Final report: Non-Interactive Zero Knowledge Protocol for Verifiable Computing
Part A – abstract description of the project:

In this project we implemented a zero knowledge protocol for verifiable computing. The protocol is based on the paper "Short Pairing-Based Non-interactive Zero-Knowledge Arguments" by Jens Groth, ASIACRYPT 2010: 321-340.

The project provides a way for the prover to produce a non-interactive proof that convinces the verifier that the circuit is satisfiable if and only if it is, with high probability. Perhaps the most probable use of such protocols may lie in replacing the misplaced trust in auditing firms, stock markets, governments, and other "trustworthy institutions" with protocols to enforce rules and insist that the various institutions prove that they are blameless via techniques such as these.

The paper is about a non-interactive protocol, involving a prover and a verifier, both of which are given [the same] Boolean circuit. We consider circuits that have only a single output wire and multiple input wires. A satisfying assignment to the circuit is an assignment to the input wires that sets the output wire to one.

Our implementation includes three major parts:

1. Generation of a Common Reference String (CRS),
2. The Prover protocol
3. The Verifier protocol

We also implemented methods for testing the protocol, on random circuits.

The Groth protocol has several components:

a. Build a Common Reference String (CRS). The CRS has The CRS needs to be accessible to both prover and verifier. In our basic implementation it is kept in a file. The CRS must be at least the size of \(((2 * n)^2 + 3 * (n * 2) - 2) * 5\), where n is the number of gates we have in our circuit.
b. Produce a representation of the circuit. We represent the circuit in a format that is easily readable by both prover and verifier. The input to the prover includes a satisfying assignment.
c. The prover reads the circuit and the CRS and produces a proof for the verifier.
d. We give two versions of the verifier:
a. Single verifier: the verifier runs as a single process, goes through the files accessible to it (proof, CRS and serialized circuit) and verifies the proof.

b. Multiple verifiers: Verification consists of two separate processes that work together: preprocessing the circuit, which does not depend on the satisfying assignment, and [very fast, constant time] verification of the proof using the preprocessed circuit.

The protocol has perfect completeness, therefore the verifier accepts every true proof (based on a satisfying assignment), to the given circuit. The verifier will not accept a proof for a circuit that cannot be satisfied (except with very small probability). Since the protocol is perfect zero-knowledge, we can be assured that implementing the protocol as specified will not leak information. The only information the verifier learns is whether the prover holds a satisfying assignment.

In general, the proof is based on the prover showing that the assignment to the circuit that he holds is consistent. That is, that all the gates in the circuit (for simplicity of the protocol, only NAND gates are used), that all wires are either zero of one, and that all copies of the output of a gate have the same value.
Part B: Background & Motivation

The ability to prove that a Boolean circuit is satisfiable allows one to perform arbitrary secure multiparty computation. In particular, institutions such as governments, banking, the stock market, auctions, and arbitrary commercial activity, have always required the assumption that "respectable parties involved" are playing according to the rules. Or, more correctly, that some parties are sufficiently trustworthy so that one may assume that they will act according to well established principles.

Recent examples, involving trillions of dollars, show that such institutions cannot be trusted. A long list of major accounting scandals can be seen in http://en.wikipedia.org/wiki/Accounting_scandals. Such events have lead to the breakup of 'the big 5', the world's 5 largest accounting firms. Even one such event (Envron, audited by Arthur Anderson, one of the worlds largest auding firms) caused damages in the hundreds of millions of dollars.

It is rather easy to make a financial system, based on digital signatures and public key encryption where the auditor produces a zero knowledge proof that a circuit that decrypts, verifies signatures, and computes the sum of income minus expenses, has a final value of, for example, ten million dollars. This circuit will also include access to a database of such public keys, and prove that the access was done correctly and that the database has not been corrupted.

But this is but one example of a major financial and economic problem that can be addressed using efficient proofs of circuit satisfiability. The recent fraud involving the Libor (London interbank offered rate) has resulted in damages in many billions of dollars, see http://en.wikipedia.org/wiki/Libor_scandal. Again, it would be trivial to design a system based using efficient ZK proofs that would make it impossible to manipulate such a quantity.

