Malicious Security and the GMW Compiler

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2025

Last time, we introduced a definition for (standalone) malicious security of protocols. Recall that, as with semi-honest security, this definition is based on a comparison between the real-world protocol and a corresponding ideal-world protocol. This ideal-world protocol involves an *ideal functionality* or *trusted party*, who locally computes the protocol's desired input/output behavior.

The main difference between the semi-honest and malicious settings is that in the malicious setting we assume the real-world adversary $\mathcal{A}$ is an *arbitrary computation* (often with the single restriction that $\mathcal{A}$ must run in polynomial time). Security of a protocol is shown by constructing an *ideal-world* adversary—a simulator $\mathcal{S}$—that participates in the ideal-world protocol and achieves the same "effect" as the real-world adversary. We capture the notion of "effect" by examining the joint outputs of all parties in (1) the real world and (2) the ideal world. If those two quantities are indistinguishable, we say our protocol is secure.

Namely, as cryptographers it is our task to show that for any real-world adversary $\mathcal{A}$, such an ideal-world adversary $\mathcal{S}$ exists. If we can show this, then it means that any "effect" caused by $\mathcal{A}$ can also be caused by $\mathcal{S}$ in the ideal world. Thus, the space of "effects" that $\mathcal{A}$ can cause is *no better than what an adversary could achieve in the extremely limiting ideal world protocol.* In other words, any such effect is, by definition, not an attack.

Recall that, unfortunately, our natural ideal functionalities from the semi-honest setting are often *too strong* for the malicious model. In particular, we had to settle for a weaker notions of *security with abort*, where $\mathcal{S}$ can stop the protocol at will, capturing the natural ability of real-world $\mathcal{A}$ to simply stop responding. Our ideal functionality with abort was as follows:

- Each party P_i sends its input x_i to the functionality (trusted party) F.
- F computes $(y_0, y_1) = f(x_0, x_1)$.
- Let party i be the corrupted party. F sends y_i to the adversary.
- The adversary now has two choices:
  - It can send `continue` to F. In this case, F sends y_{1-i} to the honest party, and the honest party outputs that quantity.
  - It can send `abort` to F. In this case, F terminates, and the honest party outputs $\perp$.

We also discussed that while this weakening of functionalities is quite severe, functionalities with abort still provide quite strong security guarantees.

Proof of the Coin Flip Protocol

Last time, we introduced a simple problem for which we would like to construct a maliciously-secure protocol: flipping a shared coin between two parties. That is, (1) each party sends $\perp$ to the functionality F, (2) the functionality F flips a coin $x \in_\$ \{0, 1\}$, (3) F sends x to the adversary, (4) the adversary may abort the protocol, and (5) if the protocol is not aborted, F sends x to the honest party.

Our natural attempts at this failed, until we invoked a cryptographic *commitment scheme* (with computational hiding and perfect binding). Our protocol running between A and B proceeded as follows:

- A and B respectively uniformly sample bits $a, b \in_{\$} \{0,1\}$.
- A generates a commitment $c = \texttt{Commit}(a, r)$ and sends c to B.
- B sends b to A.
- A sends a, r to B, and B checks $c = \texttt{Commit}(a, r)$. If not, he aborts.
- Parties locally compute and output $a \oplus b$.

We argued informally that this protocol is maliciously secure.

To get some practice with malicious security, let's prove security formally; note that [Lin16] also provides a rigorous proof of the security of this protocol. Recall that to prove security, we need to construct a simulator for both parties, and we need to argue indistinguishability.

Simulator for B. Let's start by simulating a malicious B. This means our task is to construct a simulator $\mathcal{S}_B$ that has the same "effect" as arbitrary program B, but where $\mathcal{S}_B$ interacts only with the ideal functionality. Recall that our main tool for doing this is the fact that B is assumed to be a program, and hence $\mathcal{S}_B$ can call B as a subroutine.

Let's start trying to construct $\mathcal{S}_B$. A natural first action is for $\mathcal{S}_B$ to simply send $\perp$ to the functionality F. Note that, in the background, the honest party A also sends $\perp$ to F. Thus, F now flips a coin $x \in_{\$} \{0,1\}$ and gives x to our simulator $\mathcal{S}_B$.

