Adaptive versus Static Security in the UC Model*

Ivan Damgård and Jesper Buus Nielsen

Aarhus University

Abstract. We show that for certain class of unconditionally secure protocols and target functionalities, static security implies adaptive security in the UC model. Similar results were previously only known for models with weaker security and/or composition guarantees. The result is, for instance, applicable to a wide range of protocols based on secret sharing. It "explains" why an often used proof technique for such protocols works, namely where the simulator runs in its head a copy of the honest players using dummy inputs and generates a protocol execution by letting the dummy players interact with the adversary. When a new player P_i is corrupted, the simulator adjusts the state of its dummy copy of P_i to be consistent with the real inputs and outputs of P_i and gives the state to the adversary. Our result gives a characterization of the cases where this idea will work to prove adaptive security. As a special case, we use our framework to give the first proof of adaptive security of the seminal BGW protocol in the UC framework.

1 Introduction

When defining and proving security of cryptographic protocols we want to capture properties that would make our protocols applicable in real applications. Two aspects are particularly important in this respect. First, a protocol usually is a part of larger system and therefore we want a protocol to remain secure when composed, not only with itself, but also with an arbitrary environment. Second, a protocol must remain secure, even if some of the players are corrupted by an adversary. In a real scenario, one should expect that the choice of which players to attack is made while the protocol is running, i.e., we would like to have security against adaptive corruption rather than static, where the choice is made before the protocol starts.

Capturing these goals in a definition is notoriously a difficult task, and this may be the reason why general protocols for multiparty computation [13, 4, 10] were found a long time before we had generally accepted definitions of security for which composition results could be shown.

In 1991, Micali and Rogaway [15] as well as Beaver[3] put forward definitions. Like virtually all subsequent work, these definitions use simulation-based security: given only what the adversary is supposed to learn, it should be possible to simulate his view of the protocol. However, it was not until the work around 2000 of Canetti [6] (the universal composition (UC) framework) and independently Pfitzmann, Schunter and Waidner [17] (reactive simulation) that security under arbitrary concurrent composition could be expressed. A recent related, but different approach known as "constructive cryptography" was initiated recently by Maurer [14]. This framework is also simulation-based and gives security under composition, but is technically different from the UC model in several ways.

It turns out that achieving adaptive security under, e.g., the UC definition is highly non-trivial for protocols that are based on cryptographic assumptions (although the problem can be solved at some loss of efficiency using so-called non-committing encryption [8]).

* Supported by European Research Council Starting Grant 279447. Supported by Danish Council for Independent Research via DFF Starting Grant 10-081612. Supported by the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, and also by the CFEM research centre (supported by the Danish Strategic Research Council) within which part of this work was performed.

Since these complications are tightly linked to the use of encryption in the protocol, it was for a while believed in the folklore that for protocols that are information theoretically secure, static and adaptive security should be equivalent. This is not the case, however, there are natural examples of information theoretically secure protocols that are statically secure but not adaptively secure, as shown in [11].

In [7], a systematic study of the relation between static and adaptive security was conducted. This was limited to definitions allowing only sequential, rather than concurrent composition. They found that, in most cases, static and adaptive security are not equivalent. However, there was one important exception, namely that in the definition from [15] (called the MR definition in the following), static and adaptive security are equivalent.

Our contribution. It is natural to then ask what we can say about definitions that allow for concurrent composition, such as UC. In view of the example from [11], one should of course not expect adaptive to be equivalent to static in general; but it may be possible to identify a class of protocols where equivalence holds, and where therefore we can prove static security and get adaptive security only by verifying that the protocol is in this class.

One might perhaps hope that the positive result from [7] on the MR definition could help us, but this is not clear at all: the MR definition allows the simulator to be infinitely powerful, where a UC simulator must be polynomial time. It considers only secure function evaluation, where UC considers general reactive functionalities; and finally the MR definition requires a protocol to have a certain "committal round" where all the inputs become fixed, where the UC definition makes no such requirement.

In this paper, we borrow a high-level idea from the equivalence proof in [7], which can be loosely described as follows: to do adaptive simulation, we start by running the static simulator for the case where no player is corrupted. As soon as a corruption occurs, we try to "rescue the situation" such that we can continue running the static simulator having corrected for the fact that a new player has been corrupted. We continue in this way until the protocol halts.

Our technical contribution is to first identify the constraints on the static simulator and the target functionality that one needs to make this work in the UC model and second to resolve the difficulties arising from the differences between the MR and UC definitions. As a result, we show that for a certain class of unconditionally secure protocols and target functionalities, static security implies adaptive security in the UC model. The constraints we need on the static simulator and the target functionality are quite natural and allow the result to be applied, for instance, to a wide range of protocols for honest majority based on secret sharing, including the BGW protocol from [4]. The result also holds if the protocol uses one or more auxiliary functionalities, as long as they satisfy the same constraint. The result therefore also covers the on-line phase of several recent protocols in the pre-processing model [5, 16, 12].

To avoid confusion related to security of the BGW protocol, we want to clarify the relation between our result and the recent security proof for this protocol, given by Lindell and Asharov [2]. They prove static security and then notice that BGW satisfies the MR definition, which by the result from [7] implies adaptive security (in the MR definition). While this is true, it does not imply security in the UC model: first, as we mentioned, an MR simulator has unbounded computing time while a UC simulator must be polynomial time. Second, the equivalence result from [7] depends crucially on the simulator being unbounded. Therefore there is currently no proof that the BGW protocol is adaptively UC secure. However, using our result, such a proof can be derived from the proof of static security. We make an assumption on the structure of the circuit to be computed, namely that each output value is produced by a multiplication gate – this can easily be achieved by adding dummy multiplications by 1 if needed. This certainly simplifies the proof, but might in fact even be essential to get an efficient adaptive

simulator. It is so far open whether this is the case. However, we find find it intriguing that even for a well known protocol like BGW, that is generally believed to be "clearly" adaptively secure, a proof of this is a non-trivial step beyond static security.

