{p} A_{i,k} B_{k,j}$. To compute the product $C = AB$ for $k = 0$ to $k = p$, the following steps are repeated.

1. Each system $P_{i,k}$ ($i = 1, \ldots, p$) broadcasts $A_{i,k}$ in a horizontal direction.
2. Each system $P_{k,j}$ ($j = 1, \ldots, p$) broadcasts $B_{k,j}$ in a vertical direction.
3. Update $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$ for all systems.

At each step, each system uses Strassen’s algorithm to compute $A_{i,k} B_{k,j}$. Each system then computes the $(\frac{n}{p}) \times (\frac{n}{p})$ matrix multiplication $p$ times. This approach can even be improved by implementing a kind of 'fastest node first' techniques where there are more pieces than allies. The faster systems get more pieces to compute than slower ones. So you reduce the negative influence of slow computer systems.

**Protocol for parallel multiple matrix squaring:**

The above stated algorithm can be slightly changed to fit for parallelization of multiple matrix squaring. At first Alice sends the matrix $A = B$ to Bob. Bob just relays this matrix to his allies. They can then directly compute their $C_{i,j}$ from the first squaring. This $C_{i,j}$
then is directly their new $A_{i,j} = B_{i,j}$ and we have the starting parameters to use the above parallel matrix multiplication algorithm for $k$-time squaring.

```
A,k

Bob#0  Bob#i (1 ≤ i ≥ w)

broadcast

A,k

compute $C_{0,i,j} = \sum_{k=0}^{\theta} A_{i,k}B_{k,j}$

broadcast horizontal (1)

broadcast vertical (2)

where $A_{i,j} = B_{i,j} = C_{i,j}$

update $C_{1,i,j}$ (3)

... (repeat w − 1 times)

C

C_{1,1}...C_{w-1,w-1}
```

### 4.2.4 Sorting benchmark

Proposed in "Making Benchmarks Uncheatable"[6].

**The benchmark algorithm**

As widely known, sorting algorithms have an asymptotic run time of $O(n \log(n))$. Since the result of sorting is not a single value but a sorted set of numbers, this kind of benchmarks also tests the handling of a large data set. Alice lets Bob generate the number he has to sort at his own to avoid workload for herself.

- Alice provides Bob with a seed for a pseudo random generator to generate $N$ keys.
- Alice will 'plant' in advance a pair of secret values $x$ and $y$ among the generated keys, such that $x$ and $y$ are most likely to be next each other in the final sorted order of the list.
- Bob will then sort the $N$ numbers and when he is finished, Alice will ask Bob which one of two values is in the sorted list and ask also for the successor of this value.
- If Bob has sorted the numbers, he can answer in $O(\log n)$.

The pseudo random generator is based on the RSA scheme with $L = pq$ as the modulus. The private key $d$ and the public key $e$ are generated as usual with the extended Euclid such that $d \cdot e = 1 \mod \phi(L)$ holds. Now Alice picks a random $x$ and $\delta \approx L/N$ ($L \gg N$). Let
\( u = D(x) = x^d \mod L, \) and \( v = D(y = x + \delta) = y^d \mod L. \)

The encryption function which Bob has to use to generate \( N \) number is \( E(z_i)|i = 0, 1, \ldots, N - 1. \) So Alice has to define the sequence \((z_i)\) to be sure that it will generate \( u,v \) and therefore \( x = E(u), y = E(v) \) within \( N \) steps. Alice therefore picks \( r, t > 0 \) and \( r + t < N \) randomly and let the step size \( \delta = r^{-1} \mod n \) (difference between \( u \) and \( v \) is \( r \times \) the step size). Finally let the seed \( s = v - t \times \delta \mod L. \)

The sequence

\[
z_i = s + i \times \delta \mod L, i = 0, \ldots, N - 1
\]

is then a progression that starts at \( s \), with step size \( \delta \), and it will include \( u = z_{t+r} \) and \( v = z_t. \)

It is also clear that \( x = E(u) \) and \( y = E(v) \) are among the keys generated and with high probability \( y \) will be the successor of \( x \) because of the chosen \( \delta \approx L/N. \)

It is easy to generate a random \( x' \) that is not in the list. It is not with probability of \( 1 - N/L \) \( (L \gg N) \) and you can check it by use the decryption function \( D(x') \) and attempt to solve for \( i \) in \( s + i \times \delta = D(x') \mod L \) where must be within \( 0, \cdots, N - 1. \)

### Weaknesses

Parallelizable through divide and conquer approach.

\( O(\log n) \) answer time and \( O(n \log n) \) sorting time is an asymptotic value. Single runs can not be compared because the effective amount of operations which has to be done is not stable.

Just the average over several runs may give you a comparable result.

### Parallel attack

The concrete algorithm to sort the values is not verified or even specified in their proposed algorithm. So our parallel attack we will also just assume a fast algorithm is available for sorting.

Before benchmarking Bob ask all his friends for help and numbers them. So he has \( w \) allied computer systems available which are numbered from one to \( w. \) When Bob receives \( e, L \) and \( N \) he broadcasts this values and \( w \) to all his friends. They divide \( N \) into \( \lceil m/(w+1) \rceil \) sized pieces and sort their numbers each. Bob sorts the first piece of numbers. If all allies are finished, Bob also announce this to Alice and will get two values \( x \) and \( x' \). He broadcasts this two numbers to all allies, searches also for \( x, x' \) and their successor and waits for the replies. Alice will get \( w \) messages back with four parameters each:

- \( v \in 0, 1: \) Is \( x \) in the list?
- \( v' \in 0, 1: \) Is \( x' \) in the list?
- \( y: \) What is the next higher number after \( x? \)
- \( y': \) What is the next higher number after \( x'? \)
Bob can see if \( x \) or \( x' \) was generated and can also look through \( w + 1 \) numbers to get the successor.

This approach can even be improved by implementing a kind of 'fastest node first' techniques where there are more pieces than allies. So if one system finished, he will get another pieces. So you reduce the negative influence of slow computer systems.

If Bob would have just slow connections to his allies, it can be necessary for Bob to first get all sorted sub lists and combines them to one single list. This can be done with the Merge sort approach. But most time this would not be necessary because Alice can not set the answer bounds \( (O(\log n)) \) that small, that two additional transmission delay will break it.

### 4.2.5 The Power Benchmark

Proposed in "Towards uncheatable benchmarks"[7].

In the fundamental operations equal to our proposed benchmark but without any detailed protocol description or security analysis. Additionally we will improve the verification run time for the verifier.
4.2.6 The One-way Benchmark

Proposed in "Towards uncheatable benchmarks"[7].

The benchmark algorithm

Alice selects an oneway function $e()$, for example based on discrete logarithm: $a^{h(k)} \mod n$, for $k = 1..., m$, where $h()$ is some hash function. Alice also selects a random number $k, h(k)$ and tells Bob the value $x = a^{h(k)} \mod n$. Bob has then to find the $k$, so that the equations hold.

Weaknesses

Obviously this benchmark is highly parallelizable. As the computational effort for one round can vary very heavily, just several rounds produces in average an more or less accurate result which is comparable to results of other systems.

Parallel attack

Supposing that the target domain of $h()$ is larger than $m$, and $n$ is chosen carefully, searching the correct $k$ within $m$ possibilities is the most efficient way to find correct $k$. Repeated benchmarking requires that $a$ and $h()$ is chosen differently every time to avoid table lookup attacks.

Before benchmarking Bob ask all his friends for help and numbers them. So he has $w$ allied computer systems available which are numbered from one to $w$. When Bob receives $x, m, a$ and $h()$ he broadcasts this values and $w$ to all his friends. They divide $m$ into $\lceil m/(w+1) \rceil$ sized pieces and search their piece each for the matching value. Bob searches the first piece and waits for the correct answer which he or one of his allies will find. This approach can even be improved by implementing a kind of 'fastest node first' techniques where there are more pieces than allies. So if one system finished searching he will get another piece. So you reduce the negative influence of slow computer systems.
Bob\#0 \(x,m,a,h()\) \(\rightarrow\) broadcast parameters \(x,m,a,h(),w\) \(\rightarrow\) Bob\#i \(1 \leq i \geq w\)

search k within \(m_i \ldots m_{i+1}\)
with \(m_i = \lceil m/(w+1) \rceil \cdot i\)
so that \(x = a^{h(k)} \mod n\)

Other attacks can be possible to a concrete \(h()\) function. But the hash function is not specified concretely but just assumed to be secure.

### 4.3 Implemented attacks

#### 4.3.1 Introduction

We showed that existing 'secure' benchmarks techniques theoretical are all vulnerable to parallelization. Theory and practice is often miles away from each other. So we implemented the two most promising secure benchmarks to not have just theoretical results. The implementation is done with Java and as communication framework Java RMI over a 100Mbps LAN. These are obviously not the fastest parameters for such an implementation but if it work with this setting, it work also in much better environment.

**Test environment**

**Computer systems:** Information about the used computer systems.

**Computational power:** We assume and therefore have used for all Nodes and also for the prover similar computers comparing to their computational power. Especially in the matrix multiplication attack we give every node the same sized sub matrices to multiply. The next squaring round has to wait until all nodes replied their results. Therefore the distributed calculation is as fast as the slowest node. Of course it would be possible to program a load balancing mechanism where we have more pieces than nodes and faster nodes will compute more pieces. Due the lack of time and my coding skills this was not possible.
Thread handling / multicore All computers had single core cpu’s and we had not to care about potential fluctuations in the results because of different thread behavior. Remember that we pointed out in our proposed benchmark protocol, that if we test a multicore machine, we will send also several benchmarks they has to solve concurrent.

Special preparation After several problems with time consuming processes like log-watch and syslog we disabled all services who potentially can interrupt our calculations on the nodes. We spot checked it by comparing the time the nodes needed to multiply their pieces. Remember that especially in the parallel Matrix Multiplication, the slowest Node will force the speed (after every squaring all nodes have to wait until the last Node has sent back his result).

Key data Intel Pentium 4 3.06Ghz (cpu family 15, model 4) with 1024 KB cache; 881 MB memory; Linux 2.6 Fedora 5

Programming Sun Java 1.6.0; Communication implemented with RMI

Network All computer systems were connected over a switched 100Mbit LAN reserved for only these measurements.

Communication delays

To verify the results we assume better communication paths between the nodes and the prover as we had in our lab for parallelization. Instead of computer systems connected through a slow 100MB Ethernet, we assume fiber optics and processor cluster transfer rates between two processors in a cluster.

- Fiber optic: Intel Pro 10Gbit fiber. 133Mhz clock with 10 Gbps throughput. [25]
- Bus system: Cray SeaStar2+ communication chip for each processor. 400MHz clock with 9.6 GByte per seconds throughput on every link (maximal 6). [26]

We take these two values to compute the performance of both parallel attacks if we assume faster communication paths. To add the theoretical communication delays to our data set we have to find out how many connections are needed for the parallelization and how many bytes are transferred. We did not measure the transferred data amount in the Java simulation so we have to roughly sum it up. This calculation is very roughly because we did not know the exact size of the (serialized) transferred object anyway.

Terminology

Gain factor: The y-axis in the graphs represents the gain factor the parallelization compared if the prover would solve the challenge alone. So the parallel computing is
y(= gainfactor) times faster than if the prover would compute is alone.

**Communication delays:** The communication delays not just include the time to transfer the objects over the LAN. It also includes the time a Node has to wait before he can send it because the prover is still receiving the response of another Node. Therefore we can see that sometime eight Nodes are slower than four nodes if the communications delays are not subtracted.