At this point, it may seem we are already almost done, but two crucial challenges remain:

- At this point in the ideal protocol, the simulator must decide whether protocol should abort or continue. Because our simulation must "look like" the real-world protocol execution, we cannot just make this decision arbitrarily. Instead, our decision must somehow be *consistent* with what the adversary B actually does.
- Our simulator has to output something consistent with what B outputs in the real world. Since B is an arbitrary program, we at this point have no idea what B will output.

Both challenges can be solved by the same main idea: let's use the fact that B is just a program and have our simulator invoke B as a subroutine. Our basic goal will be to try to "trick" B into thinking that it lives in the real-world, when, in fact, it only lives in the simulator's head. The idea for achieving this is that our simulator will "act like honest A". If we can do this convincingly (we can), it will lead to a proof.

So, we can start running a copy of B. It's important to observe that there are really only two possible things B can do at this point:

- B can exhibit some behavior that is obviously inconsistent with the real-world protocol. For instance, B might halt immediately, or it might start sending nonsense messages. Indeed, if B does *anything* other than simply await a message (remember, in the real-world protocol, B is suppose to wait for a commitment from A), then we can treat that behavior as B's desire to abort the protocol.
- B can wait for a message.

If B aborts, then $\mathcal{S}_B$ just sends `ABORT` to the functionality F and outputs whatever B outputs. A bit of reasoning shows that this is a good simulation. Indeed, in the real-world protocol, since B performs some nonsense action, A would have aborted and output $\perp$. Similarly, in the simulation, honest A outputs $\perp$, because it never is given output from F. At the same time, in both the real and ideal worlds, B outputs some arbitrary value based on having received no information whatsoever from A. Thus, in this case, the simulation is *perfect*. Indeed, in most cases, we will simply ignore such straightforward cases as the adversary immediately aborting before any messages are exchanged, because, as we see above, they are *trivial* to simulate.

We can now focus on the more interesting case where once we start B, it waits for a message. Notice that if B were in the real world, it would be sent a commitment to some random bit by A. Well, it is easy to arrange that B sees identically distributed information in the simulation. In particular, $\mathcal{S}_B$ (1) flips a coin $a \in_\$ \{0,1\}$, (2) generates a random commitment $c = \texttt{Commit}(a;r)$, and (3) sends c to B. Again, this is *identically distributed* to what B sees in the real world.

Again, there are now two possibilities: (1) B can abort the protocol, or (2) B can send back a single bit b, consistent with the real-world protocol specification. In the former case, we've already seen what to do: the simulator will send `ABORT` to the functionality and output whatever B outputs. In the latter case, our simulator has now seen three bits:

- It received the functionality output x from F.
- It sampled a bit a ("on behalf of the real-world honest party A") itself.
- It received a bit b from B.

Recall that in the real world protocol, the output bit x is computed as $a \oplus b$. Thus there are two scenarios worth considering:

- We go "lucky" (more on this shortly), and it happens to be the case that $a \oplus b = x$.
- We go unlucky: $a \oplus b \neq x$.

In we lucky, we are essentially done. We can complete the simulated protocol as follows:

- Send `CONTINUE` to the functionality F. This delivers x to the honest party A.
- Send the decommitment a, r to B. Notice that, again, this is exactly what B sees in the real world. At this point, we can output whatever B outputs.

But what do we do if we are unlucky? A first attempt would be to simply try to complete the simulated protocol anyway, as described above, but this would not work. The problem is while honest party A would output x in this simulation, adversarial B sees a transcript consistent with output $\neg x$. Thus, there is a distinguisher for certain adversaries B. Indeed, consider the adversarial B who happens to follow the protocol as described and simply outputs $a \oplus b = \neg x$. In the real world protocol, then, A and B both output $a \oplus b$; In the ideal world protocol A outputs x and B outputs $\neg x$. These two worlds are distinguishable.

The lesson here is that it *is typically important* to run a simulated protocol to its completion such that the simulated protocol output *is consistent* with the functionality output. So how shall we proceed?