From a more high-level point of view, our result "explains" why an often used proof technique for such protocols works, namely where the simulator runs in its head a copy of the honest players using dummy inputs and generates a protocol execution by letting the dummy players interact with the adversary. When a new player P_i is corrupted, the simulator patches the state of its dummy copy of P_i to be consistent with the real inputs and outputs of P_i and gives the state to the adversary. Our result gives a characterisation of the cases where this idea will work.

Since one of the constraints we impose on the static simulators is that one can *efficiently* patch from a static simulation of a small set of parties to a static simulation of a larger set of parties, our framework does not give adaptive security for free compared to current proof strategies. However, our framework abstracts current proof techniques and once and for all lifts all the technical details that are common for most proofs. We hope and believe that our result will make it easier to prove adaptive UC security, as it reduces the task to proving static security and checking whether the constraints we require are satisfied.

2 The UC Framework

In this section we sketch the UC framework and define some shorthand notation which we believe will make the upcoming proofs more clear.

In the framework from [6] the security of a protocol is defined by comparing its real-life execution to an ideal evaluation of its desired behavior. The protocol π is modeled by n interactive Turing Machines (ITMs), $\pi = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$, called the *parties*. In addition an *ideal functionality* is given. An ideal functionality is just an ITM. All parties can send messages to $\mathcal{R}$ and receive messages from $\mathcal{R}$, using perfectly secure channels. The input-output behavior of $\mathcal{R}$ models the communication resource available to the parties in the protocol, and can, e.g., model perfectly secure, synchronous communication or authenticated asynchronous communication, but can be arbitrarily complex. In the *execution* of π using communication resource $\mathcal{R}$ also an adversary $\mathcal{A}$ is present and an environment $\mathcal{Z}$ modeling the environment in which $\mathcal{A}$ is attacking the protocol. The environment gives inputs to honest parties, receives outputs from honest parties, and can communication with $\mathcal{A}$ at arbitrary points in the execution. The adversary can see and control the communication by interacting with $\mathcal{R}$.[1] The adversary can additionally corrupted parties adaptively. When a party is corrupted, the adversary learns the entire execution history of the corrupted party, including the random bits used, and will from the point of corruption send messages on behalf of the corrupted party. Both $\mathcal{A}$ and $\mathcal{Z}$ are PPT ITMs.

At the beginning of the protocol all parties, the communication resource, the adversary, and the environment is given as input the security parameter k and random bits. Furthermore the environment is given an auxiliary input z. At some point the environment stops activating with parties and outputs some bit. This bit is taken to be the output of the execution. We use $\textsc{Exec}_{\pi,\mathcal{R},\mathcal{A},\mathcal{Z}}(k,z)$ to denote the output of $\mathcal{Z}$ in the execution. We let $\textsc{Exec}_{\pi,\mathcal{R},\mathcal{A},\mathcal{Z}}$ denote the distribution ensemble $\{\textsc{Exec}_{\pi,\mathcal{R},\mathcal{A},\mathcal{Z}}\}_{k\in\mathbb{N},z\in\{0,1\}^*}$.

One particular adversary is the so-called *dummy adversary* $\mathcal{D}$. It simply works as a channel between $(\pi,\mathcal{R})$ and $\mathcal{Z}$. As examples, if $\mathcal{R}$ outputs a message m to $\mathcal{D}$, $\mathcal{D}$ simply outputs m

[1] The leakage seen and influence allowed by $\mathcal{A}$ is defined by the input-output behavior of $\mathcal{R}$. If $\mathcal{R}$ models only authenticated communication it would send the transmitted messages also to $\mathcal{A}$. If it models secure communication it would not. If it models asynchronous communication it could let $\mathcal{A}$ specify any delivery pattern, if it models synchronous communication it would impose restrictions on the delivery patterns $\mathcal{A}$ may specify.

to $\mathcal{Z}$, specifying that it is from the communication resource, and if $\mathcal{Z}$ instructs to corrupt P_i, $\mathcal{D}$ will do so, and return the obtained information to $\mathcal{Z}$. We use $\text{REAL}_{\pi,\mathcal{R},\mathcal{Z}}(k,z)$ to denote $\text{EXEC}_{\pi,\mathcal{R},\mathcal{D},\mathcal{Z}}(k,z)$.

Second an *ideal evaluation* is defined, which is just another protocol plus communication resource being attacked by an adversary in an environment. In the ideal evaluation again an ideal functionality F is present. However, now the input-output behavior of F is a specification of the desired input-output behavior of the protocol. Also present is an adversary $\mathcal{S}$ (a.k.a. the *simulator*), the environment $\mathcal{Z}$, and n so-called *dummy parties* $\mathsf{D}_1,\ldots,\mathsf{D}_n$ – all PPT ITMs. The only job of the dummy parties is to take inputs from the environment and send them to the ideal functionality and *vice versa*. We call $\delta = \{\mathsf{D}_1,\ldots,\mathsf{D}_n\}$ the *dummy protocol*. Again the leakage seen by the adversary and the influence that the adversary can except is defined by the input-output behavior of F, i.e., by which messages F sends to $\mathcal{S}$ and and how F responds to messages from $\mathcal{S}$. Note that δ executed with F as communication resource is a trivially secure protocol with the same input-output behavior as the ideal functionality F. For an environment $\mathcal{Z}$ we use $\text{EXEC}_{\delta,\mathsf{F},\mathcal{S},\mathcal{Z}}(k,z)$ to denote the output of $\mathcal{Z}$ after the execution. Since δ is a fixed protocol we can omit it in the notation. We let $\text{IDEAL}_{\mathsf{F},\mathcal{S},\mathcal{Z}}(k,z) = \text{EXEC}_{\delta,\mathsf{F},\mathcal{S},\mathcal{Z}}(k,z)$.

We recall the definition of UC security. It can be proven that it is sufficient to prove security against the dummy adversary, so we phrase the version where the adversary is fixed to be $\mathcal{D}$.

