Pinocchio - Nearly Practical Verifiable Computation

Advai Swamy- 106119006, Dipesh Kafle - 106119029 and Harsh Khandelwal - 106119048

National Institute of Technology, Tiruchirappalli

Introduction

What is verified computing?

- Since computational power is often asymmetric (particularly for mobile devices), a relatively weak client may wish to outsource computation to one or more powerful workers
- In all of these settings, the client should be able to verify the results returned, to guard against malicious or malfunctioning workers
- Most of VC work has either been function specific, relied on assumptions we prefer to avoid, or simply failed to pass basic practicality requirements.
- Pinocchio is a concrete system for efficiently verifying general computations while making only cryptographic assumptions. It supports even Public Verifiable Computation ,which allows an untrusted worker to produce signatures of computation.

Introduction (continued)

- Pinocchio's asymptotics are excellent: key setup and proof generation require cryptographic effort linear in the size of the original computation, and verification requires time linear in the size of the inputs and outputs.
- To achieve efficient verifiable computation, Pinocchio combines quadratic programs, a computational model introduced by Gennaro et al. , with a series of theoretical refinements and systems engineering to produce an end-to-end toolchain for verifying computations.

Homormophic Hiding

An HH E(x) of a number x is a function satisfying the following properties:

- For most x's, given E(x) it is hard to find x.
- Different inputs lead to different outputs – so if $x \neq y$, then $E(x) \neq E(y)$.
- If someone knows E(x) and E(y), they can generate the HH of arithmetic expressions in x and y. For example, they can compute E(x+y) from E(x) and E(y).

Homormophic Hiding Example

- We do regular addition but then apply (modn) on the result to get back a number in the range $\{0,\dots,n-1\}$
- For a prime p, we can use the modp operation to also define a multiplication operation over the numbers $\{1,\dots,p-1\}$ in a way that the multiplication result is also always in the set $\{1,\dots,p-1\}$ – by performing regular multiplication of integers, and then taking the result modp.
- Let us assume this set of elements together with this operation is referred to as the group $\mathbb{Z}_p^*$

Homormophic Hiding Example(continued)

$\mathbb{Z}_p^*$ has the following properties :

- It is a cyclic group; which means that $\exists g \in \mathbb{Z}_p^*$ called a generator such that all elements of $\mathbb{Z}_p^*$ can be written as g^a for some $a \in \{0, .., p-2\}$, where we define $g^0 = 1$.
- The discrete logarithm problem is believed to be hard in $\mathbb{Z}_p^*$. This means that, when p is large, given an element h in $\mathbb{Z}_p^*$ it is difficult to find the integer a in 0,..,p−2 such that $g^a = h(modp)$.
- As "exponents add up when elements are multiplied", we have for a,b in 0,..,p−2 $g^a \cdot g^b = g^{a+b(modp-1)}$.

We have now constructed an HH that "supports addition" – meaning that E(x+y) is computable from E(x) and E(y)

Blind Evaluation of Polynomial

- A polynomial P of degree d over $\mathbb{F}_p$ is an expression of the form $P(s) = a_0 + a_1.s + a_2.s^2 + ... + a_d.s^d$ for some $a_0, .., a_d \in \mathbb{F}_p$

Suppose Alice has a polynomial P of degree d, and Bob has a point $s \in \mathbb{F}_p$ that he chose randomly. Bob wishes to learn E(P(s)), i.e., the HH of the evaluation of P at s.

- Using HH, we can perform blind evaluation as follows.
  - Bob sends to Alice the hidings E(1),E(s),...,E(sd).
  - Alice computes E(P(s)) from the elements sent in the first step, and sends E(P(s)) to Bob. (Alice can do this since E supports linear combinations, and P(s) is a linear combination of 1,s,...,s^d.)

