**CS 294 – Secure Computation** February 16 and 18, 2016

# Lecture 9 and 10: Malicious Security - GMW Compiler and Cut and Choose, OT Extension

*Instructor: Sanjam Garg* *Scribe: Alex Irpan*

## 1 Overview

Garbled circuits let us securely compute functions in the semi-honest setting, but fail in the malicious setting. The GMW protocol gives information theoretic security if 1/2 of the parties are honest in the semi-honest case, and if 2/3 of the parties are honest in the malicious case. Here, we describe ways to extend garbled circuits to the malicious setting.
The two main approaches are the GMW compiler and the Cut and Choose method.

## 2 GMW Compiler

The GMW compiler is a general approach for transforming any protocol secure against semi-honest adversaries into one secure against malicious adversaries.
The core idea is that parties run the semi-honest protocol, along with a zero knowledge proof that they are running the protocol correctly. Here, we show the case for just two parties, but similar ideas extend to the multiparty case.
Let $\pi$ be a semi-honest protocol. We can treat protocols as functions

$$\pi(i, x_i, r_i, \text{transcript}_i)$$

where $i$ is the party index, $x_i$ is the $i$th party's input, $r_i$ is the $i$th party's randomness, and $\text{transcript}_i$ is a list of all messages the $i$th party has seen. Note $\pi$ is deterministic since randomness $r_i$ is passed as an argument.
At the start of the protocol, each party sends a commitment of its input and its random tape to the other. Let $t_1$ be the first message in the transcript, $t_2$ be the second, $\pi$ runs in polynomial time, so the decision problem

$$t_j \text{ is consistent with } \pi \text{ and the transcript } \{t_1, \ldots, t_{j-1}\} \text{ given the commitment to } x_i \text{ and } r_i$$

is in NP, where the witness is the key opening the commitment. Thus, there is a zero-knowledge proof of this with computational security.
Whenever a party sends $t_j$, it also does a zero knowledge proof that it is following protocol. By running this for every message, neither party learns any more information than the semi-honest case, malicious parties cannot cheat because of soundness.
These are the core ideas, but to make this fully secure we need to fix a few more things.

- In semi-honest protocols, we assume parties are honest in generating uniformly random bits. To make the protocol secure we must "force" malicious parties to use random bits.
- At the same time, each party's randomness $r_i$ needs to be kept secret.

This can be fixed with a simple coin-flipping protocol. Party $P_1$ generates $r_1, r_2'$. $P_2$ generates $r_2, r_1'$. They send commitments $Com(r_1), Com(r_2)$. Then, $P_1$ sends $r_2'$, $P_2$ sends $r_1'$, and going forward $P_i$ uses $r_i \oplus r_i'$ for their random coins.

## 3 Cut and Choose

One downside of the GMW compiler is all the zero knowledge proofs used. These proofs are very expensive because they involve reductions to NP-Complete problems like 3-Coloring or Hamiltonian Cycle, which could introduce large polynomial factors.
The Cut and Choose method is a way to achieve malicious security more efficiently. Again, these ideas can be extended to the multiparty setting, but for now we consider just the two party case.
Start with the semi-honest garbled circuit protocol. From here, we use $S$ to denote sender or party 1, and $R$ to denote receiver or party 2. $S$ has input $x$, and $R$ has input $y$.
To motivate the construction, we first consider the attacks a malicious sender could do against an honest receiver. $S$ sends two types of messages.

- The garbled circuit $G(C)$.
- The labels for $x$ (sent directly) and $y$ (sent through OT.)

Address the circuit message first. $S$ could send an arbitrary circuit instead of the circuit for $f(x, y)$. To handle this, the cut and choose method has $S$ send several garbled circuits instead of just one. $R$ then challenges $S$ to prove some of the circuits are valid garblings. $R$ will choose the challenge randomly. If all the challenges are valid, $R$ has good reason to believe the remaining circuits are valid as well.
Here is a first attempt at this protocol.