Definition 1 ([6]). *We say that* π securely realizes F *in the* $\mathcal{R}$*-hybrid model if there exists a PPT simulator* $\mathcal{S}$ *such that for all PPT environments* $\mathcal{Z}$ *we have that* $\text{IDEAL}_{\mathsf{F},\mathcal{S},\mathcal{Z}}$ *and* $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ *are computationally indistinguishable. We say that there is statistical security if* $\text{IDEAL}_{\mathsf{F},\mathcal{S},\mathcal{Z}}$ *and* $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ *are negligibly close for* all *environments* $\mathcal{Z}$*, i.e.,* $\mathcal{Z}$ *is not restricted to PPT. We say there is perfect security if* $\text{IDEAL}_{\mathsf{F},\mathcal{S},\mathcal{Z}} = \text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ *for* all *environments* $\mathcal{Z}$*.*

We use $\mathsf{A} \diamond \mathsf{B}$ to denote a system containing the two ITMs A and B and also use this notation for larger systems. In the UC framework two ITMs A and B in the same system of ITMs communicate by writing designated messages on the tapes of each other, specifying the message and the identity of the sender. We would like a more convenient terminology for this communication mechanism, so we will talk about A and B being equipped by incoming ports (inports) and outgoing ports (outports). A port is just a bit string $\mathtt{pn}$, naming the port, plus a direction. Ports are connected by identity of name and opposition of direction. I.e., if A has an outport $\mathtt{pn}$ and B has an identically named inport $\mathtt{pn}$ then we say that A can send messages to B on $\mathtt{pn}$. I.e., saying that A sends m on $\mathtt{pn}$, in $\mathsf{A} \diamond \mathsf{B}$, is equivalent to saying that A writes $(\mathtt{pn}, m)$ on a tape of B. Since ports are connected by names we clearly have that $\mathsf{A} \diamond \mathsf{B} = \mathsf{B} \diamond \mathsf{A}$.

Execution of an interactive system of ITMs works in a "sequentialized concurrent" way, where only one ITM is active at a time. The activation is passed from one ITM to the next when a message is sent to that ITM. Initially the environment is activated. Which ITMs can write and read on which tapes and how activation is passed is specified in great detail in the UC framework, but we will not need to address the particularities to prove our result.

When a system of ITMs is closed, i.e., there are no outports without an identically named inport, then it can be executed as described above, and we use the system also to denote the family of random variables describing its execution, i.e., $\delta \diamond \mathsf{F} \diamond \mathcal{S} \diamond \mathcal{Z} = \text{EXEC}_{\delta,\mathsf{F},\mathcal{S},\mathcal{Z}}$ and $\pi \diamond \mathcal{R} \diamond \mathcal{D} \diamond \mathcal{Z} = \text{EXEC}_{\pi,\mathcal{R},\mathcal{D},\mathcal{Z}}$.

If an interactive system has no protocol, then this will sometimes tacitly mean that the protocol is the dummy protocol. Equivalently, a missing adversary sometimes denotes the dummy adversary, i.e., $\mathsf{F} \diamond \mathcal{S} := \mathsf{F} \diamond \delta \diamond \mathcal{S}$ and $\pi \diamond \mathcal{R} := \pi \diamond \mathcal{R} \diamond \mathcal{D}$.

If two interactive systems are open, but would become closed by adding an environment to the system, then we compare them by comparing them in *all* environments, i.e., a missing environment designates *all* environments. Formally, if $\mathsf{F} \diamond \mathcal{S} \diamond \mathcal{Z} = \pi \diamond \mathcal{R} \diamond \mathcal{Z}$ for all environments

$\mathcal{Z}$, we write $\mathsf{F} \diamond \mathcal{S} \stackrel{\text{perf}}{\equiv} \pi \diamond \mathcal{R}$. If $\mathsf{F} \diamond \mathcal{S} \diamond \mathcal{Z}$ and $\pi \diamond \mathcal{R} \diamond \mathcal{Z}$ are negligibly close for all environments $\mathcal{Z}$, we write $\mathsf{F} \diamond \mathcal{S} \stackrel{\text{stat}}{\equiv} \pi \diamond \mathcal{R}$. If $\mathsf{F} \diamond \mathcal{S} \diamond \mathcal{Z}$ and $\pi \diamond \mathcal{R} \diamond \mathcal{Z}$ are negligibly close for all PPT environments $\mathcal{Z}$, we write $\mathsf{F} \diamond \mathcal{S} \stackrel{\text{comp}}{\equiv} \pi \diamond \mathcal{R}$. These notions can be refined by restricting the class of environments. For instance, we write $\mathsf{F} \diamond \mathcal{S} \stackrel{\text{comp}}{\equiv}_{\text{Env}} \pi \diamond \mathcal{R}$ to mean that $\mathsf{F} \diamond \mathcal{S} \diamond \mathcal{Z}$ and $\pi \diamond \mathcal{R} \diamond \mathcal{Z}$ are negligibly close for all PPT environments $\mathcal{Z} \in \text{Env}$.

We can rephrase the definition with the new notation as follows. We say that π *securely realizes* F in the $\mathcal{R}$-hybrid model if there exists a PPT simulator $\mathcal{S}$ such that $\mathsf{F} \diamond \mathcal{S} \stackrel{\text{comp}}{\equiv} \pi \diamond \mathcal{R}$. We say that there is statistical security if $\mathsf{F} \diamond \mathcal{S} \stackrel{\text{stat}}{\equiv} \pi \diamond \mathcal{R}$. We say that there is perfect security if $\mathsf{F} \diamond \mathcal{S} \stackrel{\text{perf}}{\equiv} \pi \diamond \mathcal{R}$.

We will sometimes further overload notation like $\pi \diamond \mathcal{R} \diamond \mathcal{D} \diamond \mathcal{Z}$ and use it to denote the random variable describing the *trace* of the execution, i.e., (k, z) plus the ordered list of the random tapes of all ITMs plus the ordered list of pairs $(\texttt{name}, m)$ specifying which messages were sent on which ports and in which order.

