Malicious Security

David Heath

Fall 2025

So far in this course, we have investigated protocols in the semi-honest model. In this model, we design protocols, and we assume that adversarial parties are *semi-honest* or *passive*: they follow the instructions of our protocols precisely as specified. While this is convenient for designing simple protocols, it is not necessarily realistic.

In more realistic scenarios, we can envision that an adversary might deviate from our protocol *arbitrarily*. That is, rather than running our protocol, the adversary will run some piece of code *of their choice*. Intuitively, we expect that a particularly cunning adversary will design their code carefully with details of our protocol in mind, in order to obtain a maximal advantage. For instance, the adversary might hope to learn as much as possible about honest party inputs. We refer to such entities as *malicious adversaries* or sometimes *active adversaries*.

Ideally, we would like to design our protocols such that they resist *any attempt* to cheat by a malicious adversary. If we can achieve this, our protocol has *malicious security*.

Informally, and perhaps obviously, achieving malicious security is significantly harder than achieving semi-honest security. The reason for this should be intuitive: in the semi-honest model, we need only provide security against a single, known piece of adversarial code (their part of the protocol as we specified it); In the malicious model, we need to provide security against *all possible pieces of code* (that run in polynomial time).

Amazingly, and as we will see in subsequent lectures, many of the techniques we have already seen can be upgraded to the malicious setting with (somewhat) small adjustments. Indeed, this "upgradability" is one of the reasons semi-honest security remains a definition worthy of study. Going forward, we will see two mechanisms by which to upgrade protocols:

- Protocol compilers: There exist some generic mechanisms by which to upgrade a semi-honest protocol to a malicious one *automatically*. These mechanisms, called compilers, take as input a semi-honest protocol and output a malicious one. Note that the compilation process can significantly degrade performance of the semi-honest protocol.
- Non-black-box upgrades: When one wishes for higher performance, often the more sensible approach is not to compile a malicious protocol, but rather to build a fresh protocol with malicious security. Still, from experience we have learned that such protocols are often easier to achieve by using a corresponding semi-honest secure protocol as a template.

Going forward, we will discuss several techniques for achieving malicious security. However, our main goal today is simply to understand what malicious security *means*, in a formal sense.

The Real/Ideal Paradigm, Revisited

Recall that our notion of semi-honest security is defined by comparing the adversary's view in the real-world protocol to its view in an ideal-world protocol involving a trusted third party (a functionality). These two worlds are compared by constructing a simulator that reorganizes the ideal-world view to make it "look like" the real-world view. A proof of security involves showing the output of the simulator is, indeed, indistinguishable from the real-world view.

Malicious security is defined in the same manner. Indeed, the definition is a simple (yet quite hard to understand, initially) modification of semi-honest security.

For simplicity, let's consider the two-party case; we'll look at a definition adjusted from [Lin16].

Definition 1 (Two-Party Malicious Security)**.** *Let f be a two-party functionality and Π be a two-party protocol that computes f. Π* ***securely computes f in the presence of a static malicious adversary*** *if for* every *(non-uniform, probabilistic polynomial time) adversary $\mathcal{A}$, there exists a (non-uniform, probabilistic polynomial time) adversary $\mathcal{S}$* (a simulator) *such that the following indistinguishability (in security parameter λ) holds for all $i \in \{0, 1\}$:*

$$\{\texttt{Ideal}(\lambda, i, x_{1-i}, f, \mathcal{S})\} \stackrel{c}{=} \texttt{Real}(\lambda, i, x_{1-i}, \Pi, \mathcal{A})$$

Here, i denotes which party is corrupted (P_0 or P_1), and x_{1-i} denotes the other (honest) party's input. $\texttt{Ideal}(\lambda, i, x_i, f, \mathcal{S})$ denotes the outputs of both parties in the ideal protocol when party i is corrupted (i.e., replaced by $\mathcal{S}$), and $\texttt{Real}(\lambda, i, x_i, \Pi, \mathcal{A})$ denotes the outputs of both parties in the real protocol when party i is corrupted (i.e., replaced by $\mathcal{A}$). Randomness in the ensembles is over the random coins sampled by the parties.

This definition requires some significant explanation.

Let's think about it by analogy to the semi-honest model. In the semi-honest model, we compare what the real-world adversary learns to what *can* be learned in the ideal world. If a simulator exists, then that is proof that the information in the real world is "no better" than the information in the ideal world; indeed, the ideal-world information looks the same. That must mean that the real-world adversary learns nothing in the real world, beyond what it learns in the ideal world, which is precisely what we specify the adversary is allowed to learn.

