Christoph Renner

Crypto-Breaker Reloaded

Semester Thesis
September 2007 to January 2008

Advisors:
Prof. Dr. Roger Wattenhofer
Michael Kuhn
Stefan Schmid

Abstract

A large number of CPUs connected to the Internet are idle almost all the time. These CPUs can help solving computationally difficult problems like the discrete logarithm problem. The goal of this thesis is to improve the work of Christoph Schwank towards a volunteer computing application which participates at the Certicom ECC Challenge. We decided to switch towards Koblitz curves since the ECC2K-130 seems to be the next simplest challenge to solve. This switch and the poor performance of Crypto++ [7] which was used before resulted in a complete rewrite of the client application.
Our server software consists of the BOINC [6] framework enhanced with distributed checking which prevent selfish behavior of participants with almost no additional resources used on client side. As client software we use the original BOINC client which runs our client application.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 The Task . . . 3
- 1.3 Results . . . 4
- 1.4 Overview . . . 4

2 Design **5**

3 Implementation **7**

4 Pollard's rho algorithm for Koblitz curves **9**
- 4.1 Distributed version . . . 9
- 4.2 Optimization . . . 9
- 4.3 Checking function . . . 10
- 4.4 ECC library performance . . . 11

5 Conclusion and Future Work **13**

A BOINC server installation **14**
- A.1 Requirements . . . 14
- A.2 Compiling the server programs . . . 14
- A.3 Creating a project . . . 15
- A.4 Modifying config.xml . . . 15
- A.5 Changing file permissions/ownership . . . 15
- A.6 Adding the application . . . 15
- A.7 Setting up the daemons . . . 15

Chapter 1

Introduction

1.1 Motivation

Volunteer computing uses resources which are provided by volunteers. One of the most popular volunteer computing projets is SETI@Home [5]. It uses the computers of volunteers distributed over the whole planet to analyze radio signals from space. To motivate people or organizations to donate computing power, a user gains credit proportional to the work done by his computers. On the internet one can see the user's ranking. Selfish users might try to gain credit for work they have not done to get a bether position in the ranking without donating more computing power. To avoid this selfish behavior SETI@Home uses redundancy. The whole work is divided into work units which can be computed for their own. SETI@Home then assigns each work unit to multiple users and then compares the results and calculates the credit which is then granted to all these users. The drawback of this approach is that this redundancy uses a lot of computing power. For example if each workunit is computed three times, only 33% of the donated resources are used for computing and the rest is used to check whether the computation is correct and to determine how much credit is granted for this work unit.The approach pursued in this thesis does not use redundancy to check whether a result is correct. Instead it uses checking work units which use a special function to determine whether a result is correct. The resources needed to compute such a checking work unit can be much less than those needed to compute a normal work unit. This asymmetry is essential for our approach to save resources for other work units.

1.2 The Task

The main goal of this thesis is to compete at the ECC Challenge [1]. To solve the ECC Challenge one must calculate a discrete logarithm on an elliptic curve. We decided to attack ECC2K-130 because we expect that this is the simplest challenge which has not yet been solved. To actually get a chance to solve this discrete logarithm the following tasks have to be performed:

- Port the client application to linux. That enables us to run larger tests on the infrastructure which is provided by the ETH and increases the number of potential participants for our project.
- Modify the client application to use Koblitz curves instead of curves over $GF(p)$.
- Implement optimizations which exist for Koblitz curves to reduce the number of expected iteration steps until one finds the solution.
- Speed up the client application by optimizing its C++ code.
- Implement a new checking algorithm since the one used by Christoph Schwank has a weakness which could be exploited by unhonest users.

1.3 Results

The result of this thesis is a modified version of the BOINC server which performs distributed checking for the distributed version of Pollard's rho algorithm for discrete logarithms. The checking needs almost no resources on client side and it should be easy to adapt this server to work with other algorithms. The source code on CD also includes the client application which does the actual work to solve the discrete logarithm problem for Koblitz curves.

1.4 Overview

Chapter 2 describes the general idea of the used checking algorithm whereas chapter 3 describes its implementation in our version of the BOINC framework. Chapter 4 gives an overview about the distributed version of Pollard's rho algorithm we use. And Appendix A is a short tutorial on how to setup a BOINC server with distributed checking capabilities.