3 Adaptive versus Static Security Revisited

In this section, we show a general proof strategy for proving adaptive security. The idea is to first prove static security and then construct, from the simulator $\mathcal{S}$ we built, a new simulator $\mathcal{S}'$ for the adaptive case. Roughly speaking, the strategy for $\mathcal{S}'$ is to follow the algorithm of $\mathcal{S}$, but every time a new player P_i is corrupted, $\mathcal{S}'$ cooks up a view for P_i that "looks convincing", gives this to the environment, patches the state of $\mathcal{S}$ accordingly and continues.

It turns out that there is a class of unconditionally secure protocols and functionalities where this idea works and our goal will be to characterize this class and point out what the procedure run by $\mathcal{S}'$ to handle corruptions must satisfy. We will consider the case of perfect security first and later show that the results are also true in some cases for statistical security.

So we will assume we are given protocol π, communication resource functionality $\mathcal{R}$, ideal functionality F, and simulator $\mathcal{S}$ such that $\pi \diamond \mathcal{R} \stackrel{\text{perf}}{\equiv} \mathcal{S} \diamond \mathsf{F}$ for all static, unbounded environments that corrupts only subsets from some adversary structure $\mathbb{A}$. We write $\pi \diamond \mathcal{R} \stackrel{\text{perf}}{\equiv}_{\mathbb{A},\texttt{static}} \mathcal{S} \diamond \mathsf{F}$. We will assume synchronous protocols only.

We will need the following notation.

Definition 2. *For an ITM A (ITM) that is part of an interactive system $\mathcal{IS}$, the* view *of A is a random variable, written $V_{\mathsf{A}}(\mathcal{IS})$, and is defined to be the ordered concatenation of all messages exchanged on the ports of A and of the random choices of A, i.e., a random trace of $\mathcal{IS}$ restricted to the values seen by A. We use $V_{\mathsf{A}}(\mathcal{IS}|E)$ to denote the view when conditioned on some event E occurring, and $V_{\mathsf{A}}(\mathcal{IS})_j$ to denote the view truncated to contain only the values associated with the first j rounds – we only consider synchronous protocols, so the notion of round is well defined.*

Definition 3. *For a player P_i in a protocol π running with communication resource $\mathcal{R}$, and environment $\mathcal{Z}$, the* conversation *of P_i is a random variable, written $\text{Conv}_{\mathsf{P}_i}(\mathcal{Z} \diamond \pi \diamond \mathcal{R})$, and is defined to be the ordered concatenation of all messages P_i exchanges with honest players and $\mathcal{R}$. For a set C of parties we let $\text{Conv}_C(\mathcal{Z} \diamond \pi \diamond \mathcal{R})$ be the set of $\text{Conv}_{\mathsf{P}_i}(\mathcal{Z} \diamond \pi \diamond \mathcal{R})$ for $\mathsf{P}_i \in C$. Likewise, the conversation of $\mathcal{Z}$, written $\text{Conv}_{\mathcal{Z}}(\mathcal{Z} \diamond \pi \diamond \mathcal{R})$, is the ordered concatenation of all messages $\mathcal{Z}$ exchanges with honest players in π and $\mathcal{R}$. For conversations, we denote truncation and conditioning on events in the same way as for views.*

Note that the conversation of a party is a substring of its view. Also note that when a player P_i is corrupted, its view becomes a substring of the conversation of $\mathcal{Z}$, because $\mathcal{Z}$ learns

from the (up to now) honest P_i the entire view of P_i up to the corruption. We may think of $\mathrm{Conv}_{\mathcal{Z}}(\mathcal{Z} \diamond \pi \diamond \mathcal{R})$ as the total information $\mathcal{Z}$ gets from attacking the protocol. Recall, however that $\mathcal{Z}$ also chooses the inputs of honest players and learns their outputs.

We will need to assume that the ideal functionality has a certain behavior. First it must ensure that whenever it receives input or gives output, the time at which this happens is publicly known, i.e., the functionality leaks the information that a party just received an output as well as the the name of that party. The second demand is meant to capture the idea that the functionality should treat players who are corrupt but behave honestly in the same way as if they were honest. We give an intuitive explanation after the definition.

Definition 4. *The ideal functionality F is said to be* input-based *if the following is satisfied:*

Honest behavior equivalence: *Consider executions of F where some set A is corrupted from the start and where a fixed (ordered) set of inputs I_{F} are given to F during the execution*[2]*. In any such execution the outputs produced by F and its state at the end has the same distribution, in particular the distributions do not depend on the corruptions that occur during the execution.*[3]

Publicly known input-output provision: *Each time F receives an input from P_i, F leaks a message specifying that some input from P_i has been received.*[4] *Each time F sends a private output to P_i, it also leaks a message, specifying that an output was given, but not the value.*

The "honest behavior equivalence" condition is essentially to the notion of a "well-formed ideal functionality" [9] and can be intuitively explained as follows: an ideal functionality knows which players are corrupt and its actions may in general depend arbitrarily on this information. The condition puts a limitation on this: Consider first an execution where all players outside A remain honest and F gets I_{F} as input. Compare this to a case where $\mathsf{P}_i \notin A$ is corrupted, but F still gets the same inputs. This means in particular that P_i sends the same inputs, so he "behaves honestly" towards F. Therefore, the demand we make loosely speaking means that as long as a corrupt player behaves honestly towards the functionality, the actions it takes will be the same as if that player had been honest.

Our results will also be valid for a slightly more general case where the outputs produced by F do not have to be the same in all executions, but the outputs in one execution can be efficiently computed from I_{F} and the outputs in any other execution. For simplicity we do not treat this general case explicitly in the following.