### 4.3.2 Gaussian elimination parallelization attack

Remark: The singularity of a random generated matrix is checked by using the fact that the matrix is singular or nearly singular if $A[p][p]$ from the Gaussian elimination algorithm (the actual value we want to subtract from the lower rows) is zero or nearly zero. This would throw a Runtime Exception and a new matrix had to be generated.

**Failed first attempt with successive parallel GE algorithm**

The original Gaussian elimination algorithm consists of a time intensive forward elimination and a cheap backwards substitution. The forward elimination over the full matrix takes approximately $n^3$ operations if $n$ is the size of the matrix. In this parallelization attack both nodes have first to do forward elimination for $\frac{n^2}{2}$ of the matrix. Therefore the parallelization advantage can not be greater than the remaining time after this half elimination. Unfortunately this first part is the most time intensive part because do the elimination to the half costs $\frac{7}{8}n^3$ and to eliminate the remaining $\frac{n^2}{2}$ matrix just $\frac{3}{8}n^3$. This is, because the elimination part has to do operations on every element of the remaining matrix. However, the $\frac{1}{8}n^3$ elimination has to be done anyway by a node. So actually the time we maximally can save by doing the parallel algorithm, is the time a single solving node has to backwards substitute the matrix. This part is saved in the parallel algorithm because we have parallel working nodes. Now the costs for the full backwards substitution is $\approx n$ and therefore the advantage the third root of the time a single node needs to solve the equations matrix alone.

In theory this is working nice and if I would have a fast cluster with $n$ processors probably also in practice. But I use Java as programming language and a 100Mbit network with computer systems which have $\approx 800$MB ram. The advantage of the parallel algorithm has to be clearly measurable. The available computers had a Java heap overflow ($> 1024MB$) during solving an 8000x8000 matrix after 10 minutes. It was not possible to measure the advantage because of not optimal coding and not controllable thread handling of e.g. RMI.

Even if we can power up our available computer systems, that they can solve much larger matrices within 6 hour, the advantage would be just $\sqrt[3]{36}h \approx 28$ seconds. And to measuring 28 seconds with a distributed Java application with a run time of 6 hours is quite hard to achieve at least for my skills. So we swapped to another parallelization schema with more advantage but on the other hand, more communication delays.
So we used the second parallelization algorithm.

Test parameters

- Number of nodes for the parallelization: \{2,4,8\}
- Matrix sizes: \{200,400,600,800,1000,1500,2000\}

This results in a graph where we can see the gain of parallelization in relation to the matrix size. We are also interested to see how the communication delays between the prover and the nodes influence this gain. The communication delay is the time needed to transfer benchmark requests and responses from the verifier to the prover and from the prover to the nodes (and vice versa). We measured every configuration point 15 times. So we get four graphs:

- Graph with removed communication delays (optimal case).
- Graph which includes the communication delays we had in our lab.
- Graph which includes the theoretical communication delays if we would connect our nodes over fiber optic lines.
- Graph which includes the theoretical communication delays if we would connect our nodes directly over a communication chip (processor cluster).

Calculate theoretical communication delays

First some definitions:

- \(n\): Size of matrix
- \(w\): Number of nodes
- \(s_n = n \cdot n \cdot 64\text{bit}\): Size of a \(n \times n\) matrix in bits (64bit = size of a double value)
- \(Tr\): Transfer rate of the communication lines in bits per second

If we have a \(n\) sized matrix \(A\), we will transfer approximately \(n\) times a matrix with an average size of \(s_n/2\) \((s_n \ldots s_1)\) to the nodes. We have to factor it with 2 because the nodes have also to respond the result. In our abstracted calculation the number of nodes \(w\) dropped away due the fact that the prover has to send the whole actual matrix to the nodes anyway (We do not care that he slice to several nodes). We also neglect matrix \(b\).

\[
\text{delay} = \frac{n \cdot s_{n/2}^2}{Tr}
\]

A cross check with \(80\text{Mbps}\) in our LAN gives us following results:
• 1000x1000 matrix. Calculated with $Tr = 80\text{Mbps}$: $\frac{1000 \cdot 16\text{Mbit} \cdot 280}{100} = 400s$. Average of the measured delay: 380s.

• 2000x2000 matrix. Calculated with $Tr = 80\text{Mbps}$: $\frac{2000 \cdot 64\text{Mbit} \cdot 280}{100} = 3200s$. Average of the measured delay: 3100s.

Results

Figure 4.1 shows the result with subtracted communication delays. The measured results including the communication delays of our 100Mbps LAN are shown in figure 4.2.

Figure 4.3 shows the theoretical results if we would have used a 10Gbps fiber optic connected environment and figure 4.3.2 if we would have connected the processor units over special cluster communication chips.
Figure 4.3: Gaussian Elimination parallelization results; Fiber optic communication delays

Figure 4.4: Gaussian Elimination parallelization results; Internal cluster communication delays
4.3.3 Matrix multiplication parallelization attack

The originally proposed Matrix Multiplication Benchmark has no mathematical limitation function included to prevent the numbers in the matrix to get too big (e.g. modulo). Therefore we had to work with the Java BigDecimal classes. A BigDecimal consists of an arbitrary precision integer unscaled value. The verifier initialize a matrix with one digit number. Every squaring round the digits are approximately doubled (plus $\text{digitsOf}(n) - 1$). After 10 squarings the values in the matrix have $\sim 5000$ digits. This results in a heavy cpu and memory consumption. This is the reason why we test the parallelization attack with small matrices and few squarings.

As we already mentioned, the attack is successful if the workload for a single matrix squaring is high enough. What is strongly depended on the matrix size and the size of the numbers in it. A successful attack with $n$ allies and $r$ squaring rounds is possible: If the $\text{time}_{P,\text{Alone}}$ the prover needs to square the matrix alone is greater than $\frac{1}{n} \cdot \text{time}_{P,\text{Alone}}$ (time one node needs to solve his pieces), plus the sum of the highest communication delays between the prover and the Nodes in each squaring round.

$$\text{time}_{P,\text{Alone}} > \frac{1}{n} \cdot \text{time}_{P,\text{Alone}} + \sum_{i=1}^{r} \max(\text{cdelay}_{\text{Node}1,\text{round}i}, \ldots, \text{cdelay}_{\text{Node}n,\text{round}i})$$

Remember that we assume equal fast computers for the prover and every node because every node get the same sub piece to solve and the slowest node limits the performance.

Test parameters

- Number of nodes for the parallelization: \{2,4,8\}
- Number of squarings: \{1,5,10,15\}
- Matrix sizes: \{8,32,128,256\}

Out of it we get a graph where we can see the gain of parallelization in relation to the matrix size and in another graph where you can see the gain in relation to the number squaring rounds. We fix the other value to 5, 10 squaring, respectively to a matrix size of 32, 128. We are also interested to see how the communication delays between the prover and the nodes influence this gain. The communication delay is the time needed to transfer benchmark requests and responses from the verifier to the prover and from the prover to the nodes (and vice versa). We measured every configuration point 50 times. So we get four graphs for the above four combination each:

- Graph with removed communication delays (optimal case).
- Graph which includes the communication delays we had in our lab.
- Graph which includes the theoretical communication delays if we would connect our nodes over fiber optic lines.
• Graph which includes the theoretical communication delays if we would connect our nodes directly over a communication chip (processor cluster).

Each for:
  • Graph gain - matrix size (fix 5 squarings).
  • Graph gain - matrix size (fix 10 squarings).
  • Graph gain - squarings (fix matrix size of 8x8).
  • Graph gain - squarings (fix matrix size of 128x128).

Calculate theoretical communication delays

Because the values in the matrix are growing rapidly with every additional squaring step, we had to use BigDecimal object to represent every value in the matrix. It is very difficult to know the size of a BigDecimal object, especially in serialized form. A BigDecimal object has several basic data types and an additional BigInteger as holder for the precise value. The BigInteger object has again several variables and an integer array to store the exact values byte for byte. After several tries to analytical compute the delays (and cross check it with the measure data over a 100Mbps LAN) we simply divide the communication delays through the factor which fiber optic and processor cluster communication is faster than a 100Mbps LAN. These factors are
  • Fiber optic: \( \frac{10 \text{ Gbps}}{100 \text{ Mbps}} = 100 \)
  • Processor cluster: \( \frac{9.6 \text{ Gbps}}{100 \text{ Mbps}} \approx 800 \)

Results

In the Matrix squaring parallelization attack we show the gain of parallelization in relation to the matrix size and also to the squaring rounds. Therefore we had to fix either the number of squarings or the matrix size. We have done this for a small and a large value each.

First the results with subtracted communication delays:
The measured results including the communication delays of our 100Mbps LAN:
Theoretical results if we would have used a 10Gbps fiber optic connected environment:
Theoretical results if we would have connected the processor units over special cluster communication modules:
Chapter 5

The SecRaCP protocol

In this chapter, the techniques for a secure benchmark are developed and we propose a secure benchmark protocol. The rough idea of our benchmark was already proposed before. However, we detail and improve the idea and focus hereby on the possible presence of malicious nodes. Additionally we propose a new faster verification method where the verifier outsources parts of the verification to the prover. An extended security analysis is also part of this chapter. At last we present the results of the SecRaCP protocol implementation, where we benchmark several nodes of the PlanetLab distributed server platform.

Contents

5.1 Modular squaring as benchmark code .......................... 51
  5.1.1 Fast modular squaring algorithms .......................... 54
  5.1.2 Parallelization .................................................. 55
  5.1.3 Montgomery ..................................................... 55
  5.1.4 Special groups ................................................... 57
5.2 Reduce verification effort ........................................ 58
  5.2.1 Use precomputed parameters ................................ 58
  5.2.2 Direct verifier optimizations ................................ 60
5.3 Sketch of SecRaCP ................................................. 64
  5.3.1 Verifier Setup .................................................... 65
  5.3.2 Protocol ........................................................... 65
5.4 Security analysis ..................................................... 68
  5.4.1 Resulting protocol assumptions ............................... 68
  5.4.2 Attacker model ................................................... 68
  5.4.3 Benchmark phase ............................................... 68
We have seen that the existing uncheatable benchmarks are not that uncheatable as they should be. We propose a new protocol called the Secure Ranking of Computational Power (SecRaCP) protocol. The verifier (Alice) wants to rank several nodes by their computational power abilities. To do that, she benchmarks each node (Bob) individually with the SecRaCP protocol. The idea is to let the nodes do multiple modular squaring. Every node proofs the knowledge of the unpredictable result to Alice.

1. Alice sends the parameters for the modular squaring inclusive a random seed value to Bob.
2. Bob executes the modular squaring operations as many times as requested.
3. In an additional verification phase, Bob has to proof the correct execution of the benchmark to Alice.

Clearly, the time Bob needs to do the modular squaring indicates his computational power. Alice can then rank all benchmarked nodes by their lower bound of available computational power. But let us start from scratch. Why is modular squaring ideal for benchmarking a system and how can Alice verify the correct execution of it. First we show the security of this protocol and how it actually works, and then we give an overview of the run time complexity of the verifier and the prover. At last the implementation shows the practicability in a realistic scenario.

5.1 Modular squaring as benchmark code

For the secure ranking protocol a benchmark code is needed

1. where the output value can not be predicted efficiently or computed faster than with this code if we input a random seed value
2. which can be repeated infinitely by using (parts of) the output as the new input and still satisfy condition one
3. which is not parallelizable
4. which needs relative high computational power to perform
5. which has a computational trapdoor

The first two conditions can be fulfilled with pseudo-random number generators. The variation of the Blum Blum Shub\[3\] generator as proposed in the paper about cryptographic puzzles\[2\] satisfies also the last three conditions. The proposal about time-lock puzzles\[2\], uses the same variant of the Blum Blum Shub number generator as we want to. They use it to encrypt a message that can not be decrypted by anyone, until a pre-determined amount of time has passed. To decrypt the message, a pre-defined amount of modular squaring rounds have to be done. To map rounds of modular squaring to run time, the exact modular squaring performance of the decrypter must be known. Actually, they introduce the idea of repeated modular squaring as a secure, un-parallelizable, computational workload code.