More to the point, the list is endless, whatever human activity may be, and unless one assumes that humans are indeed different than a machine, the process that humans follow (or should follow) can be described as a Boolean circuit. Using crypto tools, the inputs to the circuit can themselves be verified
as part of the circuit. Thus, the use of ZK proofs of circuit satisfiability may reintroduce trust into systems where trust has been clearly misplaced.
Part C - Detailed explanation of the project:

This project implements Groth's protocol for proving and verifying satisfiability of a circuit, without revealing anything else. As a simple example of an application this protocol could be used to prove one has a password to an electronic lock without revealing the password itself.

One of the advantages of the protocol is that it is non-interactive and very fast to verify.

Generating the Common Reference String (CRS):

The CRS is a data base of elements, in a symmetrical pairing group who follow certain rules. It need generated only once, and can be reused for different proofs for different circuits. The CRS should be itself generated by a trusted third party.

The CRS can be used for any number of proofs, for circuits of similar size or less. It is computed in quadric time, and takes quadratic space. In our implementation, the CRS is a class. Both prover and verifier use random access to elements of the common reference string. Therefore, in principle, it could be stored on a server on the internet and used by many users simultaneously (given secure access to this server).

Circuit Representation:

For an external circuit and satisfying assignment:

The format of the circuit description input file is attached as an appendix to this report. The circuit description input file must also include a satisfying assignment to the circuit.

The circuit description input file is processed and two different files produced, one for the prover (describing the circuit and the satisfying assignment) and one for the verifier (describing only the circuit).

If the circuit is randomly generated:

For testing purposes, we seek to generate random satisfiable circuits that are far from being tautologies.
The random circuit generation process gets an upper bound on the size of the circuit you want (total number of NAND gates). The circuit constructed will be nearly always of a size bigger than 85% of the size given. Also, we verify that randomly given values to the input wires do not satisfy the circuit.

Implicitly, our generation procedure describes a distribution over satisfiable circuits. This distribution has the property that every possible satisfiable circuit is in the support, that the number of NAND gates is concentrated around our preset goal, and that the circuit generated is far from begin a tautology. We remark that minor changes to our procedure can generate other distributions.

Prover:

It is straightforward to divide the construction of the prover protocol of Gens to three different steps since it involves 'knowledge commitment', 'product argument' and 'permutation argument' in its computation.

(A general) Knowledge commitment is a commitment where the prover cannot commit without knowing the underlying values. This commitment is divided to two parts:

1. A knowledge commitment – basically it is a module which multiplies elements from the group in the CRS, the elements depend on committed values and one random value. It uses the CRS file. For the input: 
   \( (r, a_1, \ldots, a_n) \)
   \[
   c = g^r \prod_{i \in [n]} g_i^{a_i}, \tilde{c} = \tilde{g}^r \prod_{i \in [n]} \tilde{g}_i^{a_i}
   \]
   Verification is to test if 
   \( e(\tilde{c}, g) = e(c, \tilde{g}) \)
   \( (e, g, \tilde{g} \text{ from the CRS}) \)

2. A restricted commitment - like the knowledge commitment, computes some product. Has a subset of \([q]\) \((n^2 + 3n - 2, n \text{ is circuit size})\) as input, and computes the suitable product.
   \( input: S \subset [q], (r, a_1, \ldots, a_n) \)
   \( output: \pi = h^r \prod_{i \in S} h_i^{a_i} \)
   (in practice used with some \((c, \tilde{c})\) knowledge commitment, \(h\) from CRS)

The function 'restricted_commitment_generator' in the file rka.cpp implements these commitments, this file is in the folder 'prover'. 
A product argument is used to prove that two vector's entry wise product equals a third vector. We us it to prove all wire values are 1 or 0, and that if A and B are inputs of a NAND, and U is the output, A*B = 1-U.