We also need to make some assumptions on how the simulator $\mathcal{S}$ behaves, more precisely on how it decides on the inputs it sends to F on behalf of corrupted players (recall that once a player is corrupted, the simulator gets to decide which inputs this player provides to F). We will assume that $\mathcal{S}$ uses a standard technique to decide on these inputs, namely at the time where the input is given, it looks at the conversation of the corrupt player and decides on its input based on this. This is formalized as follows:

Definition 5. *The simulator $\mathcal{S}$ is* conversation-based *if the following is satisfied:*

[2] Notice that these inputs will arrive from different parties, depending on whether the player giving input is honest or is controlled by the adversary/environment in the given execution.

[3] Note that technically, in the UC framework F is informed of corruptions, so its state contains information about who is corrupted and when. So strictly speaking, the state cannot be exactly the same in all cases. However, we require that up to the fact that different sets of corrupted players are stored, the state is exactly the same. This can be formalized by saying that $\mathsf{F} = \mathsf{F}_{\mathtt{wrap}}(\mathsf{F}_{\mathtt{core}})$ for a core functionality $\mathsf{F}_{\mathtt{core}}$ which has *honest behavior equivalence* in the strict sense plus a wrapper $\mathsf{F}_{\mathtt{core}}$ who is informed who is corrupted but does not forward it to the core, but otherwise acts as a channel between its environment and the core.

[4] By leaking a message we mean that the message is sent on the port connected to the adversary/simulator.

Conversation-based inputs: *If $\mathcal{S}$ sends an input x to F on behalf of P_i in round j, it computes x as $x = \mathrm{Inp}_i(c)$ where Inp_i is a PPT function depending only on the protocol and $c = \mathrm{Conv}_C(\mathcal{Z} \diamond \mathcal{S} \diamond \mathsf{F})_j$, where C is the set of corrupted parties.*

Honest behavior implies correct inputs: *If P_i is corrupt but has followed the protocol honestly then it is always the case that $\mathrm{Inp}_i(c)$ equals the corresponding input P_i was given from the environment. By a corrupt P_i following the protocol honestly, we mean that the environment decides the actions of P_i by running a copy of the code of the honest P_i on the inputs and messages that P_i receives in the protocol and a uniformly independently chosen random tape.*[5]

Corruption-consistent input functions: *Consider the conversations $c = \mathrm{Conv}_C(\mathcal{Z} \diamond \mathcal{S} \diamond \mathsf{F})_j$ of some corrupted parties, and consider the conversations $c' = \mathrm{Conv}_{C'}(\mathcal{Z} \diamond \mathcal{S} \diamond \mathsf{F})_j$ of some other set of corrupted parties, where both C and C' are allowed to be corrupted and $C \subset C'$. For all such c, c' and all input functions Inp_i for parties in C, it must be the case that $\mathrm{Inp}_i(c) = \mathrm{Inp}_i(c')$.*

Note that input functions only have to be defined on conversations that actually occur in π. Also note that the corruption consistency of the input functions model the following reasonable intuition: consider a run of π that leads to certain inputs. Now suppose we run π again with the same random coins, however some players that were honest before are now corrupted, but are told to play honestly. Since all players make the same moves in the two cases, it is reasonable to expect that the resulting inputs should be the same, and this is what the corruption-consistency of the input functions implies. This is in some sense the requirement that the input functions are "well-formed".

A typical example of an input function is where P_i provide inputs by secret-sharing them using polynomials of degree at most t. Here the input function reconstructs the input from the shares held by honest players using Lagrange interpolation. If the protocol guarantees that the shares are consistent with some polynomial of degree at most t, even if P_i is actively corrupt, then the input function only has to be defined on such sets of shares and is indeed corruption consistent.

Now, suppose we are given an adaptive environment $\mathcal{Z}$ and we want to show that the protocol is secure with respect to this environment. For this, we need to think about how we can use a static simulator $\mathcal{S}$. Of course, we cannot just run it against $\mathcal{Z}$ because $\mathcal{S}$ does not know how to handle corruptions that occur in the middle of the protocol. So instead, we will construct a family of static environments from $\mathcal{Z}$.

For each set A that $\mathcal{Z}$ may corrupt, we construct an environment $\mathcal{Z}_A$. Informally, what $\mathcal{Z}_A$ does is that it corrupts set A, but initially, it lets all players in A play honestly. It runs internally a copy of $\mathcal{Z}$ and lets it interact with the protocol as usual, where the only difference is that players in A are run honestly by $\mathcal{Z}_A$ instead of running by themselves. When $\mathcal{Z}$ corrupts a player in A, $\mathcal{Z}_A$ gives control of that player to $\mathcal{Z}$ and continues, if the corrupted player is not in A, $\mathcal{Z}_A$ outputs guess 0 and terminates. If $\mathcal{Z}$ outputs a guess $c \in \{0, 1\}$ without corrupting anyone outside A, then $\mathcal{Z}_A$ outputs the same guess c. A formal description is found below.

We know that $\mathcal{S}$ can do perfect simulation against any of the $\mathcal{Z}_A$ we just defined and this will be the basis of the adaptive simulator we construct later. Before we can construct the adaptive simulator, we need some auxiliary lemmas on how $\mathcal{S}$ behaves when interacting with the $\mathcal{Z}_A$'s.

Some notation: In the following $\mathbf{s}$ will denote an ordered sequence of players, and $A(\mathbf{s})$ will denote the set of players that occur in $\mathbf{s}$. $E_{\mathbf{s}}$ will be the event that the first $|\mathbf{s}|$ corruptions done

[5] It is slightly tricky to formally and *generally* define what "following the protocol honestly" means, as an actively corrupted party takes all its instructions from the environment. However, in our context the definition we give here will do, where we make a structural requirement on the environment that it contains a copy of the honest party.

Agent $\mathcal{Z}_A$

Static environment constructed from $\mathcal{Z}$.