Chapter 2

Design

The basic concept of our checking algorithm is to exploit that often it is easier to verify the correctness of a solution than to calculate the solution itself. A client can provide A as a fingerprint of its calculation which can be used by other clients to test the correctness of its solution. For the algorithm to be efficient several things have to be considered: First, the overhead of generating A while computing the solution should be small to not slow down the computation. Second, there should be a huge asymmetry of resources needed to verify a solution with help of the fingerprint to the resources needed to calculate the solution itself. And of course the generation of a solution and A which would validate the solution without actually calculating the solution must be at least as hard as actually calculating the solution and A. Section 4.3 shows that for the distributed Pollard's rho algorithm such a checking algorithm exists. The detailed checking algorithm is described in [8].

Cheaters might be able to create results and checks which look correct but can be created much faster than if they would really be computed. To avoid that such cheaters can do allmost all the checking and therefore check their own results. Checking units are distributed uniformly among all active clients. [1] This is only possible if checking is much faster than computation; Otherwise fast clients could create more results than the slower clients could check. A group of malicious users could accuse a honest user of cheating if they knew that they were doing all the checking for one result. They could then all together mark that result as wrong because no one else would do any checks on this work unit. To prevent this type of attack we assign checks one by one. This means we assign one checking unit, wait until we get the result back and then assign the next checking unit. Therefore one cannot know who will check the result afterwards one has submitted his check.

Since the main goal of volunteer computing is to calculate the solution and not the checking one wants to minimize the number of checks per workunit. But it must be assured that selfish behavior is not worthwhile. This means that the fraction of false positives [2] and false negatives [3] should be upper bounded by some θ_N for false negatives and some θ_P for false positives. Let α denote the number of checks in favor of the result's correctness (including the result itself) and β the number of checks which state that the result is wrong. We have the following upper bound for the probability of a false positive:

$$Pr[P|\alpha,\beta] \leq \left(\begin{array}{c} \alpha+\beta \\ \alpha \end{array} \right) p_P^{\beta}(1-p_P)^{\alpha}$$

where p_P it an upper bound for the fraction of cheaters involved to check this result. p_P can be written as:

$$p_P = \min(\tfrac{\beta}{\alpha+\beta}, p)$$

where p denotes the upper bound for the fraction of cheaters. The expressions for the false negatives are as follows:

[1] Clients are considered active if they had contacted the server withing the last N hours

[2] False positive denotes checks which indicates that the result is wrong although it was correct

[3] Wrong result which passed the checking as correct

$$Pr[N|\alpha,\beta] \leq \left(\begin{array}{c} \alpha+\beta \\ \alpha \end{array} \right) p_N^{\alpha}(1-p_N)^{\beta}$$
$$p_N = \min(\tfrac{\alpha}{\alpha+\beta}, p)$$

Let's now look at some pseudo code which illustrates the checking algorithm.

Algorithm 1 Distributed checking

```
ok = false
newcheck = true
α = 1
β = 0
users_ok = { user who reported solution }
users_failed = {}
repeat
  choose one user among the active ones
  assign check to that user
  wait for user to report result of check
  if check is ok then
    α = α + 1
    users_ok = users_ok + { user }
  else
    β = β + 1
    users_failed = users_failed + { user }
  end if
  if α < β AND Pr[P|α,β] < θ_P then
    newcheck = false
  else if α > β AND Pr[N|α,β] < θ_N then
    newcheck = false
    ok = true
  end if
until newcheck = false
if ok then
  punish users in users_failed
  give credits to users in users_ok
else
  punish users in users_ok
  drop reported solution
  give credits to users in users_failed
end if
```

Chapter 3

Implementation

The BOINC framework consists of several components listed in Table 3.1. Because the checking algorithm assigns checking units to a specific user and this feature is not contained in the original BOINC server it must be implemented into the BOINC framework. Fortunately Christoph Schwank has already done this before. So I knew exactly where I had to insert my code.