In our work we don not have to know the modular squaring performance exactly. It can be quite inexact since we use it to rank nodes in comparison to each other. Actually we can assume an average performance based on regular published performance tests of modern computer systems. We just have to be sure, that the communication delays are small enough compared to the modular squaring time.

The goal is to define a non-predictable value, where the computational power needed to compute this value, can be levelled to achieve a specific run time. With our used Blum Blum Shub pseudo-random number generator variant, such a number can be defined. Here is the approach: Suppose Alice wants to define such a value, that Bob needs a specific computational effort to compute it.

Alice generates random primes p,q such that

\[ p \neq q, \]

and

\[ p \equiv q \equiv 3 \pmod{4}. \]

Then

\[ n = p \cdot q \]

is called a Blum value. It is the modulus in our modular squaring system. She computes

\[ n := p \cdot q \]

, and generates a random number

\[ x \in \mathbb{Z}_n^* \]
and the following calculation

\[
\text{for } \cdots \text{ do } \\
\text{ } x := x^2 \mod n
\]

is done, until the specified computational effort is reached. If Alice knows the computational power of Bob in modular squarings per second, she can predict Bob’s run time for this computation.

Additionally \( n \) has to satisfy some common conditions to ensure that factorizing of \( n \) is hard. Be sure that \( p \) and \( q \) are large enough and they satisfy approximately \( 5 < (p/q) < 1000 \). To avoid certain attacks, \( p - 1, p + 1 \) and \( q - 1, q + 1 \) should have a large prime factor \( > \sqrt{n} \) respectively.

This algorithm has a trapdoor. If somebody can factorize \( n \) into \( p \cdot q \), he can compute

\[ r \text{ rounds of } x^2 \mod n \]

significant faster with

\[ x^{2^r} \mod (p-1)(q-1). \]

This is because if \( x \) is an element of \( \mathbb{Z}_n^* \) (requires \( \gcd(x,n)=1 \)). Lagrange’s group theorem states that the order of \( x \) divides \( (p - 1) \ast (q - 1) \)

because \( x \) is also a subgroup of \( \mathbb{Z}_n^* \). To compute

\[ x^{2^r} \]

we can first compute

\[ b = x^r \mod (p-1)(q-1) \]

and then we have

\[ \text{result} = x^{2^r} \mod (p-1)(q-1) \mod n. \]

The drawback is that it is difficult to evaluate the number of FLOPS needed for the calculation of such a small benchmark code. The core operation (modular squaring) has such a small load that other factors have a big influence. Linpack solves equations in form of \( n \)-sizes matrices (equation systems) and therefore has a large amount of multiplications and additions in every round. So they state their FLOP usage per round asymptotically with \( O(n^3) \). Because we do not need to state the metric FLOPS, but just have to compare a pseudo value, this drawback does not limit us.
5.1.1 Fast modular squaring algorithms

A good overview is given in "A survey of fast exponentiation methods"[15]. The best method for exponentiation depends on the used group and on the chosen parameters. For example, the repeated exponentiation with the same base or to the same exponent gives possibilities to parallelize modular squaring much more.

The question is how many modular multiplications (see addition chains chapter 5.1.1) are needed for modular exponentiation and how these modular multiplications are done (see Montgomery reduction chapter 5.1.3). For our benchmark we would prefer a situation where anybody has the same 'advantage' to compute the result and to avoid differences which are not relying on the computational power of the nodes. If we use benchmark code which is highly optimizable, we would not know which nodes have higher computational power or have just found a shortcut for the computation. We show that repeated modular squaring is a special case of modular exponentiation or generally a computation, where the optimization level is lower than for other useful benchmark codes.

Addition chains

An addition chain for $k$ is a list of positive integers

$$a_1 = 1, \ldots, a_l = k,$$

such that for each $i > 1$, there is some $j$ and $h$ with $1 \leq j \leq h < i$ and $a_i = a_j + a_h$. Due to the homomorphism, you can use $g^{a_i} = g^{a_j} \cdot g^{a_h}$.

Finding the optimal addition chain for an arbitrary exponent $k$ is an NP-hard problem[15]. The optimal computation of finite field exponentiation is a problem which is closely related to finding the shortest addition chain. The lower bound for the length of an addition chain for $k$ is $\lceil \log k \rceil$.

If we use not modular exponentiation $g^k \mod n$ but repeated modular squaring $g^{2^r} \mod n$, the shortest addition chain is exactly $\log k = r$. Improvements like addition-subtraction chains who allow other operations such subtraction are therefore not an advantage in computation. Even the use of new heuristics techniques[17] for finding optimal addition chains would be worthless.

Binary exponentiation

Binary exponentiation is also known as the "square and multiply" method. To compute $g^k$ you can use the binary representation of $k$,

$$k = \sum_{i=0}^{l} c_i 2^i,$$

where $l$ is the length of $k$.

The algorithm to compute $g^k$ is
\[ a = 1 \]
for \( d = 1 \) to \( 0 \) do
\[ a = a \cdot a \]
if \( c_d = 1 \) then \[ a = a \cdot g \]
end for

For repeated modular squaring this method is nothing else as the implementation of the optimal addition chain which is \( 1, 2, 4, 8, 16, \ldots, 2^r \).

**Window methods**

Read [15] too see that the sliding window algorithms are binary exponentiation if the exponent has a base of 2 (which is the case for our protocol).

### 5.1.2 Parallelization

Modular squaring is generally not parallelizable if you choose the parameter carefully (see 5.1.4), but single multiplications within single squarings are. It is clear, that as larger the numbers are, as larger is the parallelization advantage. We analyze the Montgomery multiplication approach, which is a fast modular reduction algorithm who can be also used in combination with binary exponentiation, to show that the parallelization advantage for our purpose is negligible. Read chapter 5.4.3 about our attempt to parallelize our specific protocol implementation.

### 5.1.3 Montgomery

Besides an appropriate algorithm to compute \( x^{2^r} \), we should also consider an algorithm for fast modular reduction to be used after every multiplication to compute \( x^{2^r} \mod n \). The paper of Bosselaers, Govaerts and Vandewalle [16] compares the classical, the Barrett’s and the Montgomery’s modular reduction algorithms. It is clearly showed, that Montgomery’s algorithm has an advantage if the numbers are large enough (\( \geq 1024\text{bit} \)).

To multiply two numbers with Montgomery, you have to transform them to the Montgomery representation, do a Montgomery multiplication and then transform the product back to the normal representation. Actually Montgomery avoids the costly remainder operation by keeping the sub results small when we are doing a long chain of modular multiplications like exponentiation or multiplications of large numbers. However, since transformation steps and the computation of \( n_t \) with the extended Euclid is time consuming, it is more suitable if there are several multiplications to the same modulus. This is the case for multiple modular squaring where the binary method computes the result with help of the optimal addition
We follow the argumentation and code proposed in "Analyzing and comparing Montgomery multiplication algorithms"[18]. Suppose we want to compute 
\[ c = a \cdot b \mod n \]
Montgomeryize \(a\) and \(b\) by multiplying them with the maximum operand size, say \(r = 2^i\), modulo \(n\). Multiplication modulo \(r\) and division by \(r\) can be done very fast because \(r\) is a power of 2.

\[ a \cdot r \mod n = a' \]
\[ b \cdot r \mod n = b' \]

And therefore

\[ c' = a' \cdot b' \cdot r^{-1} \mod n \]
\[ = a \cdot r \cdot b \cdot r \cdot r^{-1} \mod n \]
\[ = c \cdot r \mod n \]

We introduce now an additional quantity \(n_t\). Together with \(r^{-1}\) it can be computed with the extended Euclid to satisfy

\[ r \cdot r^{-1} - n \cdot n_t = 1, \quad r^{-1} \cdot r = 1 \mod n. \]

The multiplication of two already montgomerized number is done as follows:

function MonPro\((a', b')\)
Step 1. \(t = a' \cdot b'\)
Step 2. \(u = (t + (t \cdot n_t \mod r) \cdot n)/r\)
Step 3. if \(u \geq n\) then return \(u - n\) else return \(u\)

What we do, is to add a specific multiple of \(n\) to \(t\), so that we can divide (shift) by \(r\) without loosing information by 'cutting' non zero bit off.

Example: We have an integer 358 we want to compute modulo 131 and we anyway have to divide it through 10. To avoid modulo operations we add two times 131, what we are allowed to do, and we get 620 which is easily divisible by 10. So we calculated the result without a modulo operation and we have also used just the fast division through 10 where we just cut off the last digit if it is zero.

The binary method for \(x^k \mod n\) in combination with the Montgomery reduction looks then as follows:
function MonExp($x, k, n$)
Step 1. $x' = x \cdot r \mod n$
Step 2. $a' = 1 \cdot r \mod n$
Step 3. for $i = j - 1$ downto 0
    $a' = \text{MonPro}(a', a')$
    if $k_i = 1$ then $a' = \text{MonPro}(a', x')$
Step 4. return $a = \text{MonPro}(a', 1)$

Where $j$ is the number of bits in the exponent $k$ and $k_i$ is the $i$th digit of the exponent $k$. The implementations of large number operations are typically performed by breaking the number into words. In the Java BigInteger object, the multi-precision numbers are split up into integer. Therefore $r$ would be $r = 2^w s = 32^s$ where $w$ is the word size and $s$ the required words.

**Parallelization advantage:**

In most implementations, operations on large numbers are performed by breaking the numbers into words. If $w$ is the word size of the computer, then a number is a sequence of $w$-sized integers. Let us assume we divide the numbers in $n w$-sized integers. The usual long multiplication multiplies the multiplicand by each digit of the multiplier and then sums up all the properly shifted results. This $O(n^3)$ computation can be clearly parallelized. Also the $O(n \log_2 3)$ Karatsuba multiplication algorithm is parallelizable through three nodes every recursion step. In the Karatsuba multiplication the factors $a, b$ are recursively split into $a_1 a_2, b_1 b_2$ and

\[
\begin{align*}
r_1 &= a_1 \cdot b_1 \\
r_2 &= a_1 \cdot b_2 \\
r &= (a_1 + a_2) \cdot (b_1 + b_2) \\
\text{and then} \\
c &= r_1 \cdot 100 + t \cdot 10 + r_2 \text{ where } t &= r - r_1 - r_2
\end{align*}
\]

are computed. The communication overhead between computational nodes is too big if you compare it to the time needed to multiply two numbers which are smaller than half of the modulus used for the multiple modular squaring. Therefore the parallelization on this level is negligible in our proposed benchmark setting. For further analysis, read chapter 5.4.3, where we simulate the parallelization of multiple modular squaring. There we use the lowest latency and highest bandwidth between processes we found for supercomputers.

### 5.1.4 Special groups

If the parameters for repeated modular squaring, especially the modulus, are not carefully chosen, much easier computation of the result can be done. Exponentiation under modulo...
2^{593} can be done with only 129 multiplications[19]. Generally it can be said that modulus of kind of \( p^k \) and the use of the same exponent or base for several benchmarks should be avoided.