1. Initially corrupt set A. Set up internally a (honest) copy P'_j of each player $\mathsf{P}_j \in A$. Also set up internally a copy of $\mathcal{Z}$.
2. When the system executes and some $\mathsf{P}_j \in A$ is activated, then if P_j has not been corrupted by $\mathcal{Z}$ (see next item) $\mathcal{Z}_A$ does the following: $\mathcal{Z}_A$ gives a copy of the messages received by P_j to its internal copy P'_j and runs its code to decide (honestly) what to send.
3. If $\mathcal{Z}$ decides to corrupt $\mathsf{P}_j \in A$, $\mathcal{Z}_A$ gives the current state of P'_j to $\mathcal{Z}$ and gives (passive or active) control of P_j to $\mathcal{Z}$. Note that this means that after this point any messages meant for P_j (from $\mathcal{R}$ or from other players) are forwarded to $\mathcal{Z}$ and $\mathcal{Z}_A$ runs the code of $\mathcal{Z}$ to determine what messages P_j should send.
4. If $\mathcal{Z}$ decides to corrupt $\mathsf{P}_j \notin A$, $\mathcal{Z}_A$ halts and outputs the guess 0.
5. If $\mathcal{Z}$ halts (having corrupted no player outside A) with guess $c \in \{0,1\}$, $\mathcal{Z}_A$ halts and outputs guess c.

by the environment are exactly those in $\mathbf{s}$ (in the specified order). Below, when we write views or conversations with subscript $\mathbf{s}$, for instance as in $V_{\mathcal{Z}}(\mathcal{Z} \diamond \pi \diamond \mathcal{R}|E_{\mathbf{s}})_{\mathbf{s}}$, this means that if a corruption outside $\mathbf{s}$ occurs, we truncate the view at the point where this corruption happens.

Finally, consider the copy of $\mathcal{Z}$ that is run internally by $\mathcal{Z}_{A(\mathbf{s})}$ where we execute $\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{IS}$ for some interactive system $\mathcal{IS}$ (such as $\pi \diamond \mathcal{R}$). We then let $V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{IS}|E_{\mathbf{s}})$ be its view, conditioned on $E_{\mathbf{s}}$. Since $\mathcal{Z}_{A(\mathbf{s})}$ runs $\mathcal{Z}$ "in the head" it is clear that $V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{IS}|E_{\mathbf{s}})$ can be deterministically and easily extracted from $V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{IS}|E_{\mathbf{s}})$, we will write this as

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{IS}|E_{\mathbf{s}}) = \mathrm{Extr}(V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{IS}|E_{\mathbf{s}})) \ .$$

We can now show that assuming $E_{\mathbf{s}}$ occurs, then $\mathcal{S}$ can be used to perfectly simulate (a part of) the view $\mathcal{Z}$ sees in the protocol, because it can simulate the view of $\mathcal{Z}_{A(\mathbf{s})}$:

Lemma 1. *Assuming $\mathcal{R}$ is input based, we have*

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}|E_{\mathbf{s}}) \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}}(\mathcal{Z} \diamond \pi \diamond \mathcal{R}|E_{\mathbf{s}})_{\mathbf{s}} \ .$$

Proof. We have $V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}) \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R})$, since $\mathcal{S}$ is a good static simulator. So the two distributions are also the same when conditioning on $E_{\mathbf{s}}$, that is, we have

$$V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}|E_{\mathbf{s}}) \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R}|E_{\mathbf{s}}) \ . \tag{1}$$

The two distributions remain equal if we apply the same deterministic function to both of them, so if we apply Extr on both sides of (1) we get

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}|E_{\mathbf{s}}) \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R}|E_{\mathbf{s}}) \ . \tag{2}$$

Moreover, from the point of view of $\mathcal{Z}$ (and still conditioning on $E_{\mathbf{s}}$), the only difference between $\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R}$ and $\mathcal{Z} \diamond \pi \diamond \mathcal{R}$ is that in the first case the parties in A are run honestly by $\mathcal{Z}_{A(\mathbf{s})}$ until $\mathcal{Z}$ wants to corrupt them while in the second case they run as honest players in π. This makes no difference to $\mathcal{R}$ since it is input based (by the honest behavior equivalence property) and hence it makes no difference to $\mathcal{Z}$ either. So we have

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R}|E_{\mathbf{s}}) \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}}(\mathcal{Z} \diamond \pi \diamond \mathcal{R}|E_{\mathbf{s}})_{\mathbf{s}} \ . \tag{3}$$

The lemma now follows from (2) and (3).

We also need to consider a connection between simulation against several different $\mathcal{Z}_{A(\mathbf{s})}$'s: For a sequence of players $\mathbf{s}$, we can append a player P_i (who is not in $\mathbf{s}$) at the end of the sequence. We write this new sequence as $\mathbf{s}, i$, and define $E_{\mathbf{s},i}$ and $A(\mathbf{s}, i)$ as before.

Lemma 2. *Assuming $\mathcal{R}$ is input based, we have*

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i}) \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}} \ .$$

Proof. Since $\mathcal{S}$ is a good static simulator, we have by a similar argument as in the proof of Lemma 1 that

$$\begin{aligned} V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i}) &= \mathrm{Extr}(V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})) && (4) \\ &\stackrel{\text{perf}}{\equiv} \mathrm{Extr}(V_{\mathcal{Z}_{A(\mathbf{s})}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R} | E_{\mathbf{s},i})) && (5) \\ &= V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R} | E_{\mathbf{s},i}) \ . && (6) \end{aligned}$$

Note that, assuming $E_{\mathbf{s},i}$ occurs, the only difference between $\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R}$ and $\mathcal{Z}_{A(\mathbf{s},i)} \diamond \pi \diamond \mathcal{R}$ is that in the latter case P_i is run honestly by $\mathcal{Z}_{A(\mathbf{s},i)}$ whereas in the former it plays honestly as party in the protocol. As $\mathcal{R}$ is input based, this makes no difference to $\mathcal{R}$ and hence also no difference to the view of $\mathcal{Z}$ as long as we only consider what happens up to the point where P_i is corrupted. So we have

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \pi \diamond \mathcal{R} | E_{\mathbf{s},i}) \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \pi \diamond \mathcal{R} | E_{\mathbf{s},i})_{\mathbf{s}} \ . \quad (7)$$

Using again that $\mathcal{S}$ is a good static simulator, it follows in that same way as before that

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \pi \diamond \mathcal{R} | E_{\mathbf{s},i})_{\mathbf{s}} \stackrel{\text{perf}}{\equiv} V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}} \ . \quad (8)$$

The lemma now follows from (6), (7) and (8).