The changes on the database structure were straight forward. Each result must have an additional field for the α and β. To keep track of the checking state a new table called *checks* was created. It has the following structure:

integer	*id*	unique integer which identifies the check
integer	*resultid*	the result which this check should test for correctness
integer	*userid*	user to which this check is assigned
integer	*checkstate*	current state of this check
integer	*creationtime*	Unix time when this check was created

When a new result is reported a new record is inserted into this table with *resultid* set to the according *id* in the *results* table, *creationtime* set to the current time and *checkstate* set to CHECKS_UNASSIGNED which means that this check has not yet been assigned to a user. A daemon named *checkingcontroller* periodically checks for unassigned checks. For each of them it randomly chooses one user, sets the corresponding *userid* and changes the *checkstate* to CHECKS_ASSIGNED. If a client asks the *scheduler* for more work the *scheduler* checks the *checks* table for assigned checks which match that *userid*. If that set is not empty the *scheduler* creates a new checking unit for that user which contains the needed information to check this results and sets the *checkstate* to CHECKS_SENT. These work units have a very early deadline so that the client performs checking with high priority. When this checking unit gets reported the *checkingcontroller* changes the *checkstate* of the according check to

Database	BOINC uses a MySQL database to stores state information
Scheduler	A cgi program which is contacted by the client.
Feeder	Reads jobs from database and stores it in memory which is used by the scheduler. This is more efficient than to create a new database connection for every scheduler request.
Transitioner	Takes care of the state transition of a work unit. Creates result for new workunits and timed-out results. Marks workunit for validation, assimilation and deletion.
Validator	Compares different results for one workunit. Chooses canonical result and grants credit.
Assimilator	Processes canonical results. Extracts important data and stores it somewere else.
File deleter	Once a work unit is done and marked for deletion the file deleter deletes all input and result files which belong to that work unit.

Table 3.1: BOINC components

CHECKS_OK or CHECKS_FAILED depending on whether the check was successfull or not. The *checkingcontroller* also periodically checks for completed checks, and increments the α and β according to the outcome of the check. If another check is necessary it also creates a new entry with *checkstate* set to CHECKS_UNASSIGNED and it starts all over again until there are enough checks for that result. If all the checking is done the *checkingcontroller* marks the result as valid if $\alpha > \beta$ or deletes the result it $\beta > \alpha$. At the end it removes all entries from the *checks* table which are not needed anymore.

In the original design of BOINC the *validator* compares the different results for one work unit and then chooses a canonical representation for these results. Since in our design only one result per work unit is generated the *validator* has somehow become redundant. But to avoid making even bigger changes on the framework our *validator* still processes each work unit but it does only some simple format checks on the output files to ensure that the can be processed easily by the *assimilator*. Our *validator* does not grant any credit since this is done after the checking to ensure that one gets no credit for a result which does not pass the checking. The assimilator parses the output files which are received from the client and makes the needed changes in the database. The *filedeleter* and the *transitioner* needed no changes. The *feeder*, however needed one small change: it must not feed the shared memory with work units which represent checking units because they are assigned to a specific user and the work units from that shared memory are not.

Chapter 4

Pollard's rho algorithm for Koblitz curves

Given a generator P and a point Q on a group G, the Pollard's rho algorithm for discrete logarithms tries to compute l such that $Q = lP$. To achieve this, the algorithm tries to find $X_1 = a_1P + b_1Q$ and $X_2 = a_2P + b_2Q$ such that $X_1 = X_2$ and $(a_1, b_1) \neq (a_2, b_2) \pmod n$ where n is the order of G. Once such a collision is found, the computation of l is straight forward. One only needs to solve the equation $a_1 - a_2 = l(b_2 - b_1) \pmod n$. This chapter will give a short overview how we try to find such collisions for Koblitz curves.

4.1 Distributed version

The Pollard's rho algorithm uses the Floyd's cycle-finding algorithm to find the cycle in a sequence $x_0, ..., x_n$ where $x_{i+1} = f(x_i) \forall i < 0$. The function $f : G \to G$ originally proposed by Pollard look like this:

$$f(X) = \begin{cases} P + X & X \in G_0 \\ 2X & X \in G_1 \\ Q + X & X \in G_2 \end{cases} \tag{4.1}$$

where $G = G_0 \cup G_1 \cup G_2$. The order of G_0, G_1, G_2 should be approximately the same. However this algorithm cannot easily be distributed over several processors. Instead we use the following approach: each processor starts at a random point x_0 and follows the trail until it reaches a distinguished point. A distinguished point is a point with some easy to test property. Once a distinguished point D is reached, it is stored together with the corresponding values a, b such that $aP + bQ = D$. As soon as a point already exists a collision is found.

