UCSB CS 291D: Blockchains and Cryptocurrencies **Fall 2020**

Lecture 12: zkSnark Internals II

Lecturer: Shumo Chu *Scribes: Rakshith Gopalakrishna*

In the last class we studied how to perform the blind evaluation of polynomials and the Knowledge Coefficient Test and Assumption. To recap, suppose that Alice has a polynomial P of degree d, and Bob has a point $s \in \mathbb{F}_p$ and let E be a homomorphic hiding. We want Alice to *blindly* evaluate the polynomial P at point s and return $E(P(s))$ to Bob. KCA states that if Alice responds with the pair (a', b') to Bob's challenge (a, b) with non-negligible probablity over Bob's choices of a, α, then she knows a γ such that $a' = \gamma a$ and $b' = \gamma b$.

Properties of blind evaluation

We require these two properties of blind evaluation.

1. Blindness – Alice should not learn s and Bob should not learn P.
2. Verifiability – The probability that Alice sends a value different from $E(P(s))$ for a P of degree d that is still accepted by Bob is negligible.

Recall that the polynomial P is defined as

$$P(x) = a_0x^0 + a_1x + a_2x^2 + \cdots + a_dx^d$$

In short, P is defined by the coefficients $[a_0, a_1, \ldots, a_d]$

One natural way to think of arriving at verifiable blind evaluation of polynomials is to extend the KCA protocol to use d α-pairs. Essentially repeating the KCA protocol d times so that we can handle the encoding of d coefficients in the polynomial P.

12.1 An Extended KCA

As discussed above, one could run the KCA protocol d times to achieve verifiable blind evaluation. However, the communication cost is very high. Ideally we want Alice to send back to Bob a single α-pair that can verify all the d α-pairs that Bob sends to Alice. If Bob can verify this single α-pair instead of repeating this for d rounds, we arrive at a communication efficient protocol. Notice that Alice can take any linear combination of the given d pairs and define her α-pair as $(a', b') = (\Sigma_{i=1}^{d} c_i a_i, \Sigma_{i=1}^{d} c_i b_i)$ for any $c_1, \ldots, c_d \in \mathbb{F}_p$.

In other words, if Bob sends d α-pairs to Alice, Alice only needs to send back a linear combination of the pairs back to Bob to prove that she knows some linear relation between the d pairs. This is the d-KCA protocol.

More formally, suppose Bob samples d α-pairs $(a_1, b_1), \ldots, (a_d, b_d)$ at random and sends these α-pairs to Alice. Suppose that Alice sends back a single α-pair (a', b'). Then, except with negligible probability, Alice knows $c_1, \ldots, c_d \in \mathbb{F}_p$ such that $\Sigma_{i=1}^{d} c_i a_i = a'$.

12.2 Verifiable Blind Evaluation protocol

Assume that the homomorphic hiding HH is the mapping $E(x) = x \cdot g$ for a generator g of a group G of size p.

1. Bob chooses $\alpha \xleftarrow{\$} D$ and constructs the α-pairs $(E(1), \alpha E(1)), (E(s), \alpha E(s)), \ldots, (E(s^d), \alpha E(s^d))$ and sends these to Alice.
2. Alice computes $a' = E(P(s)) = P(s) \cdot g$ and $b = \alpha E(P(s)) = \alpha P(s) \cdot g$ and sends this α pair to Bob.
3. Bob checks that $b' = \alpha \cdot a'$ and accepts $(a;, b')$ iff this equality holds.

This protocol is provides blind evaluation of a polynomial as long as d-KCA is true.

12.3 Expressing computations as polynomials

Quadratic Arithmetic Programs (QAP) are the extremely useful translation of computations into polynomials. They form the basis of modern zk-SNARK constructions.

Suppose that Alice wants to prove to Bob that she knows the satisfying assignment for the equation $x^3 + x + 5 = 35$. The computation here is the function $f(x) = x^3 + x + 5$. One could represent this function in the form of the following python program.

```
def f(x):
    y = x**3
    return x + y + 5
```

In order to represent this computation as a polynomial, we need to go through the intermediate steps of representing them as arithmetic circuits, R1CS and finally QAP.

12.3.1 Computation to arithmetic circuit

First step is to flatten the program into so that all operations are of the form $var_1 = var_2 < op > var_3$ where $op = +, -, *, /$. In the process of doing this, we can also introduce as many intermediate values as required. This step is very similar to Single Static Assignment used in compilers.

