CSCI 1515: Applied Cryptography

P. Miao

Spring 2023

These are lecture notes for CSCI 1515: Applied Cryptography taught at Brown University by Peihan Miao in the Spring of 2023.

These notes are taken by Jiahua Chen with gracious help and input from classmates and fellow TAs. Please direct any mistakes/errata to me via email, post a thread on Ed, or feel free to pull request or submit an issue to the notes repository.

Notes last updated March 21, 2023.

Contents

§1 March 16, 2023

§1.1 Succinct Non-Interactive Argument (SNARG)

Definition 1.1

A non-interactive proof (unbounded prover) or argument (polybounded prover) system is succinct if

- The proof π is of length $|\pi| = \text{poly}(\lambda, \log |c|)$.
- The verifier runs in time $\text{poly}(\lambda, |x|, \log |c|)$.

A SNARK is a Succinct Non-Interactive Argument of Knowledge. A zk-SNAR{K/G} is a SNAR{K/G} with zero-knowledge.

We rely on this theorem (without proof):

Theorem 1.2 (PCP Theorem)

Every NP language has a PCP where the verifier reads only a *constant* number of bits of the proof.

This theorem states that a verifier only needs to read a constant number of bits of the proof, which could condense a proof into some constant time.

Our first attempt could be to commit to the entire proof, and the verifier challenges with the bits it would like to observe, which the prover will open.

However, this entire message will still send the entire proof, which isn't what we want.

Our solution is to use a Merkle Tree.

We hash values in a tree format, with each parent node being the hash of its children. We only send the *root note*. Whenever the verifier requests a certain bit, we send the path from the root to the bit (revealing all hashes, and siblings) to verify that this is indeed.

It's very difficult to change any bit. If we changed a bit, at some point up the path of the tree we'll have found a collision for a hash. That is to say, a specific bit being correct is predicated on whether the path to the root is valid and the root hash matches.

Can we make this hiding? Right now, we don't guarantee the hiding property. If we only had one layer, every bit would be revealed. How can we modify this algorithm to ensure that each bit is hiding?

One solution would be to add a random string r as a sibling to every leaf. However, this would require us to reveal all siblings when we're verifying a certain leaf node. We can easily modify this to *salt every* leaf node. We can add some random r_i to the hash of *every* bit that hides those bits.

Now, instead of sending a commitment of the entire proof, we send a Merkle Tree of the commitment of the proof. Then, when requested for certain bits i, j, k, we'll open those commitments as pathes on the tree.

Is this zero-knowledge? Note that in the PCP theorem, we did not have the zero-knowledge property. Our solution is that when opening commitments, we can instead provide ZK proofs for our 'reveals' instead of the actual bits themselves. Asymptotically, this still preserves our succinctness property.

Theoretically, this lets us construct zk-SNARGs. However, this is not practical.

§1.1.1 Linear PCP

We can use a Linear PCP instead. The prover sends $\pi \in \mathbb{F}^m$, instead of checking a constant number of bits, we check a constant number of inner products $\langle \pi, q_i \rangle \in \mathbb{F}$. Constructions have size $m = O(|c|^2)$ for Walsh-Hadard codes or $m = O(|c|)$ for quadratic span programs.

The verifier will preprocess first by generating public and secret keys. The verifier encrypts challenges $q_i \in \mathbb{F}^m$ and sends $c_i \leftarrow \mathsf{Enc}_{pk}(q_i)$. The public and ciphertexts are published.

The prover will generate such $\pi \in \mathbb{F}^m$, and in an additively homomorphic encryption scheme, the prover can provide $\mathsf{Enc}_{pk}(\langle \pi, q_i \rangle) \to r_i$.

The verifier checks x, τ_{LPCP} and r_is in quadratic time.

However, this is *designated verifier*. Every verifier will need to generate their values q_i and c_i. This doesn't allow for us to publicly verify, and the preprocessing doesn't really allow for the random oracle model.

We can, however, use the common reference string (CRS) model. The assumption is that some trusted third party will generate a 'common reference string' $c_i = g^{q_t}$ and $c_\tau = g^{\tau_{\text{LPCP}}}$.

The prover can still multiply ciphertexts to get $g^{\langle \pi, q_i \rangle} = g^{r_i}$ (exponents are linear). Then the verifier can take $g^x, g^{\tau_{\text{LPCP}}}, g^{r_i}$ to verify.