- $S$ constructs $s$ garbled circuits $\tilde{C}_1, \ldots, \tilde{C}_s$, where $s$ is a security parameter. $S$ sends all these circuits to $R$.
- $R$ samples $i \leftarrow [s]$, and asks $S$ to prove every circuit except $\tilde{C}_i$ is valid.
- $S$ sends all randomness used to generate those circuits.
- $R$ verifies all these garbled circuits are valid. (A garbled circuit is valid if every logic gate is garbled correctly and the decrypted logic gates match the ungarbled circuit $C$.)
- If the circuits are valid, $S$ and $R$ proceed with the garbled circuit protocol for the unchecked circuit $\tilde{C}_i$.

However, this protocol does not work, since $S$ can cheat with probability $1/s$ by sending exactly one invalid circuit. The fix is simple - $R$ will check a random subset of circuits instead of all but one, then take the majority output of all unchecked circuits. Intuitively, $S$ needs to change around half of the circuits to take over the majority output. Since each circuit is checked with probability $1/2$, $S$ can only cheat the process with probability $2^{-s/2}$.

- $S$ constructs $s$ garbled circuits $\tilde{C}_1, \ldots, \tilde{C}_s$, where $s$ is a security parameter. $S$ sends all these circuits to $R$.

- $R$ samples $r \leftarrow \{0,1\}^s$, and asks $S$ to open all circuits where $r_j = 1$.
- For every such $j$, $S$ sends the randomness for $\tilde{C}_j$.
- $R$ verifies each $\tilde{C}_j$ is valid.
- If they are, $S$ and $R$ proceed with the garbled circuit protocol for all unchecked circuits.
- $R$ evaluates the circuits and returns the majority output.

This finishes the informal handling of circuit attacks, leaving attacks on the labels for inputs $x$ and $y$.
This case further splits into subcases.

- $S$ could be inconsistent about the labels for $x$ and $y$, forcing $R$ to evaluate $f(x,y)$ on multiple input pairs $(x,y)$. Although $R$ sends only the majority output, $S$ can learn something even when the circuit outputs are unanimous. (For example, let $f(x,y) = 1$ when $x < y$. $S$ can send a different $x$ for each circuit, and will learn if a majority of the $x$s are less than $y$.)
- More subtly, $S$ can send consistently invalid labels to learn something depending on whether $R$ aborts or not. (For example, in the OTs for $R$'s input bit $y_1$, $S$ offers an invalid label for 0 and a valid label for 1. Then $R$ aborts if and only if $y_1 = 1$.

In general, $S$'s behavior should depend only on itself, and not on the actions of any honest parties. To fix this, we first amplify $C$ to take more input bits. For a $C$ taking $n$ bits of input, create a $C'$ which takes $sn$ bits, where $s$ is a security parameter. $C'$ is defined as

$$C'(z_1^1, z_1^2, \ldots, z_1^s, z_2^1, \ldots, z_2^s, \ldots, z_n^1, \ldots, z_n^s) = C(\oplus_{i=1}^s z_1^i, \oplus_{i=1}^s z_2^i, \ldots, \oplus_{i=1}^s z_n^i)$$