QAP (Quadratic Arithmetic Program)

- Gennaro, Gentry, Parno and Raykova(GGPR) defined an extremely useful translation of computations into polynomials called a Quadratic Arithmetic Program (QAP).
- QAPs are sets of polynomials.
- QAPs are building blocks to encode circuits into polynomials t and assignments into polynomials p.

How to compute QAP?

- Let I be circuit indices({1,..,m}). Then,

$$QAP := \{t, \{v_k\}_{k\in I}, \{w_k\}_{k\in I}, \{y_k\}_{k\in I}\}$$

- Choose random elements $\{m_1, \cdots m_k\}$ from base field for every multiplication vertex in the circuit.
- Target polynomial t(x) is computed as $(x - m_1) \otimes ... \otimes (x - m_k)$
- Polynomial from $\{v_k\}_{k\in I}$ is 1 at m_j , if edge k is left input to multiplication gate $\otimes m_j$ and zero at m_j , otherwise.
- Polynomial from $\{w_k\}_{k\in I}$ is 1 at m_j , if edge k is right input to multiplication gate $\otimes m_j$ and zero at m_j , otherwise.

How to compute QAP?(continued)

- Polynomial from $\{y_k\}_{k \in I}$ is 1 at m_j , if edge k is output of multiplication gate $\otimes m_j$ and zero at m_j , otherwise.
- Circuit assignment $\{c_k\}_{k \in I}$ defines the polynomial, p $:= (\sum_{k \in I} c_k v_k).(\sum_{k \in I} c_k w_k) - (\sum_{k \in I} c_k y_k)$

QAP Example

$QAP_{\mathbb{F}_{11}}(C_f) = \{x^2 + 10x + 2, V, W, Y\}$
$where$

$V = \{5x + 9, 0, 0, 6x + 3, 0\}$

$W = \{0, 5x + 9, 6x + 3, 0, 0\}$

$Y = \{0, 0, 0, 5x + 9, 6x + 3\}$

- $\{c_k\}_{k \in I}$ is valid assignment $\iff$ p is divisible by t.

Valid example I = {2, 3, 4, 6, 2}:

- (2(5x + 9) + 6(6x + 3)) · (3(5x + 9) + 4(6x + 3)) − (6(5x + 9) + 2(6x + 3))
- (2 · 5x +2 · 9+6 · 6x +6 · 3) · (3 · 5x +3 · 9+4 · 6x +4 · 3)−(6 · 5x +6 · 9+2 · 6x +2 · 3) =
- (10x + 7 + 3x + 7) · (4x + 5 + 2x + 1) − (8x + 10 + 1x + 6) =
- (2x + 3) · (6x + 6) − (9x − 5) =
- $x^2 + x + 7x + 7 + 2x + 6$
- ⇒ p(x) = $x^2 + 10x + 2$ – Equal to t hence divisible

Assumptions Moving Forward

- Let size of QAP be m and number of Input/Output variables of function F be N. m>=N as m also includes intermediate computation outputs. Degree of QAP(d) = degree(t(x)).
- Let $I_{mid} = \{N + 1, .., m\}$ be the non IO related indices.

Pinocchio Protocol

1. $(EK_F, VK_F) \leftarrow KeyGen(F, 1^\lambda)$
   $\Rightarrow$ F be the function with N input/output values from $\mathbb{F}$, λ is the security parameter, EK_F is the Evaluation Key and VK_F is the Verification Key
2. $(y, \pi_y) \leftarrow Compute(EK_F, u)$
   $\Rightarrow$ u is the input, y $\leftarrow$ F(u) and the π_y is proof of y's correctness
3. $\{0, 1\} \leftarrow Verify(VK_F, u, y, \pi_y)$
   $\Rightarrow$ The deterministic verification algorithm outputs 1 if F(u) = y, and 0 otherwise.