We now want to show that if we consider both the view of $\mathcal{Z}$ and the inputs and outputs that F exchanges, we still have a similar result as in the previous lemma. This does not have to be true in general, but is indeed true if $\mathcal{S}$ is conversation-based and if F is input-based:

Lemma 3. *Let $\mathrm{St}_{\mathsf{F}}(\cdot)$ be the state of F after running in some interactive system. Then, if $\mathcal{S}$ is conversation-based and F is input-based, we have*

$$(V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i}), \ \mathrm{St}_{\mathsf{F}}(\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i}))$$

$$\stackrel{\text{perf}}{\equiv} (V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}}, \ \mathrm{St}_{\mathsf{F}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}}) \ .$$

Proof. We already have from Lemma 2 that the view of $\mathcal{Z}$ has the same distribution in the two systems. We then prove the lemma by arguing that because $\mathcal{S}$ is conversation-based, all inputs sent to F follow deterministically from the view of $\mathcal{Z}$ and will be the same in both systems, so since F is input-based, the distribution of its state must be the same as well.

In more detail, consider a view v for $\mathcal{Z}$ that occurs with non-zero probability (in both systems). Note first that by public input-output provision of F, one can infer from v in which rounds inputs were provided to F or outputs were sent, so these must be the same in the two systems. Consider a particular input and say it comes from player P_j. We do a case analysis:

$\mathbf{P}_j \notin A(\mathbf{s}, i)$ In this case P_j is honest throughout in both systems. This means $\mathcal{Z}$ provides the input directly to F so the input occurs in v and is the same in both systems.

$\mathsf{P}_j = \mathsf{P}_i$ In the system $\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}$, P_i is honest and $\mathcal{Z}$ provides directly to F the input, say x, that occurs in v. In $\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F}$, P_i is corrupt but is told to play honestly on input x as provided by $\mathcal{Z}$. Then $\mathcal{S}$ decides on the input to F using the input function on the conversations of the corrupted parties, and this will result in x by the "honest behavior implies correct input" property.

$\mathsf{P}_j \in A(\mathbf{s})$ **and has not been corrupted by $\mathcal{Z}$ when the input is provided** In this case, in both systems $\mathcal{Z}$ provides input x to P_j who plays honestly and $\mathcal{S}$ decides the input to F using the input function on the conversations of the corrupted parties, which will be x, again by the "honest behavior implies correct input" property.

$\mathsf{P}_j \in A(\mathbf{s})$ **and has been corrupted by $\mathcal{Z}$ when the input is provided** In this case $\mathcal{S}$ will in both systems decide on the input using the input function on the conversations of the corrupted parties. Note first that the messages a corrupted party has exchanged with all players that $\mathcal{Z}$ has not corrupted yet is part of v and is therefore the same in both systems (this includes at least all players outside $A(\mathbf{s})$). However, from the point of view of $\mathcal{S}$, the conversation of a corrupted party is *not* the same in the two systems: in $\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}$, it consists of messages exchanged with players outside $A(\mathbf{s})$, while in $\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F}$ it consists of a subset of these messages, namely those exchanged with players outside $A(\mathbf{s}, i)$. However, since the input functions are corruption-consistent, the input computed by $\mathcal{S}$ is nevertheless the same in the two systems.

In the following, we will consider a situation where we execute the system $\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}$ until a point where $E_{\mathbf{s},i}$ has occurred. At this point, $\mathcal{Z}_{A(\mathbf{s})}$ would halt. However, by Lemma 3, as far as $\mathcal{Z}$ and the state of F is concerned, we might as well have been running $\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F}$, and unlike $\mathcal{Z}_{A(\mathbf{s})}$, $\mathcal{Z}_{A(\mathbf{s},i)}$ would be able to continue even after P_i is corrupted. So if we could somehow "pretend" that in fact it was the latter system we ran, we would not have to stop when P_i is corrupted.

To help us do this trick, we consider an execution of $\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F}$ where $E_{\mathbf{s},i}$ occurs. Say that the values of $V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}}$ and $\mathrm{St}_{\mathsf{F}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}}$ are v and w. We then define $D_{v,w}$ to be the joint distribution of the states of $\mathcal{Z}_{A(\mathbf{s},i)}$ and $\mathcal{S}$ at the point where P_i is corrupted, given v and w. Note that since we assume that $E_{\mathbf{s},i}$ occurred, the state of $\mathcal{Z}_{A(\mathbf{s},i)}$ consists of a state of $\mathcal{Z}$ that is fixed by v and a view of P_i who has been playing honestly so far. So we can think of the output of $D_{v,w}$ as a view of P_i plus a state of $\mathcal{S}$.

Lemma 4. *Consider an execution of the system $\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F}$ until a point where $E_{\mathbf{s},i}$ has occurred. Let*

$$V_{\mathcal{Z}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}} = v \text{ and } \mathrm{St}_{\mathsf{F}}(\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F} | E_{\mathbf{s},i})_{\mathbf{s}} = w \ .$$

Let $\mathrm{io}_{\mathcal{S}}$ be the string of inputs and outputs $\mathcal{S}$ has exchanged with F, and let $\mathrm{Conv}_{\mathcal{Z}}$ be the conversation of $\mathcal{Z}$ in the execution. Then one can sample from the distribution $D_{v,w}$ if given $\mathrm{io}_{\mathcal{S}}$ and $\mathrm{Conv}_{\mathcal{Z}}$. In particular, $D_{v,w}$ depends only on $\mathrm{io}_{\mathcal{S}}$ and $\mathrm{Conv}_{\mathcal{Z}}$.