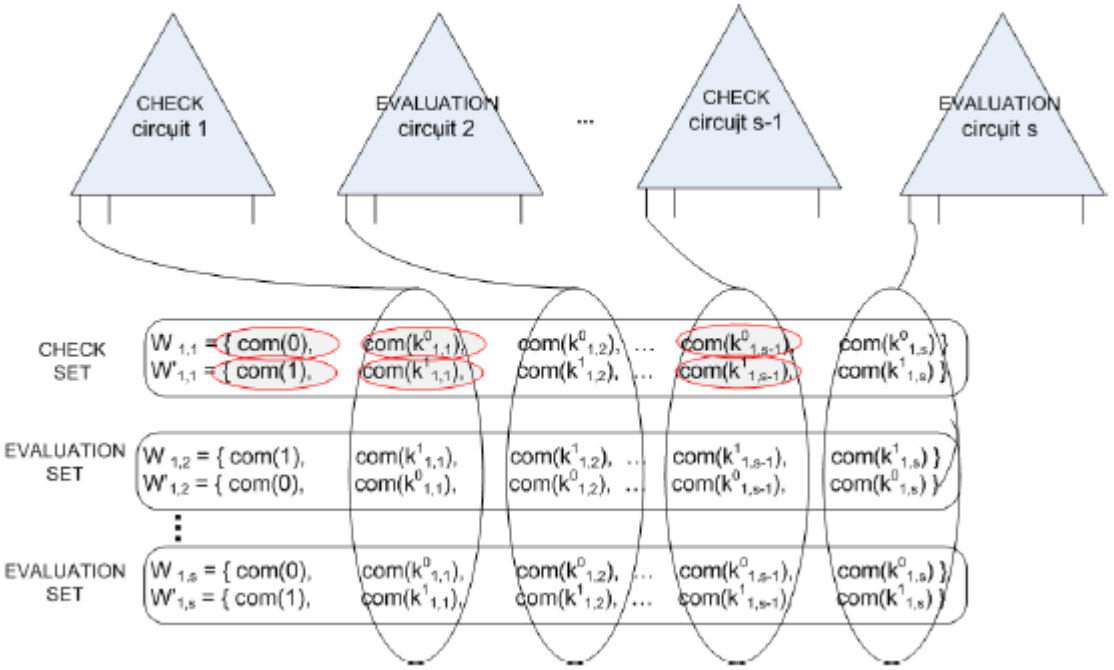
In a loose sense, this connects the difficulty of cheating on a single input bit to the security parameter $s$.
Next, $S$ will create commitments to every possible bit for each input wire. Note we are sending $s$ circuits of $sn$ inputs each, giving $O(s^2 n)$ commitments total.
Let $k_{i,j}^b$ denote the key for wire $i$, from circuit $j$, representing bit $b$. The commitments are organized into sets $\{w_{i,j}\}$ and $\{w'_{i,j}\}$. First, $S$ samples bits $b_j$. Then,

$$w_{i,j} = \{Com(b_j), Com(k_{i,1}^{b_j}), Com(k_{i,2}^{b_j}), \ldots, Com(k_{i,s}^{b_j})$$

$$w'_{i,j} = \{Com(1-b_j), Com(k_{i,1}^{1-b_j}), Com(k_{i,2}^{1-b_j}), \ldots, Com(k_{i,s}^{1-b_j})$$

where $i \in [sn], j \in [s]$. Each $w_{i,j}$ contains all the 0 keys or 1 keys, with the first commitment indicating which bit it is. $w'_{i,j}$ then contains the keys for the opposite bit. Intuitively, these sets will guarantee the consistency and validity of the input bits.
Note $\{w_{i,j}, w'_{i,j}\}$ is the same no matter what $j$ is. There are $s$ copies of each commitment set, but whether $w$ corresponds to 0 or 1 is random.
The idea is that by now, $R$ knows the secret keys for every checked circuit from the previous challenge. $R$ requests $S$ to open commitments for the checked circuits, to verify each $w_{i,j}$ is constructed correctly.
We apply the same cut and choose idea along two axes: to show a single commitment set is constructed properly, and to show all the commitment sets are constructed properly.
Here is the next modification of the protocol.