The idea here is, again, to remember that B is just a program that we can run as a subroutine, so we can run B again. In the literature, this is sometimes referred to as *rewinding* B. We "rewind" B to the moment just before we send it a commitment. In particular, if we are unlucky, we can simply (1) spin up another copy of B, (2) sample a fresh bit $a \in_\$ \{0,1\}$ and send a commitment to B, and (3) receive a bit b from the new copy of B.

Exercise 1. *Earlier we already sampled $a \in_\$ \{0,1\}$. In this second attempt, can we simply commit to $\neg a$, rather than freshly sampling a? Why/why not?*

This second attempt can, again, be lucky or unlucky; if we are lucky, we complete the simulated protocol, as already described. Otherwise, we can try again, and so on.

A remaining question is this: can we continue to be unlucky forever? A naive first "proof" would be to claim that on each iteration, we are lucky with probability $1/2$, so observing an "unlucky streak" of length n occurs with probability at most 2^{-n}, which is negligible. While this intuition is close to the truth, it is not quite precise. The problem is that we have not accounted for B here! Perhaps B is always able to "evade" a lucky outcome. Namely, by looking at the commitment we send it, B deviously sends back a bit b that, e.g., causes $a \oplus b = 0$. Here, it is important that the bit x is *fixed* over all of our attempts. We cannot re-run the ideal functionality!

But how could B be choosing its bit b this way? A bit of reasoning shows that in order to do this, B *must be breaking the hiding property of our commitment scheme.* Indeed, the algorithm B would be able to be used to break commitment scheme hiding, which contradicts the assumption that `Commit` is, in fact, hiding. Therefore, our intuition is correct: it is only negligibly likely (in n) to observe a streak of n unlucky runs.

There is only one more subtle detail remaining: Our definition of malicious security requires our simulator to run polynomial time in all cases. Thus, although it is unlikely run for a long time, we must explain why our simulator does not run forever. The trick here is simply to set an upper bound on the number of runs, say λ. After a streak of λ unlucky runs, the simulator simply gives up. Namely, it sends `ABORT` to the functionality and outputs an arbitrary value, say `FAILED`. While the real-world B might not output `FAILED`, this case occurs with negligible probability.

Simulator for A. Next, let's construct a simulator $\mathcal{S}_A$ for A. Again, we will proceed, roughly, by "tricking" A into thinking it is in the real world protocol.

We start the simulation as follows:

- $\mathcal{S}_A$ will first send $\perp$ to the functionality F and receive a bit x.
- $\mathcal{S}_A$ spins up a copy of A; recall the first (non-trivial abort) action by A is to send a commitment c.

Now, from here B is expected to send a bit to A, so our simulator should do the same. But which bit shall we send? The main idea is this: Let's send both possible bits!

In particular, we'll make a copy of B (with the same input randomness, so it will send the same commitment c). Let's call one copy B_0 and the other B_1. We'll send 0 to B_0 and 1 to B_1. From here, B_0 (resp. B_1) has two options: abort the protocol, or decommit to a valid opening. There are three possibilities:

- Both B_0 and B_1 decommit. Note that they must both decommit to the *same* bit b, as otherwise B must be breaking the binding property. Now, we have two running protocol simulations, and one of them must be consistent with the functionality output x. In particular, $\mathcal{S}_B$ (1) sends `CONTINUE` to F, (2) computes $a = x \oplus b$, and (3) outputs whatever B_a outputs.
- Both B_0 and B_1 abort. In this case, $\mathcal{S}_B$ aborts as well and outputs whatever B_0 outputs.
- One copy of B aborts, and the other decommits. Here, we perform a case analysis. Suppose the decommiting version of B_i decommits to bit b. We conditionally proceed, depending on the quantity $i \oplus b$:
  - If it happens to be that $i \oplus b = x$, then we send `CONTINUE` to the functionality and output whatever B_i outputs.
  - Otherwise, i.e. $i \oplus b \neq x$, then we send `ABORT` to the functionality and output whatever B_{1-i} outputs.

It is not hard to see that this simulation is identically distributed to the real world. Indeed, this is roughly by construction.