But the verifier does not have the secret key! There are some nasty ways around this, but we'll use a piece of new technology in cryptography called *Bilinear Pairings*.

Definition 1.3 (Bilinear Pairing)

A Bilinear Pairing is a set of groups G_1, G_2, G_T with generators g_1, g_2, g_T and a function[1]

$$e : G_1 \times G_2 \to G_T$$

such that

$$e(g_1^a, g_2^b) = g_T^{ab}$$

Using this, we can verify without knowing secrets. Note that it's crucial that the CRS is generated correctly. This is usually done using MPC (multi-party computation) in a *key ceremony*[2]

§1.2 Secure Multi-Party Computation

§1.2.1 2-Party Computation

We've seen this before, but it is whensome parties want to compute the output of some function on their individual inputs, without revealing their own inputs.

Example 1.4

Alice and Bob just returned from a date, and want to figure out if they each want a second date. Alice has some choice bit x and Bob some choice bit y. They want to jointly compute $f(x, y) = x \wedge y$.

Example 1.5

Alice and Bob want to compare riches (who is richer?). They compute

$$f(x, y) = \begin{cases} 0 & \text{if } x > y \\ 1 & \text{otherwise} \end{cases}$$

Example 1.6

Alice and Bob meet for the first time and want to see if they have friends in common. They have sets of friends X, Y, and compute

$$f(X, Y) = X \cap Y.$$

There are variants of this which only give cardinality of $X \cap Y$, etc.

[1] The target group should be different than the initial groups, but the first two groups can be the same. If all groups are the same then DDH is made easy.

[2] Zcash, at one point, had a bad vulnerability which exposed CRS privates.

In general, this is when two parties have inputs x, y and want to compute some function $f(x, y)$ on them.

Use cases include:

- Password breach alert (Chrome/Firefox/Azure/iOS Keychain) runs a set intersection on your passwords and server leaked passwords.
- Privacy-preserving contact tracing for COVID-19 (Apple and Google). We want to know if we have contact but not who had contact with.
- Ads conversion measurements/personalized advertising (Google/Meta). We want to match conversions without either party knowing who converted.

§1.2.2 Multiple Parties!

The general case of this is Secure Multi-Party Computation (MPC)

This is when we have some parties P_i and want to compute input on $f(x_i, \cdots, x_n)$.

Here are some applications:

- A bunch missed
- Federated learning (used in Google Keyboard Search Suggestion). We want to run machine learning, federated amongst multiple devices. However, we don't want to leak the actual training data from users.
- Auctions (Danish sugar beet auction). Nobody should reveal their bid in the clear.
- Also deployed in Boston area to analyze the wage gap between genders without revealing the individual salaries.

Some applications are still in the works:

- Study/Analysis on Medical Data. Every institution has limited data, but they cannot openly share that data due to regulations. How could they jointly do analysis on this data without revealing the data.
- Fraud Detection (banks). Users might have cards at multiple banks, they want to jointly detect fraud but do not want to share their transactions.

When we normally talk about cryptography, we talk about 'slowing down' the system (crypto makes everything slower). In the case of MPC, though, we've enabled new features that were not otherwise possible without these tools.

§1.3 Definition

Our setting is that we have n parties $P_1, \ldots, P_n$ with private inputs $x_1, \ldots, x_n$. They want to jointly compute $f(x_1, \ldots, x_n)$.

In terms of communication infrastructure: we usually assume point-to-point channels between each pair (p_i, p_j). We know how to do this (key exchange, authenticated encryption, etc). Sometimes, we also assume a reliable broadcast channel where every other party gets information.

There is a single adversary that can "corrupt" a subset of the parties (at most t).

What properties do we want out of this system? Here are some common security properties we might want:

Correctness. The function is computed correctly.

Privacy. Only the output is revealed.

Independence of Inputs. Parties cannot choose their inputs depending on others' inputs.

Also with security guarantees:

Security with Abort. The adversary may "abort" the protocol. This prevents honest parties from receiving the output. This is the weakest model.

Fairness. If one party receives the output, then all parties will receive the output.

Guaranteed Output Delivery (GOD): Honest parties *always* receive output. Even if adversarial parties leave, the honest parties will simply continue the protocol.