## 5.2 Reduce verification effort

By using the Euler totient, the verifier has an advantage over the prover to compute the result of the benchmark and with that to verify the correct computation. So we can reduce the effort from

\[
O(\log k \cdot M(n))
\]

to

\[
O(\log k' \cdot M(n) + \frac{\log k}{\log n} \cdot M(n))
\]

where \( M(n) \) is the time needed for a multiplication or division of two \( n \)-sized numbers. See Chapter 5.2.2 for details. However there are faster verification methods. First we propose one by using precomputed parameters. But this will give an attacker more place to cheat. So we analyze different methods to get a better direct verification method. We propose then a new verification method with a run time for the verifier of

\[
O(\frac{\log k}{\log n} \cdot M(n)).
\]

### 5.2.1 Use precomputed parameters

To comparable benchmark \( j \) systems, it would be very costly to compute every \( j \)-th trapdoor \( k_i = 2^r \mod \phi(n_i) \) and \( i \)-th result \( v_i = x_i^{2^r} \ (i = 1, \ldots, j) \) to verify the result. The idea is to precompute \( h \) results \( (h \ll j) \) and to use them to built \( j \) different benchmark settings which are fast verifiable.

- Pick two large prime number \( p, q \) as described in the protocol.
- Calculate the modulus \( n \) and the Euler totient function \( \phi(n) \).
- Pick \( r \) as the quantity of modular squaring rounds (empirical value to get a useful benchmark time at nowadays systems).
- Calculate \( k = 2^r \mod \phi(n) \).

The computational power of the benchmarked system are normally quite different. \( R \) rounds of modular squaring may not be enough to reach a useful benchmark time at every system. Remember the protocol, that if \( r \) rounds of modular squaring results in a too short benchmark
time, you have to start over and choose a higher \( r \). This proceeding is still valid. Choose a factor \( \alpha \) to multiply the quantity of rounds. Assuming that we use the same random seed value \( x \).

Let \( v = x^{2^r} \mod n \),

then we get

\[
x^{2^r \cdot \alpha} \equiv x^{(2^r)^\alpha} \equiv x^{(k+i\cdot\phi(n))^\alpha} \equiv x^{k\alpha+i\cdot\phi(n)} \equiv v^\alpha \mod n.
\]

Therefore we would be able to compute the chosen modular squaring rounds with the old result raised to a small factor \( \alpha \).

However, choosing the same seed value for every benchmark would not be a good idea. We can generate different seed values in the following way.

- Pick \( h \) primes \( x_1, \ldots, x_h, < n \).
- Calculate for every of them \( v_i = x_i^k \mod n \).

To benchmark one of the \( j \) systems, first create the seed value for it.

\[
x = (x_1^{c_1})(x_2^{c_2})\ldots(x_h^{c_h}) \mod n \text{ where } c_i \text{ randomly } \in \{0,1\}.
\]

The benchmark environment for the \( i \)-th system looks like that:

\[
((x_1^{c_1})(x_2^{c_2})\ldots(x_h^{c_h}))^{k\alpha} \equiv (((v_1^\alpha)^{c_1})((v_2^\alpha)^{c_2})\ldots((v_h^\alpha)^{c_h})) \mod n
\]

In average \( h/2 \) \((h \ll j)\) exponentiations with a very small exponent \( \alpha \) and \( h/2 \) multiplications are needed. \( H \) must be carefully chosen. It is at least \( h = \lceil \log(j) \rceil \) but large domain of possible seeds \( x \) will make it more difficult for malicious nodes to find common multiplicand (gcd()).

**Security analysis**

Let us assume the worst case where are all \( j \) benchmarked nodes are malicious and therefore share all generated \( x_i \)'s and \( h \) is chosen as \( h = \log(j) \) where \( j \) is directly \( 2^h \). The malicious nodes received all variations of the multiplied primes. But the sequence is randomly sent because the \( c_i \)'s where chosen randomly too. The digits of the values do not reveal anything because we compute in a modular group. If the primes are carefully chosen (large enough), factorizing would not be efficient enough to have a time advantage over the valid computation of the benchmark problem.
5.2.2 Direct verifier optimizations

We already optimized the verification time by using correlated parameter for several different nodes we want to benchmark. But direct verifier optimizations should reduce the verifier’s effort generally over just one benchmarked node. We have seen that with the Euler totient function a trapdoor exists to compute the result of repeated modular squaring. Even this gives the verifier a clear advantage, a $O(1)$ direct verification method would be preferred.

Notice that our benchmark handles very large numbers. The run time to 1024 bit numbers is not nearly the same as for the multiplication of 64 bit numbers. In order to not compare apples with pears, we introduce $M(n)$ which is the time needed to multiply (divide) two $n$-sized numbers. $n$ is directly the modulus we use for the multiple squaring and therefore most multiplications are done with $n$-sized numbers. Example: If we compute $2^r \mod n$ $(2^r >> n)$, this is not just a $n$-sized division with a remainder. We have to split $2^r$ into $\frac{\log 2^r}{\log n}$ blocks.

After several tries, we believe that no direct verification trapdoor $O(1)$ exists. This is just a conjecture and we argue this loosely.

We want to design a verification method which is independent of the number of modular squaring rounds. More precise we do not want to compute verification parameters with the term $2^r$ or $n$. Either we find a solution where we can shortcut the calculation heavily with the knowledge of a secret parameter,

$$(E(x, e_{\text{public}})^r = E(x, d_{\text{secret}}))$$

or we let Bob help us with the verification and Alice is just verifying if the correlation is correct. For example, we check if

$$v_{\text{result1}} = v_{\text{result2}} + \alpha_{\text{secret}}$$

holds. Here we let Bob compute two different benchmarks where we know the relation between the results of both, which is just valid if Bob computes the benchmark correct.

The problem with the first method is that the calculation of such a secret value implies the evaluation of $2^r$ or the exponentiation of the result with another high number. Say you can compute a secret key $d$, which is $d \cdot 2^r \equiv 1 (mod \phi(n))$ and let Bob compute $z^{2^r} \mod n$ where $z = x^d \mod n$. If Bob does return the result $x$, Alice will accept. But to compute the secret key and the exponentiation with the secret key would destroy the easy verification idea.

Remark: The only ‘free’ secret value Alice have, is $\phi(n) = (p - 1)(q - 1)$ which is not useful without additional expensive computations.

The second approach promises more scopes but the challenge here is to find such a correlation which is not cheatable. Because we do not want to precompute some heavy exponentiation, we send Bob all ‘raw’ computation parts like seed, modulus and number of rounds. With the correlations we found yet, Bob can easily replace the number of rounds in both benchmarks somehow to avoid computing the whole benchmark. By changing the rounds on both
benchmark requests, the correlation check value will be the same.

**New optimized verification**

Let

\[ v = x^k \mod n = x^{k'} \mod n \]

be the modular squaring result of our proposed benchmark, and

\[
\begin{align*}
    n &= pq \\
    k &= 2^r \\
    k' &= k \mod \phi(n) \\
    \phi(n) &= (p - 1)(q - 1)
\end{align*}
\]

Even if no \(O(1)\) verification for our benchmark exists, optimizations are possible. We know that the verification with the Euler totient trapdoor has a run time of

\[
O(\log k' \cdot M(n) + \frac{\log k}{\log n} \cdot M(n)),
\]

where \(M(n)\) is the time needed for a multiplication or division of two \(n\)-sized numbers. Raising \(x\) to the new exponent \(k'\) takes \(O(\log k' \cdot M(n))\) and calculating this new exponent \(k'\) takes \(O(\frac{\log k}{\log n} \cdot M(n))\) because the we split \(k\) into \(n\)-sized blocks to have \(M(n)\) multiplications and divisions. We propose an algorithm, that lets Bob do some part of the verification and we can therefore reduce the verification time to

\[
O(\frac{\log k}{\log n} \cdot M(n)).
\]

Compared to the run time complexity of the benchmark of

\[
O(\log k \cdot M(n)),
\]

the new verification advantage is

\[
O(\log n).
\]

The verification advantage is now independent of the exponent \(k = 2^r\) even if we can not choose \(n\) arbitrary large without reducing security of our benchmark.

**The idea:**

1. Alice measures the time Bob needs to compute the result of the multiple modular squaring \(v = x^k \mod n\).
2. Alice also knows the number of modular multiplications Bob had to do and can therefore estimate the computational power of Bob.

3. Alice does not have to compute \( v \) herself, to verify the correct computation.

4. Bob has just to proof knowledge of \( v \) by sending Alice the reduced exponent \( k' \).

5. Alice verifies if Bob sent her the correct \( k' \). If yes, Alice accepts the earlier estimated computational power of Bob.

In detail:

Alice starts our benchmark protocol by letting Bob compute \( v = x^k \mod n \). The verification advantage Alice has is that she knows \( p \) and \( q \) and therefore also the Euler totient \( \phi(n) = (p - 1)(q - 1) \). See Chapter 5.1 for detailed information about the Euler totient trapdoor.

Bob acknowledges finishing the computation of the result \( v \) as fast as possible. The main part of the benchmark is finished and Alice can estimate the computation power of Bob in modular multiplications per second. Alice has to ensure that Bob has computed all rounds of the multiple squaring to accept the estimated computational power.

Instead of verifying the result with help of the Euler totient trapdoor, Bob has to proof knowledge of \( v \) by his ability to compute \( k' \).

Bob knows that \( k' \) is smaller than \( \phi(n) < n \) and \( v = x^{k'} \mod n \). If he knows \( v \), he can try every exponent from 1 to \( \phi(n) - 1 < n \) to match this equation. There are few reason why Alice has to limit the number of modular multiplications which Bob has to do for finding \( k' \):

- The series \( x^1, x^2, x^3, \ldots \) is not successive in a modular group, so Bob can not use addition chains and has to do in worst case \( \phi(n) \) modular multiplications. So the verification phase would be more time consuming than the benchmark phase.

- Bob can parallelize the verification phase by using the log \( \phi(n) \) squaring results from the computation of \( v \), distribute them to other nodes which will compute the remaining values between every two squaring results. Therefore, as longer the verification phase is, as more a malicious prover can cheat.

- We know that Alice measures the time Bob has to finish the benchmark. This time is running also during the verification phase because with our new verification idea, Alice does not verify if \( v \) is correct but just that Bob knows \( v \) at the time when Bob also returns \( k' \). We want to minimize the time, Bob needs to compute \( k' \) if he knows \( v \), because this time is just in average the same for all benchmarked nodes. But we also want to minimize the probability that Bob can just guess the correct \( k' \), what we can influence with a larger range Bob has to search through.

To minimize the verification effort for Bob, Alice will give him a start exponent \( a \) and incrementor exponent \( b \). They are computed to satisfy

\[
v = x^k \mod n = x^{k'} \mod n = x^{a+b+\cdots+b} \mod n,
\]
what is actually
\[ k' = a + i \cdot b, \]
for a unknown \( i \) to Bob. The percentage \( p_{\text{veri}} \) defines how many times (at least) faster the verification phase for Bob is, compared to the benchmark phase. Alice computes the exponent incrementor as
\[ b = \left\lceil \frac{\phi(n)}{\log(p_{\text{veri}})} \right\rceil \]
and the start exponent as
\[ a = k' \mod b. \]
There is an additional condition, that \( b \) has to be divisible by 4. We discuss the reason and how to handle that, in the security analysis.