The expected number of group operations until a collision occurs[1] is given by $E[T] = \sqrt{\pi n/2}/m$ where m is the number of processors used. But since collisions are not detected directly because the iteration continues until a distinguished point is reached it takes additional $1/P[D]$ group operations. $P[D]$ denotes the probability that one randomly chosen point is a distinguished point. So we get to $E[T] = \sqrt{\pi n/2}/m + 1/P[D]$.

4.2 Optimization

This section briefly describes two optimizations to speed up Pollard's rho algorithm for the discrete logarithm problem. For further details see [4] and [3].

For any given point $P = (x, y)$ on an elliptic curve it is trivial to compute $-P$. In case of a curve over $GF(2^m)$ this is simply $-P = (x, x + y)$. So for each point calculated with our iteration function we get one for free. If we now chose allways the point from P and $-P$ which has the

[1] that is the probability of a collision reached 0.5

smaller integer representation of his y component we reduce the number of point over which we perform the collision search by factor 2. This means that the number of expected iterations will decrease by factor $\sqrt{2}$. Of course the equality $X = aP + bQ$ must still hold so if we choose $-P$ we have to adjust a and b but this is also trivial: $(a, b) \rightarrow (-a, -b)$.
One problem which arises form this optimization is the occurrence of trivial two-cycles. Assume that both X_i and X_{i+1} belong to the same group G_j, both X_{i+1} and X_{i+2} are the negative of the resulting point after applying the iteration function f and that f for that group has the form $f(X) = X + cP + dQ$. Now we have $X_{i+1} = -(X_i + cP + dQ)$ and $X_{i+2} = -(-(X_i + cP + dQ) + cP + dQ)) = X_i$. By using an iteration function f like

$$f(X) = \begin{cases} X + c_0P + d_0Q & X \in G_0 \\ \vdots & \vdots \\ X + c_{19}P + d_{19}Q & X \in G_{19} \end{cases} \tag{4.2}$$

the occurrence of trivial two-cycles can be reduced because this iteration function divides the group into 20 partitions. Therefore the probability that X_i is in G_k is reduced from $\frac{1}{3}$ to $\frac{1}{20}$. This implies that the occurrence of trivial two-cycles is reduced. However it is not reduced enough. The next step of reducing trivial two-cycles is to use a look-ahead technique which does the following: define a function $g : G \rightarrow G'$ $g(X) = \pm f(X)$ where G' are all points in $P \in G$ with $int(P.y) < int((-P).y)$ where $int(x)$ represents the integer interpretation of x. Now calculate $T = g(X_i)$ if T is in the same group G_j as X_i than calculate $T = g(X_i)$ as if X_i was in the group G_{j+1} ($j + 1$ modulo 20) if T is still in G_j try the other eighteen groups. And then use $X_{i+1} = T$ The idea is to reduce the probability that the two points X_i and X_{i+1} belong to the same group. Experiments have shown that the probability for trivial two-cycles is low enough for practical pourposes if one somehow detects[2] them and does not continue to iterate forever.

As shown by Gallant, Lambert and Vanstone [3] the number of points we use to search for collisions can even further be reduced for Koblitz curves. Consider now a Koblitz curve over $GF(2^m)$. For every point P on that curve $m-1$ other points can easily be computed by just $m-1$ times squaring their x and y coordinates. This operation can be implemented very efficiently if using a normal basis representation, it is just a cyclic shift of the binary representation of the field element. [3] By doing so we get the following m points:

$$\begin{gathered} P_1 = (P.x^2, P.y^2) = \lambda P \\ P_2 = (P.x^4, P.y^4) = \lambda^2 P \\ \vdots \\ P_m = (P.x^{2m}, P.y^{2m}) = \lambda^m P = P \end{gathered}$$

because each of these points has also a negative version we can actually reduce the number of points over which we perform the collision search by a factor of $2m$ which reduces the number of expected group operations by $\sqrt{2m}$.
One can choose the distinct representation from the group formed by those $2m$ points by first choosing the point from $\lambda^i P$ which has the smallest x coordinate when interpreted as integer. And then the one from $\pm\lambda^i P$ which has the smallest y coordinate when interpreted as integer. Of course the equation $X = aP + bQ$ must still hold. But this is again quite simple: one must just replace (a, b) with $(\pm\lambda^{2i}a, \pm\lambda^{2i}b)$. The λ^{2i} can be precomputed and λ is one of the roots of $X^2 + X + 2 = 0 \pmod p$ where p is the order of the points in G.