Requirements of the protocol

- A finite cyclic group (G, $\cdot$)
- A generator g of that group
- A bilinear map B($\cdot$, $\cdot$) : $G \times G \rightarrow G_T$, such that:
    - Order: G_T and G have the same order
    - Bilinearity: $e(g^j, h^k) = e(g,h)^{j.k}$ for all j,k $\in \mathbb{Z}, g, h \in G$

This is usually there in cryptographically strong, pairing friendly elliptic curve.

KeyGen

- Choose random elements $r_v, r_w, s, \alpha_v, \alpha_w, \alpha_y, \beta, \gamma \in \mathbb{F}$
- Set $r_y = r_v \cdot r_w, g_v = g_{r_v}, g_w = gr_w and g_y = gr_y$.

Evaluation Key (EK_F)

- Given generator g and circuit degree d:

$$\left\{ \begin{array}{lll} \{g^{r_v v_k(s)}\}_{k \in I_{mid}} & \{g^{r_w w_k(s)}\}_{k \in I_{mid}} & \{g^{r_v r_w y_k(s)}\}_{k \in I_{mid}} \\ \{g^{r_v \alpha_v v_k(s)}\}_{k \in I_{mid}} & \{g^{r_w \alpha_w w_k(s)}\}_{k \in I_{mid}} & \{g^{r_v r_w \alpha_y y_k(s)}\}_{k \in I_{mid}} \\ \{g^{s^i}\}_{i \in \{1,\ldots,d\}} & & \{g^{\beta(r_v v_k(s) + r_w w_k(s) + r_v r_w y_k(s))}\}_{k \in I_{mid}} \end{array} \right\}$$

- It is the set of group elements, used to encrypt the non I/O(internal computations) related part of the polynomial p. This is clear from the usage of I_{mid} which signifies the non IO related indices.[1]
- Size depends linear on the number of internal (non I/O) edges in the circuit.

KeyGen (continued)

Verification Key (VK_F)

- Given generator g:

$$\left\{ \begin{array}{lllc} g^1 & g^{\alpha_v} & g^{\alpha_w} & g^{\alpha_\gamma} \\ g^\gamma & g^{\beta\gamma} & g^{t(s)} & \\ & & & \{g^{r_v v_k(s)}, g^{r_w w_k(s)}, g^{r_v r_w y_k(s)}\}_{k \in I_{I/O}} \end{array} \right\}$$

- It is the set of group elements, used to encrypt the I/O related part of the polynomial p.
- Size depends linear on the number of I/O edges in the circuit.
- Delete all the randomly generated values.

Compute

- Computation
  ⇒Given input set I_{in} , execute circuit C_f to compute intermediate values I_{mid} and result I_{out} .
- Proof Generation
  - Use valid assignment I and QAP to compute polynomial p.
  - Derive quotient polynomial h = p/t.
  - Use EK_F to generate π_y

$$\left\{ \begin{array}{llll} g^{r_v v_m(s)}, & g^{r_w w_m(s)}, & g^{r_v r_w y_m(s)}, & g^{h(s)} \\ g^{r_v \alpha_v v_m(s)}, & g^{r_w \alpha_w w_m(s)}, & g^{r_v r_w \alpha_y y_m(s)} & \\ & & & g^{r_v \beta v_m(s)} \cdot g^{r_w \beta w_m(s)} \cdot g^{r_v r_w \beta y_m(s)} \end{array} \right\}$$

where $v_m(x) = \sum_{k \in I_{mid}} c_k v_k(x), w_m(x) = \sum_{k \in I_{mid}} c_k w_k(x), y_m(x) = \sum_{k \in I_{mid}} c_k y_k(x)$

Compute (continued)

How does the proof get generated from EK_F?