Let us have a look at the complexity. Alice sends \( a \) and \( b \) to Bob. Bob has to compute the start point \( x^a \mod n \) and \( x^b \mod n \) himself with addition chains and cost of \( O(\log a + \log b) \) and can then start to multiply it with \( x^b \mod n \) until he gets the result \( v \). If Bob really knows the correct result of the multiple modular squaring, he is able to find \( k' \) after at most \( \frac{\phi(n)}{b} \) steps.
The verification phase

\[ n \quad \text{modulus} \]
\[ \phi(n) \quad \text{Euler totient} \]
\[ r \quad \text{squaring rounds} \]
\[ k' \quad \text{reduced exponent} \]
\[ v \quad \text{result of the multiple modular squaring} \]
\[ t_b \quad \text{measure benchmark phase run time} \]
\[ t_{Vmax} \quad \text{maximal verification phase run time} \]
\[ p_{veri} \quad \text{verification phase run time as percentage of the benchmark run time} \]
\[ a \quad \text{verification start exponent} \]
\[ b \quad \text{verification exponent incrementor} \]
\[ t_i \quad \text{timestamps} \]

Alice
\begin{itemize}
  \item given \( t_b, r, k', \phi(n) \)
  \item compute \( b = \left\lceil \frac{\phi(n)}{r/P_{veri}} \right\rceil \)
  \item compute \( a = k' \mod b \)
  \item compute \( t_{Vmax} = \frac{t_b}{P_{veri}} \)
\end{itemize}

Bob
\begin{itemize}
  \item given \( v' \)
  \item find \( k'' = a + i \cdot b \)
  \item so that \( v' = x^{k''} \mod n \)
\end{itemize}

\[ t_1 \quad a,b \]
\[ t_2 \quad k'' \]
\[ t_2 \]
\[ \text{verify } k' = k'' \]
\[ \text{and } t_3 - t_2 \leq t_{Vmax} \]

5.3 Sketch of SecRaCP

The protocol is divided in following phases.

- Preparation phase: Alice will generate all needed parameters.
- Benchmark phase: Bob has to compute the result of the modular squaring benchmark.
- Verification phase: Bob proofs knowledge of the result to Alice.
5.3.1 Verifier Setup

The verifier (Alice) first measures the average RTT (round trip time) to the node. If this is not possible, she can assume an empirical maximum. For example 200ms over public internet communication paths.

*Average RTT: t_{rtt}*

The network between the verifier and the prover is normally not under control of the verifier. The communication delays can be quite unstable. Alice chooses a maximal error percentage the communication delays may have on the measurement. If she chooses a low error rate, the node has to compute more rounds of modular squaring to minimize the influence of fluctuating communication delays, intent end or not.

*Max communication error percentage: p_{error}*

Out of \( t_{rtt} \) and \( p_{error} \) Alice can compute the minimal modular squaring time for Bob, so that the unstable communication delays do not influence the benchmark to heavy.

*Min benchmark time: t_{min} = \frac{t_{rtt}}{p_{error}}*

Before Alice can start the protocol, she has to roughly know Bob’s computational power to compute the needed rounds of modular squaring. Actually this is the number of rounds needed to let Bob do modular squaring for minimum \( t_{min} \).

*Number of rounds: r*

This is an empirical value. The modular squaring rounds have a small load, hence pre-evaluating this value is impractical. Therefore Alice will start testing Bob with a number of rounds which Alice knows will result in a benchmark time of approximately \( t_{min} \), if Bob has average computational power. If the measured time is smaller than \( t_{min} \), Alice will increase the rounds and restart the benchmark again.

Alice has also to compute the Blum Blum Shub divisor \( n \) (see Chapter 5.1).

*Blum Blum Shub divisor: n*

Alice defines also how many percent of the actual benchmark time (\( t_b \)) the maximal verification time should be. If \( k' = \phi(n) - 1 \), the verification time is maximal.

*Verification time percentage: p_{veri}*

If Bob needs more time for the verification phase than \( t_{Vmax} \), the benchmark is not accepted.

*Maximal verification time: t_{Vmax} = t_b \cdot p_{veri}*

The verification percentage \( p_{veri} \) can not be chosen completely free as we will explain later. It influences the benchmark security which is the probability that Bob can arbitrarily cheat the protocol.

*Benchmark security, cheating probability: b_{sec} = \frac{1}{r \cdot p_{veri}}*

5.3.2 Protocol

Each step has to be done for every prover. To rank several nodes, Alice does the preparation phase for all nodes and then starts the benchmark phase in parallel.
Preparation phase

1. Alice creates the parameters for the modular squaring systems \((p, q, n, \phi(n), x)\).

2. Alice chooses the minimal run time of the benchmark \(t_{\text{min}}\), the default squaring rounds \(r\) and the verification time percentage \(p_{\text{veri}}\).

3. Alice generates the resulting verification parameters, the start exponent \(a\) and the exponent incrementor \(b\).

Benchmark phase

1. Alice sends Bob the public parameters and waits for a response.

2. Bob computes the result of the modular squaring and informs Alice when he has finished the calculation.

3. Alice checks if the measured time is smaller as the defined minimal run time. If the measured run time is smaller, new parameters with an adapted number of squarings \(r\) is sent to the prover again. If the measured run time is equal or higher, Alice starts the verification phase.

Modified verification phase

1. Alice sends the verification parameters \(a, b\) to Bob.

2. If Bob really has the estimated computational power, he can compute the new exponent \(k'\) within the defined time \(t_{V_{\text{max}}}\).

3. Alice accept the benchmark results, if Bob can send her the correct \(k'\) within the defined time. If not, Alice will exclude the benchmarked node or simply restart the benchmark.

Read Chapter 5.1 for the modular squaring theory and Chapter 5.2.2 for the verification theory.
The SecRaCP protocol

- $p, q$: random primes
- $n$: modulus
- $\phi(n)$: Euler totient
- $r$: squaring rounds
- $k'$: reduced exponent
- $x$: random squaring base
- $v$: result of the multiple modular squaring
- $t_{\text{min}}$: minimal benchmark phase run time
- $t_{V_{\text{max}}}$: maximal verification phase run time
- $t_b$: measured benchmark phase run time
- $p_{\text{veri}}$: verification phase run time as percentage of the benchmark run time
- $a$: verification start exponent
- $b$: verification exponent incrementor
- $t_i$: timestamps

**Alice**

- Given $r, t_{\text{min}}, p_{\text{veri}}$
- Pick random primes $p, q$
- Compute $n, \phi(n), t_{V_{\text{max}}}$
  - Pick $x \in_R \mathbb{Z}_n$
  - Compute $k' = 2^r \mod \phi(n)$
  - Compute $b = \left\lfloor \frac{\phi(n)}{r/p_{\text{veri}}} \right\rfloor$
  - Compute $a = k' \mod b$

**Bob**

- Compute $v' = x^{2r} \mod n$

- Compute $t_2$
  - If $t_b \leq t_{\text{min}}$, restart
  - Compute $t_{V_{\text{max}}} = \frac{t_b}{p_{\text{veri}}} t_3$

- Compute $v' = x^{k''} \mod n$
  - $k = a + b + \cdots + b$

- Verify $k' = k''$
  - And $t_4 - t_3 \leq t_{V_{\text{max}}}$
5.4 Security analysis

5.4.1 Resulting protocol assumptions

The modular squaring rounds are chosen, so that the network delay is less than a specified percentage of time needed for modular squaring. This reduces the influence of unpredictable network delays. On the other hand, a higher benchmark time can give malicious nodes the chance to communicate with other nodes (additional communication time is small compared to the benchmark time). But they can not efficiently parallelize the executing and if the node outsources the whole modular squaring to a faster system, he actually has more computational power available.

If a node claims to have multiple processor units available, the verifier sends also multiple benchmark request to the node which must be parallely executed and answered in time. The protocol securely ranks the nodes by the lower bound of their available computational power. The upper bound of computational power can not be benchmarked because every node can slow down the executing of the benchmark code arbitrarily. Therefore the pretesting approach is more useful.

5.4.2 Attacker model

The protocol is between two entities, the verifier and the node. We assume that none of these entities is compromised and they have access to a secured communication line. The node does not act malicious but just fraudulently what means that the node will not completely interrupt the benchmark and will provide his whole available computational power, once he is selected as service provider. But he can try to claim a higher computational power during benchmark.

The node and/or the attacker are computational bounded. If the attacker is the node itself, we assume that he can do arbitrary computations and modifications on the benchmark parameters and the return values. Additionally he can arbitrarily communicate with other entities. If the attacker is not the node itself, we assume that he can act malicious and have access to communications paths to the verifier and the node but can not eavesdrop or modify transmitted messages between the verifier and the node.

5.4.3 Benchmark phase

To proof the security of the protocol we will show soundness and completeness. Let $cp$ the real available computational power and $cp'$ a faked higher computational power.
The protocol is *sound* if there is no strategy for Bob to fake a proof of higher available computational power (that Alice accepts).

The protocol is *complete* if Alice accepts a proof from Bob of his real available computational power.

Further we will discuss the efficiency of the computation done by Alice and Bob. We will also see, that the type of security is based on cryptographic assumptions rather than on information theoretical ones.

The goal of Alice is to measure the available computational power of Bob what she can interpret as a lower bound. So if Alice chooses Bob as a service provider, she can be sure that Bob can provide at least the measured computational power. Due the ranking of all benchmarked nodes, Alice can choose the nodes with the highest proofed lower bound of computational power. Therefore, the goal of Bob is to proof a computational power which is as high as possible. As we stated in the attacker model, he has no interest in claiming a lower computational power. The goal of an attacker (Eve) is to increase or reduce the measured computational power. If Eve tries to increase the measured computational power she is actually working for Bob, so we will analyze this case as if Bob is the attacker. If Eve tries to lower the measured computational power she is working against Bob and with that for another node which tries to get a better ranking.

Therefore, there are two different attack scenarios. In the first one, the benchmarked node itself or with external help tries to claim higher computational power. In the second, other nodes try to disturb the protocol to make the verifier believe that the benchmarked node has less computational power than it actually has.

**Fraudulent node**

**Definition:** The fraudulent node attacks the protocol to claim a higher computational power that it really has. As mentioned before, he wants to collaborate but tries to answer the benchmark request faster than he normally is able to, according to his computational resources.

**Soundness:** The requested computational effort is modular squaring. But is there another possibility to get the correct result? Modular squaring is an operation stated as hard as factoring the modular divisor $n$ to cheat. As long as the defined conditions are followed, the described trapdoor can not be used by the prover. No other faster computation is known yet. The possibility to guess the correct result is negligible (e.g. $2^{-1024}$) if again, the divisor $n$ is properly chosen.

We already stated that nothing else than modular squaring is needed to get the correct result. The remaining question is if there are unknown fast algorithms for modular multiplication available and if those can be parallelized? We assume that because of the fundamental operation of modular squaring, the optimization potential is smaller than
for more complex calculations. It is important to see, that our protocol is using modular squaring and not just modular exponentiation for the estimation of the computational power (see also Chapter 5.1.1). The reason is that it is NP-hard to find the smallest addition chain for an arbitrary exponent and the length of the chains is correlated to the number of needed multiplication operations. But what we want is a computation where we can know the amount of modular multiplications needed to get the result. If we let the exponent be a power of 2, the minimal addition chain is easy to find \((2, 4, 8, 16, \ldots)\) and we reduced the possible variation of needed multiplication operations.

And as also stated before (see chapter 5.1.2), modular squaring is not parallelizable and outsourcing the whole computation is not a threat. The computation is not heavily parallelizable and if the attacker can outsource the computation to a faster node, he actually has access to this resource and can therefore provide this computational power.

**Completeness:** If no external malicious attacker is present, Bob can compute the requested rounds of modular squaring as fast as he can. The communication delays are then within the allowed bounds and Alice will accept the returned value and the measured time as valid.

**Malicious attacker**

**Definition:** This malicious external attacker (Eve) wants to make Alice believe that Bob has a lower computational power than he really has.