The malicious model is analogous. In the real world, the adversary $\mathcal{A}$ may attempt various forms of attacks. We compare the space of possible attacks to what attacks can be performed in the ideal world. To show security, we (the cryptographers proving security) construct a new *ideal-world adversary*—the simulator. This simulator can perform "attacks" in the ideal world. However, note crucially that the space of attacks in the ideal world are *extremely limited*, since the ideal-world adversary *only* gets to interact with the ideal functionality. If a simulator exists, and indistinguishability holds, then that is proof that attacks the adversary attempts in the real world are "no better" than attacks that can be attempted in the ideal world. Indeed, the adversary's output and the effect on the honest party (its output) are *indistinguishable* across the two worlds. That must mean that the real-world adversary can perform no attacks in the real world, beyond those it can launch in the ideal world, which is precisely what we specify the adversary is allowed to do.

Remark 1 (Static security)**.** *Roughly, the above definition states that the adversary chooses which party to corrupt when the protocol starts. There exist much stronger of notions called* adaptive *security, whereby the adversary can corrupt parties as the protocol proceeds, perhaps even ultimately corrupting all parties. Such notions can be* much, much *harder to realize. We will not focus on them in this course.*

Ideal Functionalities with Abort

Recall that when we defined semi-honest security, we did so by giving a simple, specific ideal MPC functionality:

- Each party P_i sends its input x_i to the functionality (trusted third party) F.
- F computes $(y_0, ..., y_{p-1}) = f(x_0, ..., x_{p-1})$ and sends y_i to P_i.

Can we use this same ideal functionality the malicious setting? Unfortunately, the answer is often no. The reason is that this functionality is remarkably strong; it essentially states that the adversary has *no attack capabilities whatsoever*, beyond choosing its input.

Unfortunately, in many scenarios there is a simple attack that we cannot possibly hope to mitigate: the adversary "switches off its machine". Consider this: suppose that the real-world protocol Π is such that the adversary learns the protocol output just before an honest party would. A bit of reasoning shows that some event like can *necessarily* occur. Since messages are sent one at a time, some party *must* get output first, and we may as well assume the adversary is that party. Now suppose that as soon as that adversary learns its output, it turns off its machine, stopping the protocol. This necessarily leads to a certain kind of attack where the adversary learns the protocol output, but the honest party does not. The protocol is inherently *unfair*.

Remark 2. *The above discussion of fairness is a significant simplification of the adverary's capability in an arbitrary protocol. It is the case that the adversary can, in general, learn more than the honest party, but there are some (very limited) techniques for bounding this advantage. See [Cle86] for a careful proof of the inherent difficulty of fairness.*

Thus while we might like our simple functionality, it is often too strong. Our simple functionality requires that the protocol achieve so-called *guaranteed output delivery* (GOD). GOD can only be achieved in certain scenarios. For instance, generic MPC protocols can only achieve GOD when we assume a majority of the parties are honest (i.e., not corrupted by $\mathcal{A}$).

In general and in the setting where a majority of parties might be dishonest, we will need to resort to weaker notions of security. In particular, we will now introduce a change to the 2PC ideal functionality that we refer to as *security with abort*:

- Each party P_i sends its input x_i to the functionality (trusted third 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 $\perp$.

This change to the functionality probably seems a significant weakening of protocol guarantees, and indeed it is. Unfortunately, in many cases, the above functionality is essentially *the best we can hope for*, due to the fundamental ability of the adversary to simply stop participating in the protocol.

And, while our new ideal world is significantly weaker, it still provides many guarantees. For instance:

- The adversary cannot learn anything about the honest party's input, except by carefully choosing its own input and inferring what it can from the output of f.
- The adversary cannot cause the honest party to output anything inconsistent with an output of f; again, all it can do is carefully choose its own input.
- The adversary cannot choose its input depending on the honest party's input. Indeed, the ideal-world adversary must send its input to the functionality before it sees any messages.

Thus, while this ideal-world is a weakening as compared to GOD security, it is still quite strong.

Intuition of How to Prove Malicious Security

In our new security model, we are now tasked with producing an ideal-world adversary (a simulator) that "does whatever the real-world advesary does". How can we achieve such a notion?

The key idea is that our simulator's design can *depend on the behavior of the real-world adversary* $\mathcal{A}$. In particular, $\mathcal{A}$ is simply a piece of code, and as cryptographers we can design our simulator $\mathcal{S}$ in such a way

that it calls $\mathcal{A}$. When a protocol Π is indeed secure, we can interact with $\mathcal{A}$, and thereby extract insight into the "attack" $\mathcal{A}$ is attempting.

Roughly, such simulations end up involving $\mathcal{S}$ "tricking" $\mathcal{A}$ into thinking it is participating in the real-world protocol, when it is in fact not. Additionally, such simulations sometimes involve running $\mathcal{A}$ multiple times; it is fine to do this, as $\mathcal{A}$ is just a piece of code. Ultimately, if $\mathcal{S}$ can figure out $\mathcal{A}$'s intentions (e.g., its input to the protocol, whether or not it intends to abort), then $\mathcal{S}$ can properly participate in the ideal-world protocol.

An Insecure Protocol

To understand how malicious security works, it can be instructive to see examples of protocols that are *not* maliciously secure. Let us attempt a particularly simple functionality: The parties output a common uniform bit b. As a first attempt, let's try our simple semi-honest XOR protocol

- A and B respectively uniformly sample bits $x, y \in_\$ \{0,1\}$.
- A sends x to B.
- B sends y to A.
- A and B each locally compute and output $x \oplus y$.