Exercise 2. *Ensure you understand why party outputs from the above simulation are indistinguishable from their real world counterparts.*

Zero Knowledge Proofs; the GMW Compiler

Now that we understand the definition of malicious security, how can we actually achieve maliciously secure protocols for less contrived protocols? We'll start by understanding that *every* semi-honest protocol can be upgraded with malicious security by means of a *cryptographic compiler*. In particular, we'll use the famous GMW compiler [GMW87], which is based on so-called Zero Knowledge Proofs.

A Zero Knowledge Proof is a protocol involving two parties—a prover P, and a verifier V. Both parties have in mind some mathematical statement x, and P would like to convince V that the statement x is *true*. Formally, we typically think of x as a string, and we consider some *language* $\mathcal{L}$. P would convince V that x is in the language $\mathcal{L}$, $x \in \mathcal{L}$. To help P with this task, P has some extra information, called its *witness* w. Somehow w makes it clear that $x \in \mathcal{L}$. However, P would like to keep w secret.

In particular, a ZKP protocol Π should provide three properties:

- **Completeness.** If $x \in \mathcal{L}$, then V outputs 1 at the end of the protocol (with high probability).
- **Soundness.** If $x \notin \mathcal{L}$, then not even a malicious P can cause V to output 1 with high probability.
- **Zero Knowledge.** Not even a malicious V can learn anything about P's witness w.

In fact, (a slight strengthening of) these properties can be formalized as a special case of malicious security. In particular, consider a functionality where:

- P has input w and V has input $\perp$. Parties send their inputs to the functionality.
- The functionality checks if w is a valid witness. If so, it sends 1 to V, else it sends 0.

Now, malicious security in the context of this functionality implies (strengthenings of) the ZK properties:

- **Completeness:** The ideal functionality insists V gets 1 if the witness is valid. Thus, the real-world protocol must cause V to output 1 with overwhelming probability.
- **(Knowledge) Soundness:** A simulator for P must "extract" from P its witness w in order to send it to the functionality. Because behavior is idealized, the simulator has no way to "trick" the honest party, and so neither does the real-world adversarial prover, except with negligible probability.
- **Zero Knowledge:** The fact that V "learns nothing" about w is directly implied by V's simulator.

The community knows many protocols for achieving Zero Knowledge proofs for any language in the complexity class NP. Roughly, this means that it's effectively possible to "prove anything" in ZK. Indeed, we will see an MPC-based ZK protocol next time.

Now, supposing we have such a protocol, there is a relatively straightforward idea for upgrading a semi-honest protocol to a malicious one:

> Run the semi-honest protocol, with the following modification. Whenever a party sends a message m, they prove in Zero Knowledge to the recipient that m was constructed according to the rules of the semi-honest protocol. If such a proof fails, the recipient aborts.

In particular, as a first attempt, the sender can prove "this message m was constructed according to the semi-honest protocol, given that my input is x my randomness is r, and the messages I have received so far are τ." Namely, the sender proves his message is consistent with his *view*. As described, there is one serious problem with this proof: the statement depends on the prover's input x (which is supposed to be secret) and the prover's private randomness. Fixing this is relatively straightforward: Each party *commits* to their respective inputs, and the proof is with respect to the commitment (which is hiding), not the input itself. One final detail is that the sender might choose it's randomness maliciously. This can be fixed by using the above coin flipping protocol to sample randomness, and then forcing the sender to prove its random bits were chosen according to the output of the coin flipping protocol.

This indeed can be shown to work, in the sense that any protocol can be proved maliciously secure. Thus, we now have an important feasibility result: *Any* semi-honest protocol can be upgraded to a malicious protocol. Unfortunately, this GMW compiler can be quite expensive, since if the semi-honest protocol Π involves use of any cryptographic primitives, then the compiled protocol requires proving correct execution of that primitive, *a non-black-box use of cryptography.* Still, this feasibility gives evidence that we can be hopeful that efficient malicious protocols can exist.

Next Time

We will continue our discussion of Zero Knowledge proofs, and in particular show connections between our MPC protocols and ZK. We will specifically show how to construct a ZK proof scheme from an MPC scheme.

References

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[Lin16] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016.