**Soundness:** Soundness in this case means, that no third party can negatively influence the benchmark between Alice and Bob. According to our attacker model, Eve can not eavesdrop or modify messages between the verifier and the node. Trials by the attacker to delay messages be detected by inserting timestamps. If a message needs unrealistic amount of communication time, the protocol will be restarted. Adding undetectable small delays would not effect the result more than by the preset maximum error probability \(p_{\text{error}}\). However the attacker can try to slow down the benchmarked node by e.g. distributed Denial of Service. If the node has a web server running, the malicious nodes can flood this service with massive request and therefore reduce the available computational power for the modular squaring. This attack is difficult to prevent if the web server should stay available for everyone. But if possible, the node can close down all other services during the benchmark time. The close down is best made through an independent external firewall who blocks all other traffic.

**Completeness:** Eve can force a re-benchmarking if he can delay the messages between Alice and Bob so that the protocol will be stopped and restarted. If Eve is able to do that as long as needed, Alice can be precluded from benchmarking Bob. So the completeness can not be fully guaranteed.
Parallelization attack

As we stated before, we have an advantage of \( \log n \) for the Verifier over the prover. If we would parallelize the binary exponentiation (with Montgomery reduction), we would have to parallelize a 1024 bit multiplication and the resulting Montgomery reduction. The effort to do this is approximately 32 integer multiplications which are done also on slow nowadays computer systems in less than 1 millisecond. Measurement within the Java implementation of our protocol confirmed that. The time needed to do one 1024 bit multiplication and the related reduction is less than 1 millisecond. Even if we would assume very fast cluster communication delays, a parallelization of our protocol would not be possible respectively not be useful as also already stated by Rivest, Shamir and Wagner \[2\]. See 5.1.2 for detailed information about the parallelization possibilities.

Here a simulation of the parallel attack of our protocol and under what circumstances it would be successful.

We assume a computer which is 20 times faster than the laptop stated below:

**model name:** Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz (running just on one cpu; checked with system monitor)

**cache size:** 4096 KB

We take the measured run time (laptop) for the squaring but then assume 20 times more squaring rounds for the ‘virtual’ system.

Reference: Fastest systems comparing the latency\[28\]. The values are

- Random Ring Bandwidth: 0.3 Gbyte/s
- Random Ring Latency: 0.001 ms

Explanation of these parameters\[28\].

**Random Ring Bandwidth (per process):** Randomly Ordered Ring Bandwidth, reports bandwidth achieved in the ring communication pattern. The communicating processes are ordered randomly in the ring (with respect to the natural ordering of the MPI default communicator). The result is averaged over various random assignments of processes in the ring. Unit: Giga Bytes per second

**Random Ring Latency (per process):** Randomly-Ordered Ring Latency, reports latency in the ring communication pattern. The communicating processes are ordered randomly in the ring (with respect to the natural ordering of the MPI default communicator) in the ring. The result is averaged over various random assignments of processes in the ring.

In table 5.4.3 we take a bit length of 1024 and 14000k squaring rounds to compute the parallelization advantage with the above defined parameters. Here a short legend for the
**Rt total:** Runtime for the computing modular squaring result.

**Rt parallel:** The run time of the part which can be parallelized. This is the actual multiplication of two 1024 bit numbers and the related Montgomery reduction.

**Nodes:** Number of additional nodes are helping the prover.

**Delay:** The communication delays computed with the formula stated.

**Advantage:** The advantage of the parallelization.

Computation of the theoretical communication delay:

\[ 2 \times \text{Nodes} \times \text{Rounds} \times (\text{latency} + \text{transferTimeForXbitNumber}) \]

We can clearly see, that for numbers of at least 1024 bit, the parallelization of the multiple squaring algorithm has no advantage. If the parallel computing with one additional node should lead to an advantage, we would need a latency of less than 187\(\mu\)s. Due of lack of time, we did no tests with multicore processors and whether or not the communication delays between these cores are small enough. But generally we can say, that if they were fast enough, we can also accept those processors as one single computational unit. The verifier may also define, that the sum of all computational power of all cores will be taken for the ranking. Trying to parallelize the benchmark code onto several cores would then be useless since the parallelization will produce an overhead anyway.

<table>
<thead>
<tr>
<th>Rt total (ms)</th>
<th>Rt parallel (ms)</th>
<th>Nodes</th>
<th>Delay (ms)</th>
<th>Advantage (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11249</td>
<td>10481</td>
<td>1</td>
<td>39946.67</td>
<td>-34706.17</td>
</tr>
<tr>
<td>11262</td>
<td>10391</td>
<td>1</td>
<td>39946.67</td>
<td>-34751.17</td>
</tr>
<tr>
<td>11215</td>
<td>10546</td>
<td>1</td>
<td>39946.67</td>
<td>-34673.67</td>
</tr>
<tr>
<td>11240</td>
<td>10907</td>
<td>1</td>
<td>39946.67</td>
<td>-34493.17</td>
</tr>
<tr>
<td>11126</td>
<td>10774</td>
<td>1</td>
<td>39946.67</td>
<td>-34559.67</td>
</tr>
<tr>
<td>11219</td>
<td>10203</td>
<td>1</td>
<td>39946.67</td>
<td>-34845.17</td>
</tr>
<tr>
<td>11311</td>
<td>10321</td>
<td>1</td>
<td>39946.67</td>
<td>-34786.17</td>
</tr>
<tr>
<td>11267</td>
<td>10442</td>
<td>3</td>
<td>119840</td>
<td>-112008.5</td>
</tr>
<tr>
<td>11207</td>
<td>10968</td>
<td>7</td>
<td>279626.67</td>
<td>-270029.67</td>
</tr>
<tr>
<td>11288</td>
<td>10431</td>
<td>15</td>
<td>599200</td>
<td>-589420.94</td>
</tr>
</tbody>
</table>
5.4.4 Verification phase

This security analysis handles the additional problems due to the new verification method. As mentioned before, Bob can cheat in the second part of the protocol. This results in inexactness of the estimated computational power which is not negligible but can be limited through configuration to be insignificant. Recall that

- The protocol is *sound* if there is no strategy for Bob to fake a proof of higher available computational power that Alice accept.
- The protocol is *complete* if Alice accepts a proof from Bob of real available computational power.

We analyze the new verification idea towards these two conditions.

**Soundness:** There are two possibilities to successfully claim higher computational power. Guess $k'$ or parallelize the verification part of the protocol. We will show that the first is negligible and the second insignificant.

**Guessing $k'$:** Of course Bob can also guess just $v$, but we already stated that the range given by $a, b$ is smaller than $n$ and therefore also the chance to guess it right. The probability that Bob can cheat is now directly depended on the size of the sequence within Bob has to look for $k'$. Either Bob computes $v$ correctly and can compute $k'$ within an appropriate time or he just guesses it. The probability of guessing $k'$ can be set by the verifier. Read also chapter 5.4.4 on how predictable $k'$ is.

**Parallelize verification:** Let us look at the worst case. Suppose Bob is the fastest node from a grid system. After Bob gets $x, r, n$ he forwards $n$ to all other nodes in the grid. During Bob is calculating $v$, the other nodes compute distributed every value $x^2, \ldots, x^{n-1}$. Once Bob has computed $v$, he broadcasts $v$ again to all other nodes, and the node which has computed this value sends the corresponded exponent back. Therefore Bob can continue computing $v$ also during the actually verification phase because he does not need to search $k'$ himself. He states too early that he have computed $v$ and Alice will estimate a higher computational power. Again, the maximal cheating influence can be set by the verifier with $p_{veri}$.

**Other considerations:** Sending $a, b$ directly with the parameters $x, r, n$ or not sending them until Bob finished computing $v$, does actually not make a huge difference. But the later we send $a, b$ the less time Bob and his allies have to precompute possible $x_i^k$ values.

**Completeness:** Bob can compute $k'$ within the limited sequence if he knows $v$. Then Alice accepts Bob’s proof of knowledge of the correct $v$ and accepts also the resulting computational power.
Remark:
Alice knows exactly the computational effort of Bob, to compute \( v \) and \( k' \). So why not compare the number of modular multiplications per seconds for the benchmark phase and the verification phase? We can take at the end the lower of both values.

- First problem is that the available computational power of Bob can fluctuate quite heavily. Other processes or system interrupts can tamper the continuous execution of the benchmark. The verification phase runs several times faster than the benchmark phase. Peaks in both directions won’t be averaged enough. This can result in heavy discrepancies.

- The second problem is, that the verification phase can be parallelized by the prover. To just compare the modular multiplications per seconds is then useless. That is why we define an upper bound for the verification run time. If it is not exceeded, the cheating error is not higher than we defined it.

**Predictability of \( k' \)**

It is very important to know, how predictable \( k' = k \mod \phi(n) \) is. Naively we would say, that \( k' \) is a random value between zero and \( \phi(n) - 1 \) because \( p, q \) and with that \( \phi(n) \) are random numbers.

We will show, that \( k' \) is uniformly distributed and the prover just knows, that it is divisible by 4.

This implies that the incrementor the verifier sends to the prover during the verification phase, is also divisible by 4. The computed start exponent will then be automatically also divisible by 4 and every successive exponent too.

Why \( k' \) is divisible by 4. We know that \( \phi(n) = (p - 1)(q - 1) \) is divisible by four because each factor is divisible by two. The exponent \( k = 2^r \) is also divisible by four because it has the base of 2. The reduced exponent \( k' = k \mod \phi(n) \) can also be written as \( k' = k - i \cdot \phi(n) \).

The minuend and also the subtrahend are therefore divisible by four. Therefore \( k' \) has also to be. Higher common divisors are not imperative as \( \phi(n) \) does not satisfy any other useful
regularity. A short Java simulation, where we computed several times the gcd of 100 thousand different \( k' \), confirmed us that the greatest common divisor is 4.

Why the incrementor should also be divisible by 4 and how to handle that condition. As we know, the prover will raise \( x \) to the initial exponent \( a \) and will multiply it by \( x \) raised to the incrementor \( b \) value until he finds \( v \). So if the series \( a, a + b, a + b + b, \ldots, k' \) is not always is divisible by 4, the prover although has to do all modular multiplications if he wants to find \( k' \) by finding first the correct \( v \) value. But a malicious prover can also just try to guess \( k' \) without running the benchmark. We defined the probability that he can guess the correct value as \( \frac{\phi(n)}{b} \). But if some sub steps result in an exponent which is not divisible by 4, we would be able to exclude these, and with that lower the security of the verification method. To ensure that the incrementor value is divisible by 4, we can just reset the last less significant bits after we computed the incrementor initially. The only effect can be a negligible longer run time for the prover, because he had to do some few more multiplications steps to find \( k' \).

Now we take a look at the distribution of \( k' \). We know now, that the gcd is 4. But are they also uniform distributed between zero and \( \phi(n) \)? We classify every \( k' \) within which percentage (1\% – 100\%) it is compared to \( \phi(n) \). The distribution of one million \( k' \) with fixed \( k = 2^r \) and random \( p, q; \phi(n) = (p - 1)(q - 1) \) where we use 1024 bit system. We do it for three different \( k = 2^r \).