We also have some characterizations of adversaries:

- Allowed adversarial behavior:
  - Semi-honest (or passive/honest-but-curious): They follow the protocol description honestly, but they try to extract more information by inspecting the transcript. This is the weaker model.
  - Malicious/active: These adversaries can deviate arbitrarily from protocol description.
- Adversary's computing power:
  - Unbounded computing power: this gives us information-theoretic (IT) security.
  - PPT bounded: this gives us computational security.

If you're interested, you can look into the literature of how to define security for MPCs. The idea is similar to that of ZK proofs—everything an adversary can do (see the transcript) can be simulated by a simulator who only has the input and output.

§2 March 21, 2023

Survey results: generally that course is well paced, some mentioned too slow or too fast. Seems to be a healthy in-between.

We'll also be having a guest speaker! They are from Google and have implemented MPC in real life.

Last time, we mentioned Bilinear pairings. It was left unresolved whether the target group can be the same group as the domain groups. We should have them be different. Specifically, this allows us to do a *single* multiplication in the exponent. If they are all the same group, we can do arbitrary polynomials in the exponent, which is not desired.

To do an arbitrary n number of equations is called a multilinear map. There are no known secure constructions of which.

§2.1 Secure Multi-Party Computation, *continued*

To quickly recap, a two-party computation is a computation where two parties want to jointly compute a function $f(x, y)$ on their private inputs x, y—but they do not reveal to each what their inputs are.

In the multi-party case, there will be n parties $P_1, P_2, \ldots, P_n$ with inputs $x_1, x_2, \ldots, x_n$ wishing to jointly compute $f(x_1, x_2, \ldots, x_n)$. We generally assume there are secure point-to-point channels, but some models assume broadcast channels. A single adversary can "corrupt" a subset of the parties, say t.

Here are properties we wish to attain in our protocol:

Correctness. The function is computed correctly.

Privacy. Only the output is revealed.

Independence of Inputs. Parties cannot choose their inputs depending on others' inputs.

Also with security guarantees:

Security with Abort. The adversary may "abort" the protocol. This prevents honest parties from receiving the output. This is the weakest model.

Fairness. If one party receives the output, then all parties will receive the output.

Guaranteed Output Delivery (GOD): Honest parties *always* receive output. Even if adversarial parties leave, the honest parties will simply continue the protocol.

§2.1.1 Feasibility Results

In the computational security setting, if we have a fundamental building block, a semi-honest oblivious transfer (OT), we can get semi-honest MPC for any function $t < n$. At a high level, using zero-knowledge proofs to enforce correctness of the protocol, we can convert any semi-honest MPC into a malicious MPC.

In terms of information-theoretic (IT) security. We can also get semi-honest and malicious MPC for any function with $t < \frac{n}{2}$. We call this an honest majority. This is a necessary bound, we cannot do any better than this.

§2.2 Oblivious Transfer

Definition 2.1 (Oblivious Transfer)

An oblivious transfer is a protocol in which a sender, with messages $m_0, m_1 \in \{0,1\}^l$ gives a choice to the receiver to receive either m_0, m_1.

Given a choice bit from the receiver $b \in \{0,1\}$, the receiver gets m_b and the sender also gets no information about the messeage transferred.

We'll learn about constructions of OT later, but we black-box its implementation until later.

Using a semi-honest OT, we can use Yao's Garbled Circuit to construct semi-honest 2PC for any function. We can also use the GMW compiler to compile this into a semi-honest MPC for any function. We'll focus on the first approach in this lecture, but we'll learn GMW in the following lectures.

§2.3 Yao's Garbled Circuit

Example 2.2 (Private Dating/AND Gate)

Alice and Bob want to figure out whether they want to go on a second date. Alice has single bit $x \in \{0,1\}$, and Bob also has single bit $y \in \{0,1\}$.

They want to compute a single AND gate.

Alice will *garble* circuit wires by generating some random l_0, l_1 for each wire corresponding to each bit possibility. We call these *labels*.