The argument for the input:

\[ r, a_1, ..., a_n \]
\[ s, b_1, ..., b_n \]
\[ t, u_1, ..., u_n \]

(proving \( \forall i \in [n]: u_i = a_i b_i \))

\[ \text{in } c = g^r \prod_{i \in [n]} g_i^{a_i}, d = g^s \prod_{j \in [n]} g_j^{b_j}, v = g^t \prod_{i \in [n]} g_i^{u_i} \]

Will be the tuple:

\[ \pi = g^{rs} \prod_{i \in [n]} g_i^{a_i s}, \prod_{j \in [n]} g_j^{b_j r-t}, \prod_{i \in [n]} g_j^{a_i j-n+1+i}, \prod_{j \in [n]} g_j^{b_j r-t}, \prod_{i \in [n]} g_i^{a_i j-n+1+i} \]
\[ \hat{\pi} = \hat{g}^{rs} \prod_{i \in [n]} \hat{g}_i^{a_i s}, \prod_{j \in [n]} \hat{g}_j^{b_j r-t}, \prod_{i \in [n]} \hat{g}_j^{a_i j-n+1+i}, \prod_{j \in [n]} \hat{g}_j^{b_j r-t}, \prod_{i \in [n]} \hat{g}_i^{a_i j-n+1+i} \]
\[ \pi = \hat{h}^{rs} \prod_{i \in [n]} \hat{h}_i^{a_i s}, \prod_{j \in [n]} \hat{h}_j^{b_j r-t}, \prod_{i \in [n]} \hat{h}_j^{a_i j-n+1+i}, \prod_{j \in [n]} \hat{h}_j^{b_j r-t}, \prod_{i \in [n]} \hat{h}_i^{a_i j-n+1+i} \]

The function that makes the product commitments is in the folder 'prover', in the file product.cpp, and is called 'compute_all_products'.

A permutation commitment shows that given a permutation, and two vectors, of values, the values are following the permutation. We use it to show inputs that are supposed to be the same wire, have the same value, and that outputs have the same value as the inputs using the same wires.

When implementing this commitment, we found a problem in the self-loop originally described in the article. This self-loop has a wire with more outputs than inputs, and therefore cannot be committed. To solve this, we made a new self-loop, and its description could be found in the appendecses.

More technically, given as input a knowledge commitment to two vectors of values: \( b_i, a_j \), one of which is a permutation of the other, \( j = \rho(i) \).

The commitment will be the following products:
\[ \pi = \prod_{j \in [n]} \prod_{i \in [n]} g_{j(n+1)}^j \prod_{i \in [n]} g_{\rho(j)(n+2)-j}^{a_i} \prod_{i \in [n]} g_{\rho(j)(n+2)+i-j}^{b_i} \]

\[ \hat{\pi} = \prod_{j \in [n]} \prod_{i \in [n]} \prod_{j \in [n]} \hat{g}_{j(n+1)}^j \prod_{i \in [n]} \hat{g}_{\rho(j)(n+2)-j}^{a_i} \prod_{i \in [n]} \hat{g}_{\rho(j)(n+2)+i-j}^{b_i} \]

\[ \hat{\pi} = \prod_{j \in [n]} \prod_{i \in [n]} \prod_{j \in [n]} \hat{g}_{j(n+1)}^j \prod_{i \in [n]} \hat{g}_{\rho(j)(n+2)-j}^{a_i} \prod_{i \in [n]} \hat{g}_{\rho(j)(n+2)+i-j}^{b_i} \]

The function that makes the permutation commitments is in the folder 'prover', in the file perm.cpp, and is called 'all_perm_gen'.

During the making of

Verifier:

The verifier as one process:

The verifier uses a linear time check, with high probability, for the satisfiability of the circuit. It uses the CRS source, the circuit and the proof for this.

The verifier as two processes:

In this implementation, the verifier is divided to a preprocessor and a verifier.

The preprocessing is done in linear time. It could be used by many verifiers (for same circuit), and it’s correctness could be also verified in linear time. The output of the preprocessing is a file, that takes constant space. The same preprocessing file could also be used for several proofs, as long as they are for the same circuit. It input is a circuit and a CRS source (no proof is needed).