If we would use less digits or if we have to generate that much different \( p \) and \( q \) (resp. \( k' \)) that we are afraid of generating too much equal \( k' \) with a fixed \( k \), we can just use for \( p, q \) strong primes. If \( p, q \) are strong primes, \( p - 1 \) and \( q - 1 \) have at least one big prime factor. Therefore all generated \( \phi n \) would be with a high probability, co-prime \( (gcd(\phi_1, \phi_2) = 1) \) what means that both \( k' \) are with high probability not the same.
Figure 5.2: Distribution of $k'$ with 2 million squaring rounds

Figure 5.3: Distribution of $k'$ with 123456789 squaring rounds
5.5 Runtime analysis

\[ p, q, n = pq \] Parameters for the modular field
\[ k = 2^r \] Exponent the prover has to raise the base
\[ x \in \mathbb{Z}_n^* \] The base the prover has to use for the repeated modular squaring
\[ \phi(n) = (p-1)(q-1) \] Euler totient which can be used as a trapdoor for the computation of \( v \)
\[ v = x^k \mod (n) \] The result of the repeated modular squaring
\[ k' = k \mod \phi(n) \] The reduced exponent
\[ a < \log n \] \( x^a \) is the start value, from where the prover has to search for \( k' \)
\[ b < \log n \] \( b \) is the incrementor used to compute \( x^{a+b}, x^{a+b+b}, \ldots, x^{k'} \) until \( k' \) is found

<table>
<thead>
<tr>
<th>Description of the task</th>
<th>run timea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verifier</td>
<td></td>
</tr>
<tr>
<td>Generate ( p, q (1 - 2^{-c} \text{ certainly}) )</td>
<td>( \sim O(c \cdot \log^2({p, q})) ) each</td>
</tr>
<tr>
<td>Compute parameters ( n, \phi(n) )</td>
<td>( \frac{2}{\log k} )</td>
</tr>
<tr>
<td>Compute verification value ( k' )</td>
<td>( \frac{\log k}{\log n} )</td>
</tr>
<tr>
<td>Compute verification start values ( a, b )</td>
<td>( \frac{2}{\log n} )</td>
</tr>
<tr>
<td>Total</td>
<td>( \sim O(\frac{\log k}{\log n} + (\log n)^2) )</td>
</tr>
<tr>
<td>prover</td>
<td></td>
</tr>
<tr>
<td>Compute ( v )</td>
<td>( \log k = r )</td>
</tr>
<tr>
<td>Compute ( k' )</td>
<td></td>
</tr>
<tr>
<td>Compute ( x^a \mod n )</td>
<td>( \log x^a = a &lt; \log n )</td>
</tr>
<tr>
<td>Iterate with ( b ) until ( k' ) found</td>
<td>( \frac{k'-a}{b} )</td>
</tr>
<tr>
<td>Total</td>
<td>( \sim O(\log k + \log n) )</td>
</tr>
<tr>
<td>Advantage of the Verifier over the prover (without computing primes)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>( \sim \log n )</td>
</tr>
</tbody>
</table>

\( a \) modular multiplications of \( n \)-sized numbers

That result implies, that we should make \( n \) really large. On the other side, this would increase the chance of a parallelization attack. If \( n \) has 100'000 digits, the time for a single multiplication with the related Montgomery reduction takes on a nowadays notebook about 25 milliseconds (measured in our Java implementation of our protocol). Another point is, that the verifier can define how many times the verification phase should be faster than the benchmark time (computing of \( v \)). If we take a time which for the calculation of \( v \) just 10000 squarings are necessary (if we use large numbers, less squarings for the same computation time is needed), and the Verifier chooses \( p_{veri} = 1000 \), the malicious prover has a chance of
to guess $k'$.

5.6 Implementation of SecRaCP

5.6.1 Setup

Programming language: We used Sun Java 1.6.0_11 JRE for the Verifier as well as for the prover. For extended studies of the used algorithms and also for performance measuring, we used and customized the sources files from the Sun Java JDK 1.6.0_11.

Communication framework: For the communication between the verifier and the provers, we chosen the JADE[29] framework. JADE (Java Agent Development Framework) is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications.

PlanetLab: For the systems of the verifier and the provers we used a global search network called Planetlab[30]. Planetlab is a platform for developing, deploying, and accessing planetary-scale services. You can add several nodes from all over the world into your account. Even if these nodes are just virtual server instances, a protocol between these nodes can be tested in a real life scenario. The access to this nodes are done over public key authenticated SSH connections. Every new instance has a basic installation of Fedora Linux on it. It is possible to install additional packages.

Size of numbers: The modulus is a 1024 bit number and limits therefore the used number in the system also to 1024 bit.

Random primes: We are using the java.math.BigInteger.probableprime(bigLength, SecureRandom) method to get a bitLength-sized prime. The java.security.SecureRandom() function satisfies the FIPS 140-2 security requirements for cryptographic modules and is cryptographic strong as defined in rfc1750. The probablePrime() methods verifies the primality of numbers with the Rabin-Miller algorithm and has a certainty of more than $1 - 4^{-c}, c = 100$ where the run time is $\sim O(c \cdot \log^2(p, q))$.

Modular powering: We are using java.math.BigInteger.modPow(). This uses the binary exponentiation algorithm which gives us the smallest addition chain since our exponent has base 2 (see 5.1.1). Additionally it makes use of Montgomery reduction to speedup the modulo operations (see 5.1.3). We are aware of the fact, that probably faster implementations exist. Due of time limitations we use the binary exponentiation with Montgomery reduction for our implementation. Due the best of our knowledge, the usage of an exponent of base 2, limits the advantage of a malicious prover which are using some advanced new algorithms. Mainly because it is faster to square the base $x$
every time and use the result as the new base than keeping the base and multiply it with a special algorithm.

5.6.2 Results

In table 5.6.2 and 5.6.2 we see the result of our ranking protocol against several random chosen Planetlab nodes. One is sorted by the computational power index Mms (modular multiplications per second). For easier comparison of our metric Mms with the actual available CPU time, the second table is primarily sorted by the CPU type and secondarily by the idle CPU time. The following columns are shown.

Hostname: The absolute hostname of the server, which hosts our virtual server instance.

CPU id: CPU identifier (/proc/cpuinfo). It is irrelevant whether the system has one or ten processors. The repeated modular squaring does run as one Java thread and therefore just on one processor (core). We double checked this also with a system monitor. But of course, if several CPU or cores are available, the available CPU time tends also to be higher. Theoretical it can be, that the shown idle time is splitted to two or more processors. But it seems, that the virtualization manager which assigns resources to each instance, does just assign one processor to each instance. Otherwise we would have irregularities on nodes with more than one processor.

Idle: CPU idle time for our virtual server instances right before we start our benchmark.

Mms: Our computational power metric of the protocol. This is actually the number of modular multiplications per milliseconds the prover reaches for the computation of the benchmark result $v$.

Verification: The prover’s verification phase effort compared to his benchmark effort. As we mentioned earlier, the verification phase can be parallelized by the prover. About this percentage, the prover can claim higher computational power as he actually have.

Security: The probability that the prover can cheat the whole benchmark by doing no computation and just guess the new exponent $k'$.

Even if the verifier can choose the ratio between the prover’s verification and benchmark time, the parameter can not be chosen completely free. The parameters had to be adjusted according to the requirement of the benchmark. So if we chose the verification time to small, the number of possible $k'$ values would also get smaller and the security would decrease. Let us see what happens if we choose the ratio between the benchmark time and the verification time too high. For a ratio of 1% or 0.1% the cheating probability behaves as it should. But if the verification time is getting to small, the overhead is too big and we defined level can not be hold. See table 5.6.2. If we used e.g. 0.01%, we had to increase the big length or the squaring rounds. But as stated above, these two parameters have also their disadvantages, if they are too large.
Table 5.2: SecRaCP results, sorted by CPU identifier and Idle CPU time

<table>
<thead>
<tr>
<th>Hostname</th>
<th>CPU id</th>
<th>Idle</th>
<th>Mms</th>
<th>Verification</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>planck228.test.ibbt.be</td>
<td>AMD Athlon(tm) 64 Processor 3200+</td>
<td>2.60%</td>
<td>12.19%</td>
<td>&lt; 1%</td>
<td>0.0000556</td>
</tr>
<tr>
<td>planetlab2.cs.umass.edu</td>
<td>Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz</td>
<td>40.00%</td>
<td>31.18%</td>
<td>&lt; 1%</td>
<td>0.0000119</td>
</tr>
<tr>
<td>planetlab3.csee.usf.edu</td>
<td>Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz</td>
<td>58.70%</td>
<td>30.39%</td>
<td>&lt; 1%</td>
<td>0.0000123</td>
</tr>
<tr>
<td>planetlab4-dsl.cs.cornell.edu</td>
<td>Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz</td>
<td>63.80%</td>
<td>36.05%</td>
<td>&lt; 1%</td>
<td>0.0000200</td>
</tr>
<tr>
<td>myLaptop</td>
<td>Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz</td>
<td>76.00%</td>
<td>38.11%</td>
<td>&lt; 1%</td>
<td>0.0000133</td>
</tr>
<tr>
<td>station4.lab</td>
<td>Intel(R) Pentium(R) 4 CPU 3.06GHz</td>
<td>92.00%</td>
<td>31.72%</td>
<td>&lt; 1%</td>
<td>0.0000208</td>
</tr>
<tr>
<td>planetlab2.cs.uoi.gr</td>
<td>Intel(R) Pentium(R) 4 CPU 3.20GHz</td>
<td>30.70%</td>
<td>15.81%</td>
<td>&lt; 1%</td>
<td>0.0000200</td>
</tr>
<tr>
<td>planetlab-04.naist.jp</td>
<td>Intel(R) Pentium(R) 4 CPU 3.40GHz</td>
<td>2.70%</td>
<td>5.9</td>
<td>&lt; 1%</td>
<td>0.0000714</td>
</tr>
<tr>
<td>plab2-itec.uni-klu.ac.at</td>
<td>Intel(R) Pentium(R) D CPU 3.00GHz</td>
<td>26.20%</td>
<td>15.24%</td>
<td>&lt; 1%</td>
<td>0.0000263</td>
</tr>
<tr>
<td>pl2.unm.edu</td>
<td>Intel(R) Pentium(R) D CPU 3.20GHz</td>
<td>6.40%</td>
<td>7.48</td>
<td>&lt; 1%</td>
<td>0.00000221</td>
</tr>
<tr>
<td>planetlab2.ucsd.edu</td>
<td>Intel(R) Pentium(R) D CPU 3.40GHz</td>
<td>14.10%</td>
<td>18.22%</td>
<td>&lt; 1%</td>
<td>0.0000435</td>
</tr>
<tr>
<td>planet1.zib.de</td>
<td>Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz</td>
<td>66.30%</td>
<td>31.7</td>
<td>&lt; 1%</td>
<td>0.0000159</td>
</tr>
<tr>
<td>earth.cs.brown.edu</td>
<td>Intel(R) Xeon(R) CPU 3060 @ 2.40GHz</td>
<td>46.60%</td>
<td>28.01%</td>
<td>&lt; 1%</td>
<td>0.0000185</td>
</tr>
<tr>
<td>planetlab-02.naist.jp</td>
<td>Intel(R) Xeon(R) CPU 3075 @ 2.66GHz</td>
<td>19.70%</td>
<td>29.72%</td>
<td>&lt; 1%</td>
<td>0.0000208</td>
</tr>
<tr>
<td>planet-lab1.itba.edu.ar</td>
<td>Intel(R) Xeon(R) CPU X3220 @ 2.40GHz</td>
<td>73.30%</td>
<td>41.67%</td>
<td>&lt; 1%</td>
<td>0.0000137</td>
</tr>
<tr>
<td>aladdin.planetlab.extranet.uni-passau.de</td>
<td>Intel(R) Xeon(R) CPU E5420 @ 2.50GHz</td>
<td>63.80%</td>
<td>45.59%</td>
<td>&lt; 1%</td>
<td>0.0000109</td>
</tr>
<tr>
<td>iraph.lab1.iralab.uni-karlsruhe.de</td>
<td>Intel(R) Xeon(R) CPU X3220 @ 2.40GHz</td>
<td>73.30%</td>
<td>41.67%</td>
<td>&lt; 1%</td>
<td>0.0000137</td>
</tr>
</tbody>
</table>