For each AND gate[3], she'll generate 4 ciphertexts,

$$\begin{aligned}&\mathsf{Enc}_{\alpha_0}(\mathsf{Enc}_{\beta_0}(0))\\&\mathsf{Enc}_{\alpha_0}(\mathsf{Enc}_{\beta_1}(0))\\&\mathsf{Enc}_{\alpha_1}(\mathsf{Enc}_{\beta_0}(0))\\&\mathsf{Enc}_{\alpha_1}(\mathsf{Enc}_{\beta_1}(1))\end{aligned}$$

If we have some α_a, β_b, then we can decrypt $\mathsf{Enc}_{\alpha_a}(\mathsf{Enc}_{\beta_b}(\cdots))$ and all other ciphertexts will look like garbage (we gain no information). This is to say, we can only decrypt the ciphertext of the keys we know. The overarching idea is that we'll only know the right labels for our inputs.

Alice will send the circuit (the 4 encryptions) as well as the input label for x, α_a. Bob now needs to get the label correponding to his input wire, β_0, β_1. We can perform an oblivious transfer!

Bob has a choice bit and gets one of β_0, β_1 without Alice knowing his choice bit. Having attained β_b, Bob will try the encryption on all 4 ciphertexts with α_a, β_b, and sees which output is valid and returns that.

For Alice to learn this output, Bob will send the output back to Alice. In the semi-honest setting, Bob will honestly send the result back to Alice. In the malicious case, we might require Bob to provide some zero-knowledge proof in the end to prove that their plaintext result came from their circuit.

We'll generatlize this single-gate computation for arbitrary functions. We'll represent any arbitrary function as a boolean circuit consisting of only AND and XOR gates[4].

Every wire gets two labels, corresponding to a 0 bit or 1 bit. Each label is $\xleftarrow{\$} \{0,1\}^\lambda$. For each gate, we construct a 'mini' garbled table, where the encrypted message is the output 0 or 1 labels[5,6]. We vary the encryptions based on the gate we're trying to implement.

Using the garbled circuit, we can construct an arbitrary 2PC. We call the party who generates the circuit the 'garbler', and the other party the 'evaluator'.

Alice garbles the circuit, and sends it to Bob. Alice can easily send her own labels. For the labels corresponding to Bob's input, we run oblivious transfer for each input wire to get Bob's input bits without Alice knowing.

In the final output, we can encrypt plaintext $0, 1$. The other way is for Alice to send the final

[3] We can change the values depending on the different logic gates.

[4] Recall that any boolean circuit can be represented using only AND and XOR gates.

[5] *How will we know which is garbage?* Naïvely, we could just try every label. However, this is an exponential blowup for every gate we run the labels through. The solution is to attach a bitstring 'tag' (could just be a string of 0s) that indicates whether a decryption is indeed a label.

[6] One more subtle thing we should take care of! We show our ciphertexts in order of $00, 01, 10, 11$. This reveals information! We should take care to shuffle the ciphertexts everywhere.

random labels to Bob along with their corresponding bits.

§2.3.1 Optimizations

There are some optimizations we can make:

Point-and-Permute. For each wire, we'll randomly sample signal bits $\sigma_\alpha, \sigma_\beta$, and flip it for the other input. (Note that this doesn't reveal anything about α, β). In the circuit, we can indicate using the signal bit which ciphertext to decrypt.

We reduce Bob's computation complexity by at least a constant of 4, and saves communication complexity by half (we don't need to expand our garbled circuit size anymore).

Row Reduction. In this construction, there are 4 ciphertexts per gate. We can just hash the labels and XOR with the corresponding output label (this is not CPA-secure, but that is fine). From the 4 ciphertexts, we can set γ_0 to exactly the hash $H(\alpha_0||\beta_0)$[7]. This is compatible with point-and-permute. We hide every row, which is fine. This gives us a $\frac{3}{4}$ space decrease.

Free XOR. Sample a global $\Delta \xleftarrow{\$} \{0,1\}^\lambda$. Every pair of labels differ by Δ. That is,

$$\begin{aligned}\alpha_1 &:= \alpha_0 \oplus \Delta \\ \beta_1 &:= \beta_0 \oplus \Delta \\ \gamma_1 &:= \gamma_0 \oplus \Delta\end{aligned}$$

and $\gamma_0 = \alpha_0 \oplus \beta_0$. To compute the output label, you just perform the XOR plainly.

This is to say, XOR is free. We don't need to send labels and Bob doesn't need to encrypt/decrypt.

We can also use *half-gates* which give us 2λ bits per AND gate + free XOR. A recent development, *slicing-and-dicing*, gives us around $\sim 1.5\lambda$ bits per AND gate + free XOR.

[7] Not really the $0, 0$ labels, but they can correspond to the signal bits.