4.3 Checking function

As described in Chapter 2 for each result a client needs to supply proof that he has really done the requested work. The other clients must then be able to check the correctness of the result with this proof. A checking function could works as follows:

- the client follows the iteration function until he finds a distinguished point X_n

[2] e.g. define an upperbound for the number of iterations and abort after reaching this bound
[3] Unfortunately there was no time left to implement the client using normal basis representation

- it reports (X_n, a_n, b_n) and $(X_{n-50}, a_{n-50}, b_{n-50})$
- a client which should do checking gets (X_n, a_n, b_n) and $(X_{n-50}, a_{n-50}, b_{n-50})$
- it starts at $(X_{n-50}, a_{n-50}, b_{n-50})$ and does 50 iteration steps
- the result passed the check if the client found (X_n, a_n, b_n)

To fulfill the requirements from Chapter 2 one needs to show three things:

- generating the proof must not be to difficult.
- generating a proof to a solution without calculation the solution must be as hard as calculation the solution and the proof.
- there must exist a large asymmetry between the resources needed to calculate a solution and the resources needed for checking

The first point is quite simple: If X_n is a distinguished point it is easy to always store the 50 last points and also report X_{n-50} as proof for the correctness.
If you just create a point which is a distinguished point it is hard and very often even impossible to calculate X_{n-50}. Simulation has shown that the probability that such a X_{n-50} exists is already below 10%. This means that the expected number of distinguished points one needs to create is circa 10. Now it can be argued that there is no personal gain for a user to generate X_{n-50} and the according a and b unless he can create the distinguished point at least 10 times faster than to actually search a distinguished point by our algorithm. But to our knownledge generating the triple (a, b, X) can only be done by choosing a and b at random (one might also set either a or b to some fixed value), calculate $X = aP + bQ$ and then chose $X' = \pm\lambda^i X$ according to the schema explained above. The expected number of tries until one finds a X' which is a distinguished point is $1/P[D]$ and this is exactly the expected number of iteration steps until one finds a distinguished point. Therefore one can argue that generating (a', b', X') is not much easier than really doing the work. If one now considers that one has to create ten such triples and in addition also needs to compute X'_{n-50} doing the actual work seems far more worthwile.
The asymmetry clearly exists because calculating 50 iteration steps is almost nothing when compared to maybe 10^6 steps which are required to find a distinguished point from a random start point.

4.4 ECC library performance

Christoph Schwank decided to use the Crypto++ library to implement the client. For our client application the speed of the group operation on elliptic curves has a really large impact on the number of iterations per second we can perform. I decided to evaluate other libraries and then compare them. The only alternative I found is OpenSSL which also contains an ECC library. To give Crypto++ a fair chance I first optimized the iteration function as implemented by Christoph Schwank. The most important optimizations were:

- remove all output to stdout and stderr in the iteration function since this is only needed for debugging. This made the program 1.5 times faster.
- since the program seemed to spend a lot of time calculating hashes and converting Integers to strings, I removed the hash functions which were used to decide if a point is a distinguished point and to which subgroup of G the point belongs. Instead I made this decisions only based on the integer representation of the point's x coordinate. This improvement increased the iterations per second by a factor of 2.6

Then I implemented the same functionality using the OpenSSL library. The time needed for a given number of iterationsteps is shown in Figure 4.1. One can see that the implementation using OpenSSL is almost 10 times faster. Based on this result I implemented the client application using OpenSSL instead of Crypto++. If one assumes that the average computer can do ten thousend iterations per second. What is then the estimated runtime until one finds a solution? The order of the group for ECC2K-130 is $n = 6.8 \cdot 10^{38}$. This brings us to the expected