The verification is done in constant time (assuming the CRS and preprocessor are trustworthy).

Apart from serializing and reading the preprocess file, no additional computation is needed in this implementation, and as you can see in the running time charts, no noticeable time is added in total.

Verification pre-processing:

All group elements needed by the verifier that take linear time to compute, are computed in the pre-processing stage. These do not depend on the satisfying assignment but only on the circuit itself. For example, the product \( \prod_{j \in [n]} g_{j(n+1)} \), that is needed as part of the verification process.
Verifying Knowledge commitment:

Input will be a restricted knowledge argument \((c, \pi)\). Output is 1 iff
\[
e(c, h) = e(g, \pi)
\]

Verifying product commitment:

Output 1 iff the following equations hold:
\[
e(g, \hat{\pi}) = e(\pi, \hat{g}), e(g, \hat{\pi}) = e(\pi, \hat{h}),
\]
\[
e(c, d) = e(v, \prod_{j \in [n]} g_{j(n+1)}) e(g, \pi)
\]

Verifying permutation commitment:

Output 1 iff the following equations hold:
\[
e(g, \hat{\pi}) = e(\pi, \hat{g}), e(g, \hat{\pi}) = e(\pi, \hat{h}),
\]
\[
e\left(c, \prod_{j \in [n]} g_{j(n+1)}\right) = e\left(d, \prod_{j \in [n]} g_{\rho(j)(n+2) - j}\right) e(g, \pi)
\]

All verification functions exist in the folder 'verifier' (apart from verifying the CRS, which is in the folder 'shared')

For more detail please read Groth's article:
http://www0.cs.ucl.ac.uk/staff/J.Groth/ShortNIZK.pdf

A small bit on implementation choices:

We used files to communicate between processes (CRS generation, prover, verifier, pre-process verifier, and reading the circuits).

We use the boost serialization library, www.boost.org/libs/serialization/, by Robert Ramey.

For bilinear pairable group operations, we use the PBC library, http://crypto.stanford.edu/pbc/, primary contribution and upkeep by Ben Lynn.
**Project overview** (rectangle – file, ellipse – process):

- File contacting circuit with satisfying assignment
- Randomly generated satisfiable circuit
- Produce Prover/Verifier input files
- Circuit + wire values
- Prove
- CRS
- Produce CRS
- Circuit without values
- Pre-processor
- Pre-process output
- commitments
- Single process verifier
- True / False
- Verifier
- True / False
Part D - Methodology:

We are well satisfied with what we achieved. We did add two additional features to the protocol:

a. As referenced before, the CRS in our implementation is a separate 'entity' in the protocol. Both prover and verifier use it as a separate class. This means the prover and verifier's code could be used with a CRS outside the computer (a.k.a. on a server).

b. The two part verifier is another addition to the project, which we did not plan on initially.

Our computational time for the prover, single process verifier and pre-processing for the verifier became larger than expected in large circuits. The reason for that is that the computer ran out of RAM space, and needed to read and write to the disk (which is much slower).

As for following our schedule, as presented in the project plan, we managed to stay in the time frames:

- By milestone 1 we implemented all the basic modules, which include the CRS, restricted knowledge arguments generation and verification.

- By milestone 2 we were able to run basic complete full protocol tests - generate circuits (from file), make full proofs for them and verify them.

Now our project includes the ability to generate arbitrary large circuits, run the verifier as two processes and use the CRS as an external entity.
Time measurements:

- **Prover computation time (ms):**
  - The graph shows an increasing trend in computation time as the number of gates increases. The time increases exponentially with the number of gates.

- **Verifier pre-process (ms):**
  - Similarly, the verifier pre-process time also shows an increasing trend. The time increases linearly with the number of gates.
As you can see, the time it takes to read the CRS is the fastest increasing graph. This is because the bigger the CRS is, the more of it is on the disk, and not the RAM.
Step two of verification (with pre-process) (ms)

Single process verification