Our flattened program looks like this. Note that the variables once defined are immutable.

```
sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

From the flattened program, we can now construct an arithmetic circuit by replacing each variables on the left, right and the operator with output of a gate, input to a gate and an arithmetic gate respectively.

The arithmetic circuit for the given program is given in figure 12.1.

12.3.2 Circuit to R1CS

We define the vector C to denote the assignment to the variables used in the program. Let $C = [\sim one, x, \sim out, sym\_1, y, sym\_2]$. Let C_0 denote the concrete assignment to the variables. For instance, $C_0 = [1, 5, 7, 8, 6, 10]$.

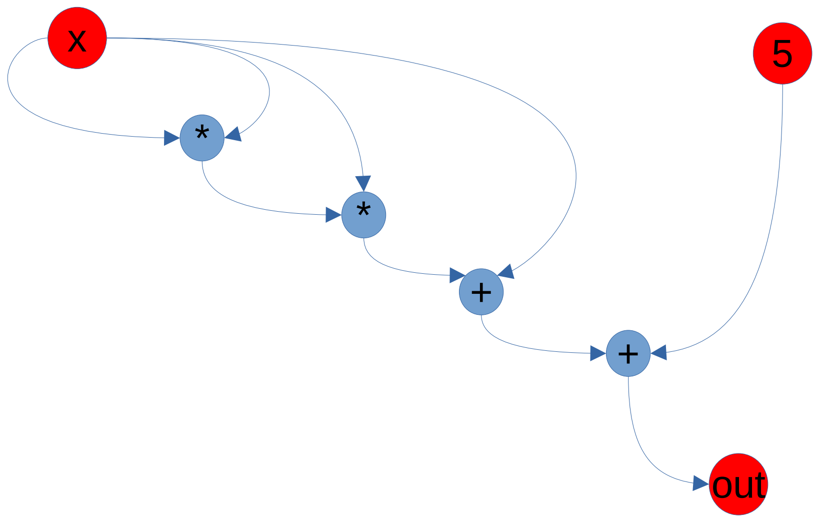

Figure 12.1: Arithmetic circuit for the computation $x^3 + x + 5$

R1CS is a sequence of groups of three vectors (l, r, o). Suppose c is the concrete assignment to the variables, the computation in each gate should obey the following format

$$< c.l > \times < c.r > = < c.o >$$

where $< . >$ denotes the inner product of two vectors.

Let us now visualize how each line in the flattened program can be represented in the R1CS form.

1. $sym\_1 = x \times x$

$$l = [0, 1, 0, 0, 0, 0]$$
$$r = [0, 1, 0, 0, 0, 0]$$
$$o = [0, 0, 0, 1, 0, 0]$$

2. $sym\_1 * x = y$

$$l = [0, 0, 0, 1, 0, 0]$$
$$r = [0, 1, 0, 0, 0, 0]$$
$$o = [0, 0, 0, 0, 1, 0]$$

3. $y + x = sym\_2$

$$l = [0, 1, 0, 0, 1, 0]$$
$$r = [1, 0, 0, 0, 0, 0]$$
$$o = [0, 0, 0, 0, 0, 1]$$

4. $sym\_2 + 5 =\sim out$

$$l = [5, 0, 0, 0, 0, 1]$$
$$r = [1, 0, 0, 0, 0, 0]$$
$$o = [0, 0, 1, 0, 0, 1]$$

Now that we have an R1CS with four constraints, the witness is simply the assignment to all variables including the input, output and intermediate variables.

12.3.3 R1CS to QAP

The next step is to convert the R1CS into QAP form. In order to do this, we go from four groups (for the four gates) of three vectors of length 6 (for 6 variables) to six groups of three degree-3 polynomials where evaluating the polynomimals at each x represents one of the constraints.

Let $L(x)^T = [L_1(x), L_2(x), \ldots, L_6(x)]$, $R(x)^T = [R_1(x), R_2(x), \ldots, R_6(x)]$ and $O(x)^T = [O_1(x), O_2(x), \ldots, O_6(x)]$.

Rewriting the R1CS for the arithmetic circuit $C \cdot L(x) \times C \cdot R(x) - C \cdot O(x) = 0$ as equation 12.1

$$C \cdot \begin{bmatrix} L_1(x) \\ L_2(x) \\ \vdots \\ L_6(x) \end{bmatrix} \times C \cdot \begin{bmatrix} R_1(x) \\ R_2(x) \\ \vdots \\ R_6(x) \end{bmatrix} - C \cdot \begin{bmatrix} O_1(x) \\ O_2(x) \\ \vdots \\ O_6(x) \end{bmatrix} = 0 \tag{12.1}$$

helps us visualize the QAP. Each row can be considered a group of polynomials and this is how we arrive at six groups of three degree-3 polynomials.

We can find out the the coefficients of the polynomials $L_1, \ldots, L_6, R_1, \ldots, R_6, O_1, \ldots, R_6$ using Lagrange interpolation. Essentially, given a set of points, Lagrange interpolation allows us to construct a polynomials that passes through all of them. Suppose we want a polynomial that passes through $(1, 3), (2, 2)$ and $(3, 4)$. We can start by constructing a polynomial that passes through $(1, 3), (2, 0), (3, 0)$, a second polynomial that passes through $(1, 0), (2, 2), (3, 0)$ and a third polynomial that passes through $(1, 0), (2, 0), (3, 4)$. Now, adding these polynomials gives us the required polynomial that passes through all the points.

12.4 Quadratic Arithmetic Program

A quadratic arithmetic program Q of degree d and size m consists of polynomials $L_1, \ldots, L_m, R_1, \ldots, R_m, O_1, \ldots, O_m$, a target polynomial T of degree d. Define $L \coloneqq \Sigma_{i=1}^{m} c_i \cdot L_i$, $R \coloneqq \Sigma_{i=1}^{m} c_i \cdot R_i$ and $O \coloneqq \Sigma_{i=1}^{m} c_i \cdot O_i$ and then define $P \coloneqq L \cdot R - O$. An assignment $(c_1, \ldots, c_m)$ satisfies Q iff T divides P.