- All v_k's, w_k's and y_k's are part of the QAP
- We don't know any of the random values used during the Keygen phase.
- We use EK_F and exponential laws to generate the proof
  - $g^x . g^y = g^{x+y}$
  - $(g^x)^y = g^{x.y}$
- All c_k are known from execution
- All $g^{r_v . v_k(s)}, g^{r_v . \alpha_v . v_k(s)}$ are provided in the EK_F

$$g^{r_v v_m(s)} = g^{r_v \sum_{k \in I_{mid}} c_k v_k(s)} = \prod_{k \in I_{mid}} (g^{r_v v_k(s)})^{c_k}$$

$$g^{r_v \alpha_v v_m(s)} = g^{r_v \sum_{k \in I_{mid}} c_k \alpha_v v_k(s)} = \prod_{k \in I_{mid}} (g^{r_v \alpha v_k(s)})^{c_k}$$

Verification

- Given input set I_{in} , output set I_{out} and proof π
- Verify computer(worker) knows p such that p is divisible by t(target polynomial).
- The verification of an alleged proof with elements $\pi = \{g^{V_{mid}}, g^{W_{mid}}, g^{Y_{mid}}, g^{H}, g^{V'_{mid}}, g^{W'_{mid}}, g^{Y'_{mid}}, g^{Z}\}$ uses the public verification key VK_F and the pairing function e for the following checks.

Verification

- *Divisibility check for the QAP: using elements from VK_F compute $g_v^{v_{io}(s)} = \prod_{k\in[N]} \left(g_v^{v_k(s)}\right)^{c_k}$ (and similarly for $g_w^{w_{io}(s)}$ and $g_y^{y_{io}(s)}$), and check:*

$$e(g_v^{v_0(s)} g_v^{v_{io}(s)} g_v^{V_{mid}}, g_w^{w_0(s)} g_w^{w_{io}(s)} g_w^{W_{mid}}) = \tag{1}$$

$$e(g_y^{t(s)}, g^H) e(g_y^{y_0(s)} g_y^{y_{io}(s)} g_y^{Y_{mid}}, g). \tag{2}$$

- *Check that the linear combinations computed over $\mathcal{V}$, $\mathcal{W}$ and $\mathcal{Y}$ are in their appropriate spans:*

$$e(g_v^{V'_{mid}}, g) = e(g_v^{V_{mid}}, g^{\alpha_v}), \quad e(g_w^{W'_{mid}}, g) = e(g_w^{W_{mid}}, g^{\alpha_w}),$$

$$e(g_y^{Y'_{mid}}, g) = e(g_y^{Y_{mid}}, g^{\alpha_y}).$$

- *Check that the same coefficients were used in each of the linear combinations over $\mathcal{V}$, $\mathcal{W}$ and $\mathcal{Y}$:*

$$e(g^Z, g^\gamma) = e(g_v^{V_{mid}} g_w^{W_{mid}} g_y^{Y_{mid}}, g^{\beta\gamma}).$$

Zero Knowledge

- Suppose the worker does not want to publish (some of) the inputs

Verifier side changes

- Extend verifier key in setup phase with $\{g^{r_v.\alpha_v.t(s)}, g^{r_w.\alpha_w.t(s)}, g^{r_y.\alpha_y.t(s)}, g^{r_v.\beta.t(s)}, g^{r_w.\beta.t(s)}, g^{r_y.\beta.t(s)}\}$

Worker Side Changes

- Generate random elements δ_v , δ_w , δ_y , and change
  - $v_{mid}(x) \Rightarrow v_{mid}(x) + \delta_v t(x)$
  - $v(x) \Rightarrow v(x) + \delta_v t(x)$
  - $w(x) \Rightarrow w(x) + \delta_w t(x)$
  - $y(x) \Rightarrow y(x) + \delta_y t(x)$

$$p := (\sum_{k \in I} c_k.v_k + \delta_v.t(x)) \cdot (\sum_{k \in I} c_k.w_k + \delta_w.t(x)) - (\sum_{k \in I} c_k.y_k + \delta_y.t(x))$$

has the same divisibility property with respect to t.

References

- Electric Coin Blog
- Vitalic Buterin Medium Blog on QAP
- risencrypto zkSnarks blog
- The Mathematics behind zkSnarks(LeastAuthority Youtube Video)