PRINCETON UNIV. F'16 COS 521: ADVANCED ALGORITHM DESIGN

# Lecture 21: Taste of cryptography: Secret sharing and secure multiparty computation

Lecturer: *Sanjeev Arora* Scribe:*Sanjeev Arora*

Cryptography is the ancient art/science of sending messages so they cannot be deciphered by somebody who intercepts them. This field was radically transformed in the 1970s using ideas from computational complexity. Encryption schemes were designed whose decryption by an eavesdropper requires solving computational problems (such as integer factoring) that're believed to be intractable. You may have seen the famous RSA cryptosystem at some point. It is a system for giving everybody a pair of keys (currently each is a 1024-bit integer) called a *public key* and a *private key*. The public key is published on a public website; the private key is known only to its owner. Person $x$ can look up person $y$'s public-key and encrypt a message using it. Only $y$ has the private key necessary to decode it; everybody else will gain no information from seeing the encrypted message.

It is interesting to note what it means to gain no information: it means that the eavesdropper is unable to distinguish the encrypted messages from a truly random string of bits. (Remember we discussed using markovian models to check if your friend is able to produce a truly random bit sequence. That test, and every other *polynomial-time procedure* will fail to distinguish the encrypted message from a random sequence.)

Since the 1980s though, the purview of cryptography greatly expanded. In inventions that anticipated threats that wouldn't materialize for another couple of decades, cryptographers designed solutions such as private multiparty computation, proofs that yield nothing but their validity, digital signatures, digital cash, etc. Today's lecture is about one such invention due to Ben-or, Goldwasser and Wigderson (1988), secure multiparty computation, which builds upon the Reed Solomon codes studied last time.

The model is the following. There are $n$ players, each holding a private number (say, their salary, or their *vote* in an election). The $i$th player holds $s_i$. They wish to compute a joint function of their inputs $f(s_1, s_2, \ldots, s_n)$ such that nobody learns anything about anybody else's secret input (except of course what can be inferred from the value of $f$). The function $f$ is known to everybody in advance (e.g., $s_1^2 + s_2^2 + \cdots + s_n^2$).

Admittedly, this sounds impossible when you first hear it.

## 0.1 Shamir's secret sharing

We first consider a *static* version of the problem that introduces some of the ideas.

Say we want to distribute a secret, say $a_0$, among $n$ players. (For example, $a_0$ could be the secret key to decrypt an important message.) We want the following properties: (a) every subset of $t+1$ people should be able to pool their information and recover the secret, but (b) no subset of $t$ people should be able to pool their information to recover any information at all about the secret.

For simplicity interpret $a_0$ as a number in a finite field $Z_q$. To share this secret, pick $t$ random numbers $a_1, a_2, \ldots, a_t$ in $Z_q$ and construct the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 +$

$\cdots + a_t x^t$ and evaluate it at $n$ points $\alpha_1, \alpha_2, \ldots, \alpha_n$ that are known to all of them (these are the "names"of the players). Then give $p(\alpha_i)$ to person $i$.

Notice, the set of shares are $t$-wise independent random variables. (Each subset of $t$ shares is distributed like a random $t$-tuple over $Z_q$.) This follows from polynomial interpolation (which we explained last time using the Vandermode determinant): for every $t$-tuple of people and every $t$-tuple of values $y_1, y_2, \ldots, y_t \in Z_q$, there is a unique polynomial whose constant term is $a_0$ and which takes these values for those people. Thus every $t$-tuple of values is equally likely, *irrespective of* $a_0$, and gives no information about $a_0$.

Furthermore, since $p$ has degree $t$, each subset of $t+1$ shares can be used to reconstruct $p(x)$ and hence also the secret $a_0$.

Let's formalize this property in the following definition.

DEFINITION 1 ($(t,n)$- SECRETSHARING) *If $a_0 \in Z_q$ then its* $(t,n)$- secretsharing *is a sequence of $n$ numbers $\beta_1, \beta_2, \ldots, \beta_n$ obtained above by using a polynomial of the form $a_0 + \sum_{i=1}^{t} a_i x^i$, where $a_1, a_2, \ldots, a_n$ are random numbers in $Z_q$.*

## 0.2 Multiparty computation: the model

Multiparty computation vastly generalizes Shamir's idea, allowing the players to do arbitrary algebraic computation on the secret input using their "shares."

Player $i$ holds secret $s_i$ and the goal is for everybody to know a $(t,n)$-secretsharing for $f(s_1, s_2, \ldots, s_n)$ at the end, where $f$ is a publicly known function (everybody has the code). Thus no subset of $t$ players can pool their information to get any information about anybody else's input that is not implicit in the output $f(s_1, s_2, \ldots, s_n)$. (Note that if $f()$ just outputs its first coordinate, then there is no way for the first player's secret $s_1$ to not become public at the end.)

We are given a *secret* channel between each pair of players, which cannot be eavesdropped upon by anybody else. Such a secret channel can be ensured using, for example, a public-key infrastructure. If everybody's public keys are published, player $i$ can look up player $j$'s public-key and encrypt a message using it. Only player $j$ has the private key necessary to decode it; everybody else will gain no information from seeing the encrypted message.

## 0.3 Example: linear combinations of inputs

First we describe a simple protocol that allows the players to compute $(t,n)$ secret shares for the sum of their inputs, namely $f(s_1, s_2, \ldots, s_n) = \sum_i s_i$.

As before, let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be $n$ distinct nonzero values in $Z_q$ that denote the player's names.