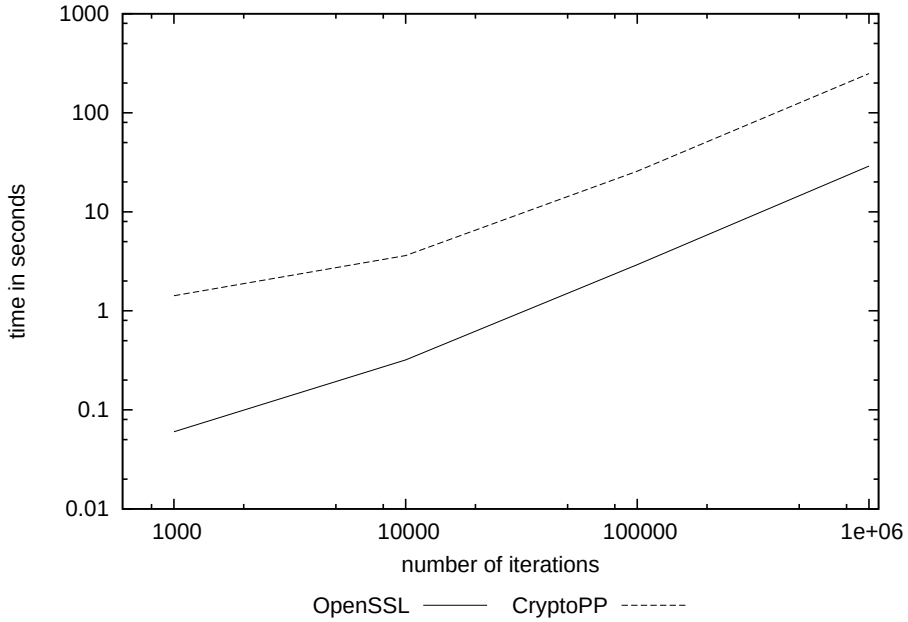

Figure 4.1: Comparison of OpenSSL and Crypto++

number of iterations until we find a collision: $\sqrt{n\pi/2}/\sqrt{262} \approx 2 \cdot 10^{18}$. On a single computer with ten thousend iterations per second this takes $6 \cdot 10^6$ Years. If we now have hundred thousend computers which calculate 24 hours a day for our project we still need 60 years. If one assumes that the computing power doubles all two years one still needs circa 18 years. Besides the running time until we find a collision there might be another problem: the memory which is used to store the distinguished points. If we want one workunit to take 24 hours we need to store $2 \cdot 10^{18}/(10000 \cdot 3600 \cdot 24) = 2 \cdot 10^9$ points. Even if we need 100 Bytes per distinguished point this should not be a problem.

Chapter 5

Conclusion and Future Work

This thesis has described an implementation of a distributed checking algorithm. Our modified BOINC server with the client application could be used to participate at the Certicom ECC Challenge.

Our runtime estimations show that optimizing the client for speed must have top priority. One important step towards a faster client would be to use the normal basis representation. This would especially speed up the squaring of $X.x$ and $X.y$. Even more optimization should be possible because for allmost all group operations one operand is known at compile time and can therefore be precomputed. One might be able to optimize the group operation for exactly this operand. Since one probably wants a new property for the distinguished points when using normal basis representation, one has to throw away all old distinguished points. This means that all the work already done is lost. Therefore it would make sense to wait until the client uses normal basis representation and then start attacking the ECC Challenge.

Appendix A

BOINC server installation

This chapter explains how to setup a server to compete at the ECC challenge [1] using a modified BOINC server which implements the distributed checking algorithm described in this report.

A.1 Requirements

To install a BOINC server you need a UNIX like operating system. The code for this thesis was developed and tested under Ubuntu 7.04 and 7.10. Furthermore you will need the GNU C++ compiler, GNU make and some other tools and libraries. But at the moment you don't have to worry about them because `_autosetup` and `configure` will complain if any of those are missing.

BOINC uses MySQL to store its data and Apache to communicate with the clients. So one has to setup MySQL and Apache. I assume this has already been done and that the user has full access to the database on localhost with username `boincadm` and password `myPassword`.

Since in our setup the cgi program will create files in the download directory the filedeleter will not be able to delete these files. To allow him to delete these files owned by the user/group of apache we need to add the user which runs the daemons (in our case boincadm) to Apache's group (in our case www-data).