This protocol is semi-honest secure, but it is does not achieve malicious security.

The way to see the problem is to consider what happens when B is corrupted. Notice that B is allowed to see x *before* it has to choose what y is. Thus, for instance, an adversarial B can always set the output to be 0, by simply echoing x back to A.

Notice that this behavior is *not possible* in the ideal world. In the ideal world, we force parties to choose inputs independently. Thus, in this protocol the real-world adversary can do something the corresponding ideal-world adversary cannot. The protocol is not secure.

We can make an attempt to fix the protocol by having A *encrypt* its first message:

- A and B respectively uniformly sample bits $x, y \in_\$ \{0,1\}$.
- A samples a random bit $r \in_\$ \{0,1\}$ and sends $c = (x \oplus r)$ to B.
- B sends y to A.
- A sends r to B.
- Parties locally compute and output $x \oplus y$.

It might seem that this helps, because now B must choose y before it learns x; indeed, r acts as a one-time pad, so from B's perspective, x can be anything at the time it must choose y. Unfortunately, the protocol is still insecure, because we can now consider what happens when A is corrupted. Indeed, we can reason that A can choose x at will, after it learns y. To do so, it just sends a random ciphertext c in the first round, and in the third round sends a bit that decrypts c to its chosen x.

However, this attempted fix gives a sense of what is needed. We want A to encrypt its message in such a way that A cannot "change" what was encrypted later on. A primitive that achieves such a notion is called a *commitment scheme*.

Commitment Schemes

A commitment scheme is the digital analog of a secure lock box. Intuitively, A can place a message into the lock box, use a key to lock it, then give the box to B. B cannot see what is inside the box; is it *hiding*. Later, A can give the key to B. Now, B can see the contents of the box, and he can be certain that those

contents did not "magically change" between the time Alice gave him the box and the time she gave him the key; the box is *binding*.

Formally, a commitment scheme is an algorithm `Commit` that takes as input (1) a message m and (2) randomness r. To commit, A (1) samples randomness r, (2) calls $c = \texttt{Commit}(m, r)$, and (3) sends c to B. To later open, A simply sends m, r to B, and he checks $\texttt{Commit}(m, r) = c$. If not, he can be certain A aborted. We will formally need a commitment scheme with *computational hiding* and *perfect binding*.

Definition 2 (Binding)**.** *A commitment scheme has* ***binding*** *is defined in terms of inputs m_0, m_1 and an adversary $\mathcal{A}$. Consider the following interaction:*

- *The adversary publishes a commitment c.*
- *The adversary is sent a bit $b \in \{0, 1\}$.*
- *The adversary attempts to open the commitment to m_b. In particular, it sends randomness r. The adversary* wins *if $\texttt{Commit}(r, m_b) = c$.*

The scheme is binding if no such $\mathcal{A}$ can win with probability greater than $\frac{1/2}{+}\texttt{negl}(\lambda)$. A scheme is statistically binding *if security holds against even computationally unbounded adversaries, and it is* perfectly binding *if the adversary wins with probability exactly $\frac{1}{2}$.*

Definition 3 (Hiding)**.** *A commitment scheme has* ***computational hiding*** *that for all inputs m_0, m_1, the following are indistinguishable:*

$$\{\ \texttt{Commit}(m_0, r) \quad \textit{where} \quad r \in_\$ R\ \} \stackrel{c}{=} \{\ \texttt{Commit}(m_1, r) \quad \textit{where} \quad r \in_\$ R\ \}$$

Here, R denotes the space of randomness for the scheme. Statistical hiding *and* perfect hiding *are defined in the natural manner.*

Exercise 1. *There is a simple commitment scheme from any pseudorandom generator that achieves statistical binding and perfect hiding. Read about this scheme (see* ***https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf****, chapter 3.12) and prove it secure.*

A secure protocol

Now, with our commitment scheme, we can build a maliciously-secure protocol:

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

Remark 3. *Since a malicious adversary can do* anything*, there are many opportunities for adversaries to perform nonsensical actions. For example, in our above protocol, the adversary might send a string that is clearly not a commitment. For example, the string is too short or too long. We treat these degenerate signals as an abort. I.e., we model it by having the adversary send* `abort`*, and the honest party obliges.*

Intuitively, the protocol is now secure: B can no longer cheat, because he must choose y after having seen only a *hiding commitment* to x. Breaking independence of inputs would require him to break security of the commitment scheme. And A can no longer cheat, because she can only open her commitment to x; successfully opening something else would require breaking the *binding* property of our scheme.

Next Time

We will proceed to show this protocol is secure by constructing simulators and arguing security. Then, we will move on to discussing more advanced malicious protocols. In particular, we will begin discussing Zero Knowledge proofs, and how they can be used to upgrade *any* semi-honest secure protocol to the malicious model.

References

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press, May 1986.

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