Each player does a version of Shamir's secret sharing. Player $i$ picks $t$ random numbers $a_{i1}, a_{i2}, \ldots, a_{it} \in Z_q$ and evaluates the polynomial $p_i(x) = s_i + a_{i1}x + a_{i2}x^2 + \cdots + a_{it}x^t$ at $\alpha_1, \alpha_2, \ldots, \alpha_n$, and sends those values to the respective $n$ players (keeping the value at $\alpha_i$ for himself) using the secret channels. Let $\gamma_{ij}$ be the secret sent by player $i$ to player $j$.

After all these shares have been sent around, the players get down to computing shares for $f$, i.e., $\sum_i s_i$. This is easy. Player $k$ computes $\sum_i \gamma_{ik}$. In other words, he treats the shares he received from the others as *proxies* for their input.

OBSERVATION: *The numbers computed by the kth player correspond to value of the following polynomial at $x = \alpha_k$:*

$$\sum_i s_i + \sum_r (\sum_i a_{ir}) x^r.$$

This is just a random polynomial whose constant term is $\sum_i s_i$. Thus the players have managed to do $(t, n)$-secret sharing for the sum.

It is trivial to change the above protocol to compute any *weighted* sum: $f(s_1, s_2, \ldots, s_n) = \sum_i c_i s_i$ where $c_i \in Z_q$ are any constants known to all of them. This just involves taking the corresponding weighted sum of their shares.

Furthermore, this can also be used to compute *multiplication by a matrix*: $f(s_1, s_2, \ldots, s_n) = M \cdot \vec{s}$ where $M$ is a matrix. The reason is that matrix vector multiplication is just a sequence of weighted sums.

## 0.4 Breaking up computations into straight line programs

The above protocol applies only to a simple function, the sum. How can we generalize it to a larger set of functionalities?

We define the set of functionalities via algebraic programs, which capture general algebraic computation over a finite field.

DEFINITION 2 (ALGEBRAIC PROGRAMS) *A size $m$ algebraic straight line program with inputs $x_1, x_2, \ldots, x_n \in Z_q$ is a sequence of $m$ lines of the form*

$$y_i \leftarrow y_{i_1} \, op \, y_{i_2},$$

*where $i_1, i_2 < i$; $op =$ "$+$"or "$\times$,"or "$-$"and $y_i = x_i$ for $i = 1, 2, \ldots, n$. The output of this straight line program is defined to be $y_m$.*

A simple induction shows that a straight line program with inputs $x_1, x_2, \ldots x_n$ computes a multivariate polynomial in these variables. The degree can be rather high, about $2^m$. So this is a powerful model.

(Aside: Straight line programs are sometimes called *algebraic circuits*. If you replace the arithmetic operations with boolean operations $\vee, \neg, \wedge$ you get a model that can do any computation at all, where $T$ steps of the Turing machine correspond to a straight line program of length $O(T \log T)$.)

## 0.5 General protocol: $+$ and $\times$ suffice

Our definition of algebraic programs shows that if we can design a protocol that allows *addition* and *multiplication* of secret values, then that is good enough to implement any algebraic computation. All players start by writing out for themselves the above straight line program. Let the variables in the algebraic program be $y_1, y_2, \ldots, y_m$.

The protocol has $m$ rounds, and maintains the invariant that *by the end of the ith round the players hold $n$ values in some $(t, n)$-secret sharing for $y_i$*. In the first $n$ rounds they just send each other shares in their private inputs, so the protocol becomes interesting in the $n + 1$th round.

Say the $i+1$th line in the algebraic program is $y_{i+1} = y_{i_1} + y_{i_2}$. Then we already know what the players can do: just add up the share they have in the secret sharings of $y_{i_1}$ and $y_{i_2}$ respectively. We already saw that this works.

So assume $y_{i+1}$ is the $\times$ of two earlier variables. If these two earlier variables were secretshared using polynomials $g(x) = \sum_{r=0}^{t} g_r x^r$ and $h(x) \sum_{r=0}^{t} h_r x^r$ then the values being secretshared are $g_0, h_0$ and the obvious polynomial to secretshare their product is $\pi(x) = g(x)h(x) = \sum_{r=0}^{2t} x^r \sum_{j \le r} g_j h_{r-j}$. The constant term in this polynomial is $g_0 h_0$ which is indeed the desired product. Secretsharing this polynomial means everybody takes their share of $g$ and $h$ respectively and multiplies them. Are we done?

Unfortunately, this polynomial $\pi$ has two problems: the degree is $2t$ instead of $t$ and, more seriously, its coefficients are not random numbers in $Z_q$. (For example, polynomials with random coefficients are very unlikely to factor into the product of two polynomials.) Thus it is not a $(t, n)$-secretsharing of $g_0 h_0$.

The degree problem is easy to solve: just drop the higher degree terms and stay with the first $t$ terms. Dropping terms is a linear operation and can be done using a suitable matrix-vector product, which is done by the simple protocol of Section 0.3. We won't go into details.

To solve the problem about the coefficients not being random numbers, each of the players does the following. The $k$th player picks a random degree $2t$ polynomial $r_k(x)$ whose constant term is 0. Then he secret shares this polynomial among all the other players. Now the players can compute their secretshares of the polynomial

$$\pi(x) + \sum_{k=1}^{n} r_k(x),$$

and the constant term in this polynomial is still $g_0 h_0$. Then they apply truncation to this procedure to drop the higher order terms. Thus at the end the players have a $(t, n)$-secretsharing of the value $y_{i+1}$, thus maintaining the invariant.

**Important Subtlety:** The above description assumes that the malicious players follow the protocol. In general the $t$ malicious players may not follow the protocol in an attempt to learn things they otherwise can't. Modifying the protocol to handle this —and proving it works—is more nontrivial.

BIBLIOGRAPHY

1. M. BenOr, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault Tolerant Distributed Computation Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC'88), Chicago, Illinois, pages 1-10, May 1988.

2. A. Shamir. How to share a secret", Communications of the ACM 22 (11): 612–613, 1979.