**CPU id**: CPU identifier (/proc/cpuinfo)

**Idle**: CPU idle time right before the benchmark

**Mms**: Modular multiplications per milliseconds

**Verification**: provers verification phase effort compared to his benchmark effort

(this is also the maximal gain the prover can claim)

**Security**: Probability that the prover can cheat the verification phase (guess k')
Table 5.3: SecRaCP results, sorted by Mms

Sorted by CPU identifier and idle CPU time (CPU Speed: Pentium 4 < Pentium D < Core2 Duo < Xeon)

<table>
<thead>
<tr>
<th>Hostname</th>
<th>CPU id</th>
<th>Idle</th>
<th>Mms</th>
<th>Verification</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>planetlab-04.naist.jp</td>
<td>Intel(R) Pentium(R) 4 CPU 3.40GHz</td>
<td>2.70%</td>
<td>5.9</td>
<td>1%</td>
<td>0.0000714</td>
</tr>
<tr>
<td>pl2.unm.edu</td>
<td>Intel(R) Pentium(R) D CPU 3.20GHz</td>
<td>6.40%</td>
<td>7.48</td>
<td>1%</td>
<td>0.0000321</td>
</tr>
<tr>
<td>pland228.test.ibbt.be</td>
<td>AMD Athlon(tm) 64 Processor 3200+</td>
<td>2.60%</td>
<td>12.19</td>
<td>1%</td>
<td>0.0000556</td>
</tr>
<tr>
<td>plab2-itec.uni-klu.ac.at</td>
<td>Intel(R) Pentium(R) D CPU 3.00GHz</td>
<td>26.20%</td>
<td>15.24</td>
<td>1%</td>
<td>0.0000263</td>
</tr>
<tr>
<td>planetlab2.cs.uoi.gr</td>
<td>Intel(R) Pentium(R) 4 CPU 3.20GHz</td>
<td>30.70%</td>
<td>15.81</td>
<td>1%</td>
<td>0.0000200</td>
</tr>
<tr>
<td>planet1.zib.de</td>
<td>Intel(R) Pentium(R) D CPU 3.40GHz</td>
<td>14.10%</td>
<td>18.22</td>
<td>1%</td>
<td>0.0000435</td>
</tr>
<tr>
<td>earth.cs.brown.edu</td>
<td>Intel(R) Xeon(R) CPU 3060 @ 2.40GHz</td>
<td>46.60%</td>
<td>28.01</td>
<td>1%</td>
<td>0.0000185</td>
</tr>
<tr>
<td>planetlab2.ucsd.edu</td>
<td>Intel(R) Pentium(R) D CPU 3.20GHz</td>
<td>20.00%</td>
<td>29.35</td>
<td>1%</td>
<td>0.0000238</td>
</tr>
<tr>
<td>planetlab-02.naist.jp</td>
<td>Intel(R) Xeon(R) CPU 3075 @ 2.66GHz</td>
<td>19.70%</td>
<td>29.72</td>
<td>1%</td>
<td>0.0000208</td>
</tr>
<tr>
<td>planetlab3.csee.usf.edu</td>
<td>Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz</td>
<td>58.70%</td>
<td>30.39</td>
<td>1%</td>
<td>0.0000123</td>
</tr>
<tr>
<td>planetlab2.cs.umass.edu</td>
<td>Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz</td>
<td>40.00%</td>
<td>31.18</td>
<td>1%</td>
<td>0.0000119</td>
</tr>
<tr>
<td>planet-lab1.itba.edu.ar</td>
<td>Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz</td>
<td>66.30%</td>
<td>31.7</td>
<td>1%</td>
<td>0.0000159</td>
</tr>
<tr>
<td>station4.lab</td>
<td>Intel(R) Pentium(R) 4 CPU 3.06GHz</td>
<td>92.00%</td>
<td>31.72</td>
<td>1%</td>
<td>0.0000208</td>
</tr>
<tr>
<td>planetlab4-dsl.cs.cornell.edu</td>
<td>Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz</td>
<td>63.80%</td>
<td>36.05</td>
<td>1%</td>
<td>0.0000200</td>
</tr>
<tr>
<td>myLaptop</td>
<td>Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz</td>
<td>76.00%</td>
<td>38.11</td>
<td>1%</td>
<td>0.0000133</td>
</tr>
<tr>
<td>iraplab1.iralab.uni-karlsruhe.de</td>
<td>Intel(R) Xeon(R) CPU X3220 @ 2.40GHz</td>
<td>73.30%</td>
<td>41.67</td>
<td>1%</td>
<td>0.0000137</td>
</tr>
<tr>
<td>planetlab4.hiit.fi</td>
<td>Intel(R) Xeon(R) CPU E5542 @ 2.50GHz</td>
<td>63.80%</td>
<td>45.59</td>
<td>1%</td>
<td>0.0000109</td>
</tr>
<tr>
<td>aladdin.planetlab.extranet.uni-passau.de</td>
<td>Intel(R) Xeon(R) CPU E5542 @ 2.50GHz</td>
<td>87.70%</td>
<td>50.97</td>
<td>1%</td>
<td>0.0000118</td>
</tr>
</tbody>
</table>

CPU id: CPU identifier (/proc/cpuinfo)
Idle: CPU idle time right before the benchmark
Mms: Modular multiplications per milliseconds
Verification: provers verification phase effort compared to his benchmark effort (this is also the maximal gain the prover can claim)
Security: Probability that the prover can cheat the verification phase (guess k')
Table 5.4: Verification time vs. cheating probability vs. security level

(1024 bit modular system)

<table>
<thead>
<tr>
<th>Verification time</th>
<th>Host</th>
<th>Squaring rounds</th>
<th>B.time</th>
<th>V.time</th>
<th>Mms</th>
<th>Mms max</th>
<th>Gain max</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00%</td>
<td>myLaptop</td>
<td>6500000</td>
<td>154067</td>
<td>1093</td>
<td>42.190</td>
<td>42.491</td>
<td>0.7132%</td>
<td>0.0000154</td>
</tr>
<tr>
<td>0.10%</td>
<td>myLaptop</td>
<td>6500000</td>
<td>172174</td>
<td>74</td>
<td>37.753</td>
<td>37.769</td>
<td>0.0417%</td>
<td>0.0001538</td>
</tr>
<tr>
<td>0.01%</td>
<td>myLaptop</td>
<td>6500000</td>
<td>170611</td>
<td>55</td>
<td>38.099</td>
<td>38.111</td>
<td>0.0306%</td>
<td>0.0015385</td>
</tr>
<tr>
<td>0.001%</td>
<td>myLaptop</td>
<td>6500000</td>
<td>165034</td>
<td>29</td>
<td>39.386</td>
<td>39.393</td>
<td>0.0171%</td>
<td>0.0153846</td>
</tr>
</tbody>
</table>

Verification time: provers verification phase effort compared to his benchmark effort
B.time: Measured benchmark time
V.time: Measured verification time
Mms: Resulting modular multiplications per millisecond for computing the benchmark
Mms max: The maximal modular multiplications per millisecond value, a malicious prover can claim
Gain max: The maximal percentage a malicious prover can cheat the Mms value with the measured V.time
Security: Probability that the prover can cheat the verification phase (guess k’)


5.6.3 Interpretation

To be able to interpret the results, we have to know the behavior of the Planetlab nodes. What we access through SSH is a virtual server instance on the node. Next to our instance, several other instances are computing their stuff. We recognized that the idle CPU time (Linux top command) is more or less stable and showed the available computational power our instance has available for computing. But this is just the case in average. Short computations (peaks) can gain more CPU time. But through a run time of several minutes, the average CPU time is more or less the idle time before the start of the computation. This gives us the possibility to rank different nodes with the same CPU by their available computational power by comparing the idle time before the benchmark. This is important to know if our protocol can actually rank the nodes by their available computational power. The results gave us right. The table where we sorted for the CPU identifiers and then for our cp value, shows a direct correlation of the available computational power and the idle CPU time if the processor types are equal.

We took 120 seconds for the minimal run time of the benchmark and a factor of 100 for the ratio of benchmark time and maximal verification time. These values guaranty a good advantage in addition with a good security level. Of course they can be adjusted for different needs.

But what can we say, if the verification time is higher than one percent of the benchmark time? In this case it can be just a fluctuation of the available computational power. Because the verification time is much smaller, short fluctuation can influence the result more heavily. On the other side, it can also be a cheating attempt. The prover can try to need the 'available' verification time to continue the computation of the benchmark. Recall that this is possible because the verifier does not check for the correct result of the benchmark, but just if the provers knows the correct new exponent $k'$.

Remark: Someone would think that the best detection of a cheating approach would be, to compute the modular multiplications per seconds of the prover for the benchmark itself and compare it to the modular multiplications per seconds for the computing of $k$. But again because of the fluctuations in the available computation power, this is not possible. Additionally we have to do an extremely low level analysis of both algorithms (compute $v$ and $k'$) to know the exact correlations between them. Because we just know the exact number of multiplications the prover has to do, but not the actual run time to compute it. This because the multiplications of the computing $v$ are multiple squarings (which can be optimized in a way) and computing $k'$ is the additional multiplication of different numbers. And because these low level operations are done within microseconds, also programming language specific considerations must be done. To instantiate of copy a BigInteger Object or several Integer arrays, can lead to a heavy reduction of the modular multiplications per seconds performance.
Chapter 6

Conclusion

This thesis addresses the problem of efficiently and secure verifying computational power. We motivated, that verifying computational power capabilities is important and pushing trust towards the measured hosts is not acceptable, in most situations. Our analysis and implementation results reveal that a malicious host can abuse the operation of current benchmarks and trick a measuring entity into accepting false computational power capability. Our proposed secure ranking protocol, based on repeated modular squaring, has an optimized verification phase and our implementation results confirmed its practicability. The computation of the repeated modular squaring result can be done in $O(\log k \cdot M(n))$. The verification with the Euler totient in $O(\log k' \cdot M(n) + \frac{\log k}{\log n} \cdot M(n))$ and with our new approach in $O\left(\frac{\log k}{\log n} \cdot M(n)\right)$. The verifier has an advantage of factor $\log n$ compared to the prover, and we improved the existing verification method complexity about $\log k'$. We also demonstrated the security and the performance of our proposed benchmark both analytically and by implementation over, 16 randomly chosen, PlanetLab nodes. The implementation results depend strongly on the implementation code since every unnecessary code fragment within the squaring loop, increases the run time heavily. Therefore we used as much as possible, existing and optimized Java methods. Improved results are possible with hand made, even more optimized code. The optimal protocol parameterization, like number of digits, can change case-by-case. But our discussion about the parameterization supports this process and helps to avoid misconfigurations. However, we believe that we achieved a secure (ranking) protocol with comprehensive implementation results and that we considered all important aspects connected with this topic.
Bibliography


[14] Yuhsuke Ohtaki, Daisuke Takahashi, Taisuke Boku and Mitsuhisa Sato ”Parallel Implementation of Strassen’s Matrix Multiplication Algorithm for Heterogeneous Clusters”, In Parallel and Distributed Processing Symposium, 2004


[21] Papi Framework, icl.cs.utk.edu/papi


[27] Performance ranking of actual CPUs, www.cpubenchmark.net/common_cpus.html

[28] Performance ranking of actual supercomputers, icl.cs.utk.edu/hpcc/hpcc_results.cgi

86
[29] Java Agent Development Framework, jade.tilab.com
Appendix

Source code and raw data

This thesis includes a compact disc with the source code of all implementations we did during progress of this work. Additionally we include the raw data output of our implementation, which are the foundations of our presented results.