A.2 Compiling the server programs

The source code can be found on the CD in the boinc directory. The first step is to run automake and friends. This is done by typing the following command: (the $ must not be typed)

```
$ ./_autosetup
```

Then one needs to call the configure script which checks whether all the needed libraries are installed on your system. This command will output a lot of text. Don't worry about a warning which indicates that openGL, glut and GLU are missing since we don't need the graphical part of the BOINC API.

```
$ ./configure --disable-client
```

Now we have to compile all the needed parts of BOINC. This is done by typing

```
$ make
```

A.3 Creating a project

The following command will copy all needed files to `$HOME/projects/hello` and create the database. Replace URL with the hostname of your server.

```
$ cd tools
$ ./make_project -v --delete_prev_inst --drop_db_first
    --db_user boincadm --db_passwd myPassword
    --url_base http://URL/ hello Hello@Home
```

Follow step one and two of the instructions at the end of this output. The project has been installed into `$HOME/projects/hello`. From now on it is assumed that this is your working directory.

```
$ cd ~/projects/hello
```

A.4 Modifying config.xml

In config.xml you need to set `disable_account_creation` to 0 and add `<profile_screening/>` to the config element.

A.5 Changing file permissions/ownership

```
$ chgrp www-data download
$ chmod g+w download
```

A.6 Adding the application

In project.xml you will need to modify the `<app>` element that it looks like this

```
<app>
  <name>hello</name>
 <user_friendly_name>Hello Project</user_friendly_name>
</app>
```

Then we need to copy our client executable to a location where BOINC can find it:

```
$ mkdir -p apps/hello/hello_1.00_i686-pc-linux-gnu
$ cp /path_to_hello/hello_1.00_i686-pc-linux-gnu
    apps/hello/hello_1.00_i686-pc-linux-gnu/
```

Now we can add the application to the BOINC server

```
$ ./bin/xadd
$ ./bin/update_versions
```

A.7 Setting up the daemons

At first copy the template files which are needed to create work from the templates directory on the CD to `$HOME/projects/hello/templates/`. In the `<daemons>` section in config.xml replace feeder with distributed_checking_feeder and add the following elements at the end of the `<daemons>` element.

```
<daemon>
  <cmd>
    hello_validator_with_checking -d 3 -app hello
  </cmd>
</daemon>
<daemon>
  <cmd>
    hello_assimilator_with_checking -d 3 -app hello
  </cmd>
</daemon>
<daemon>
  <cmd>
    hello_make_work -appname hello -d 3 -sleep_time 30
      -nr_of_bits_p 131 -wu_name wu_name_template
      -wu_template templates/world_wu_mod.xml
      -result_template templates/hello_re_mod.xml
      wu_filename_template
  </cmd>
</daemon>
<daemon>
  <cmd>
    collision_finder -d 2 -number_of_results_to_check 1000
  </cmd>
</daemon>
<daemon>
  <cmd>
    checking_controller -d 2 -appname hello
  </cmd>
</daemon>
```

Then you can start the daemons with

```
$ ./bin/start
```

Bibliography

[1] The Certicom ECC Challenge,
http://www.certicom.com/index.php?action=ecc,ecc_challenge,
December 2007

[2] OpenSSL: The Open Source toolkit for SSL/TLS,
http://www.openssl.org/,
December 2007

[3] R. Gallant, R. Lambert and S. Vanstone, “Improving the Parallelized Pollard Lambda Search on Binary Anamolous Curves”, Research Report No. CORR98-15, Department of Combinatorics and Optimization, University of Waterloo, 1998.

[4] M. J. Wiener and R. J. Zuccherato, “Faster Attacks on Elliptic Curve Cryptosystems”, IEEE P1363: Research Contributions, 1999.

[5] SETI@Home: A scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence
http://setiathome.berkeley.edu/,
January 2008

[6] BOINC: Berkeley Open Infrastructure for Network Computing
http://boinc.berkeley.edu/,
January 2008

[7] Crypto++: A free C++ class library of cryptographic schemes.
http://www.cryptopp.com/,
January 2008

[8] Michael Kuhn and Stefan Schmid and Roger Wattenhofer, Distributed Asymmetric Verification in Computational Grids, 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Miami, Florida, USA
April 2008