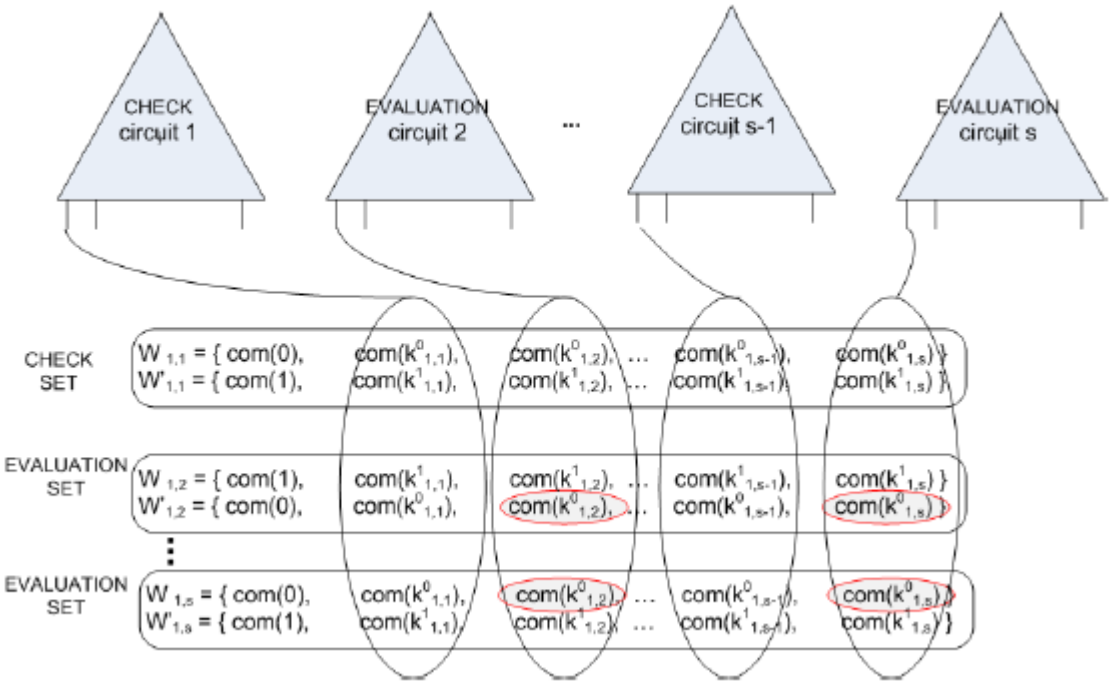
- $S$ constructs $s$ garbled circuits, sending all of them to $R$.
- $S$ samples $b_j$, and constructs the matrix $\{w_{i,j}, w'_{i,j}\}$ as described earlier. $S$ sends the entire commitment matrix.
- $S$ and $R$ do OT to transfer the keys for $R$'s input $y$ on all circuits. Recall the circuit is modified to XOR $s$ bits for each bit $y_i$. When $R$ requests keys for $y_i$, $R$ sends $s-1$ random bits to the OT, then sends the matching last bit such that everything XORs to $y_i$.
- $R$ samples two random strings, $r$ and $r'$, sending them to $S$. The first, $r$, tells $S$ which circuits to $R$ wants to check. The second, $r'$, gives which commitment sets $R$ wants to check.
- For every checked commitment set, $R$ opens the indicator commitment and the entries for the checked circuits. Notably, $R$ **does not** open the entire row.




$R$ now verifies the opened commitments are valid. The opened commitments are valid if they are consistent with the indicator bit. Verifying a random subset of entries lets $R$ trust that entire row is valid, and verifying a random subset of rows lets $R$ trust the entire matrix is valid.
Note the opened commitments on the checked circuits include the input wires for $y$. So, $R$ can verify the keys for $y$ match the requested bit, and this doubles as a consistency check for $y$.
If all verification passes, the protocol continues.

- $S$ sends keys for $x$ on all unchecked circuits.
- However, before running the unchecked circuits, $R$ asks $S$ to prove $x$ is consistent. For every unchecked commitment set, $S$ opens commitments from $w_{i,j}$ or $w'_{i,j}$, based on whether $x_i = 0$ or 1. In those sets, $S$ opens only the entries for unchecked circuits.

- If the revealed keys match the keys $R$ received, $R$ will believe $x$ is consistent. $R$ can now run the circuits and return the majority output.




Now, we informally argue why this setup is secure. Suppose $S$ tried to make $x$ inconsistent or invalid. Of the $w_{i,j}, w'_{i,j}$ opened right before evaluation, every single one must contain a commitment to 0 and 1. If that row $\{w_{i,j}, w'_{i,j}\}$ was picked in the check step, and the checked circuits included the entries for those two conflicting commitments, $S$ would be caught.
Similarly to the argument for valid vs invalid circuits, if $S$ changes enough to make an input wire inconsistent, it is caught almost all the time.

### 3.1 Final Protocol Modifications

There are a few remaining details needed for the proof of security.
So far, we have assumed the receiver is honest. Suppose the receiver was malicious instead. The receiver controls the following messages

- The random challenge strings $r$ and $r'$.
- Which bit to ask for in the oblivious transfers
- What output to send back to the verifier.

To fix the first, we use a coin flipping protocol. For random string $r$, each party samples $r_i$, sends a commitment $Com(r_i)$, then opens their commitments and use $r = r_1 \oplus r_2$. A similar process is done on $r'$. The other two issues are simulatable given oracle access to the malicious receiver's behavior.

### 3.2 Simulators

These will only be described informally.

#### 3.2.1 Cheating Sender Simulator

As constructed, challenges $r, r'$ are uniformly distributed even when the sender is malicious. The simulator runs the protocol until $S$ opens the commitments as defined by $r$ and $r'$. The simulator then rewinds to right after $S$ sends the circuits, and runs the protocol with new $r$ and $r'$. On expectation, about 1/4 of the circuits are checked the first time and unchecked the second.
In the circuits checked the first time, $R$ learns the mapping from garbled keys to input bits. In circuits evaluated the second time, $R$ learns which keys are for $S$'s input. Combined, $R$ can extract $S$'s input and get output $f(x, y)$.

#### 3.2.2 Cheating Receiver Simulator

$S$ first extracts $R$'s input from the simulator for oblivious transfer. Let $z = f(x, y)$ be the correct output. The simulator generates the garbled circuits, sometimes. Each circuit is either a valid garbling or a circuit that always outputs $z$, with the simulator choosing which circuits are valid randomly. The simulator then influences the generation of $r$ such that every checked circuit is valid and every unchecked circuit always outputs $z$. (To do so, when creating $r$, the simulator runs until $R$ opens their committed randomness $r_2$, then rewinds and sends the appropriate $Com(r_1)$.) All validity checks will go through, and $R$ will always get the right output $z$.

## 4 Oblivious Transfer Extension

In practice, oblivious transfer is the bottleneck for secure computation protocols. It turns out we can use a small number of OTs to transfer many messages. An OT of $k$ messages can be extended to $n = poly(k)$ messages.
This is done by constructing an appropriate circuit, then running a secure computation protocol on that circuit. Party $P_1$ has $m_1^0, m_1^1, \ldots, m_n^0, m_n^1$, and $P_2$ has $b_1, \ldots, b_n$.

- $P_1$ generates $2n$ random strings $r_1^0, r_1^1, \ldots, r_n^0, r_n^1$.
- $P_2$ samples $x \leftarrow \{0,1\}^k$. Let $g : \{0,1\}^k \to \{0,1\}^n$ be a PRG.

$P_1$ constructs a circuit $C(x_1 x_2 \ldots x_k)$ defined as

> Input $\{r_i^b\}, x$.
> $y \leftarrow g(x)$.
> Return $\perp$ to $P_1$. Return $\{r_i^{y_i}\}$ to $P_2$.

Evaluate this circuit securely, where $P_1$ gives $\{r_i^b\}$ and $P_2$ gives $x$. Now, $P_2$ knows $y_i, r_i^{y_i}$, and $P_1$ does not know what randomness $P_2$ requested.
Proceed with the final message transfer.

- For every $i$, let $c_i := b_i \oplus y_i$. $P_2$ sends all $c_i$ to $P_1$.

- $P_1$ sends $m_i^0 \oplus r_i^{c_i}, m_i^1 \oplus r_i^{1-c_i}$.
- $P_2$ XORs appropriately to get $m_i^{b_i}$.

## References


[1] Lindell, Yehuda. "Foundations of Cryptography 89-856." Chapter 13. (2010).

[2] Lindell, Yehuda, and Benny Pinkas. "An efficient protocol for secure two-party computation in the presence of malicious adversaries." Advances in Cryptology-EUROCRYPT 2007. Springer Berlin Heidelberg, 2007. 52-78.

[3] Ishai, Yuval, et al. "Extending oblivious transfers efficiently." Advances in Cryptology-CRYPTO 2003. Springer Berlin Heidelberg, 2003. 145-161.