Proof. Recall that $\mathcal{Z}_{A(\mathbf{s},i)}$ consists of a copy of $\mathcal{Z}$ and copies of players in $A(\mathbf{s}, i)$. However, since $E_{\mathbf{s},i}$ occurs, all players in $A(\mathbf{s})$ have been corrupted earlier by $\mathcal{Z}$, so their entire view until they were corrupted by $\mathcal{Z}$ is part of $\mathrm{Conv}_{\mathcal{Z}}$.

The sampling procedure we claim is now very simple: for each possible set of coins for P_i and for $\mathcal{S}$, we will test if these coins are consistent with the values of $\mathrm{io}_{\mathcal{S}}$ and $\mathrm{Conv}_{\mathcal{Z}}$ we are given. We do the test by simulating P_i and $\mathcal{S}$ running as part of the system $\mathcal{Z}_{A(\mathbf{s},i)} \diamond \mathcal{S} \diamond \mathsf{F}$. This is possible because the given values $\mathrm{io}_{\mathcal{S}}$ and $\mathrm{Conv}_{\mathcal{Z}}$ specify the entire communication that P_i and $\mathcal{S}$ should have with F and $\mathcal{Z}$. If the current random coins lead to P_i or $\mathcal{S}$ sending a message that is inconsistent with $\mathrm{io}_{\mathcal{S}}, \mathrm{Conv}_{\mathcal{Z}}$, we throw away this set of coins. Finally, we choose randomly a set of coins among those that survived and output the resulting view of P_i and state of $\mathcal{S}$.

Note that we only prove that one can sample from $D_{v,w}$, we do not claim that one can sample *efficiently* from this distribution. In fact, this does not hold in general. In view of the result of Lemma 4, we will write $D_{\mathrm{io}_{\mathcal{S}},\mathrm{Conv}_{\mathcal{Z}}}$ instead of $D_{v,w}$ in the following. We can now specify the final tool we need to build an adaptive simulator, namely the sampling we have just seen must be possible to do efficiently:

Definition 6. *Consider a probabilistic algorithm* Patch *that takes as input strings* $\mathrm{io}_{\mathcal{S}}$ *and* $\mathrm{Conv}_{\mathcal{Z}}$ *of the form as specified in Lemma 4.* Patch *is said to be a* good sampling function *if it satisfies the following:*

- *It is polynomial time computable.*
- *The output* $\mathrm{Patch}(\mathrm{io}_{\mathcal{S}}, \mathrm{Conv}_{\mathcal{Z}})$ *is distributed according to* $D_{\mathrm{io}_{\mathcal{S}},\mathrm{Conv}_{\mathcal{Z}}}$.

We are now finally ready to specify the main result of this section:

Theorem 1. *Assume we are given a simulator* $\mathcal{S}$ *for protocol* π *and functionality* F *such that* $\pi \diamond \mathcal{R} \overset{\mathrm{perf}}{\equiv}_{\mathbb{A},\mathtt{static}} \mathcal{S} \diamond \mathsf{F}$. *Assume further that* $\mathcal{S}$ *is conversation-based,* F *and* $\mathcal{R}$ *are input-based, and that we are given a good sampling function* Patch. *Then there exists a simulator* $\mathcal{S}'$ *such* $\pi \diamond \mathcal{R} \overset{\mathrm{perf}}{\equiv}_{\mathbb{A},\mathtt{adaptive}} \mathcal{S} \diamond \mathsf{F}$, *i.e.,* $\pi \diamond \mathcal{R} \diamond \mathcal{Z} \overset{\mathrm{perf}}{\equiv}_{\mathbb{A},\mathtt{adaptive}} \mathcal{S} \diamond \mathsf{F} \diamond \mathcal{Z}$ *for all for all adaptive and synchronous environment* $\mathcal{Z}$ *corrupting only subsets from* $\mathbb{A}$.

Proof. We specify the algorithm of our adaptive simulator $\mathcal{S}'$. To do this, suppose that if we are given a string $\mathrm{io}_{\mathcal{S}}$ containing inputs and outputs that $\mathcal{S}$ has exchanged with F on behalf of corrupted players in a set A. Suppose we are also given the inputs and outputs io_i that some honest player P_i has exchanged with F in the same execution. Then we can merge these strings in a natural way: we define $\mathrm{Merge}(\mathrm{io}_{\mathcal{S}}, \mathrm{io}_i)$ to be the string that contains, for every protocol round, the inputs to F that occur in either $\mathrm{io}_{\mathcal{S}}$ or io_i, and also outputs from F that occur in either $\mathrm{io}_{\mathcal{S}}$ or io_i. Note that $\mathrm{Merge}(\mathrm{io}_{\mathcal{S}}, \mathrm{io}_i)$ is a string of inputs and outputs that $\mathcal{S}$ might have exchanged with F if $A \cup \mathsf{P}_i$ had been the corrupted set (and P_i had behaved honestly).

Agent $\mathcal{S}'$

Adaptive simulator constructed from $\mathcal{S}$.

1. Set $\mathbf{s}$ be the empty sequence. Set $\mathrm{io}_{\mathcal{S}}, \mathrm{Conv}_{\mathcal{Z}}$ to be the empty strings. Set up a copy of $\mathcal{S}$ in its initial state. Tell $\mathcal{S}$ as input (in the preamble) that the empty set is the corrupted set.
2. Whenever $\mathcal{S}'$ is activated, if the input is a request to corrupt a new player P_i, it goes to the next step. Otherwise, it runs $\mathcal{S}$ on the input received and sends the output $\mathcal{S}$ produces on the corresponding output port of its own. Messages exchanged with $\mathcal{Z}$ are appended to $\mathrm{Conv}_{\mathcal{Z}}$, inputs/outputs exchanged with F are appended to $\mathrm{io}_{\mathcal{S}}$.
3. Set $\mathbf{s} = \mathbf{s}, i$. Send a request to corrupt P_i to F and get s string io_i back. Set $\mathrm{io}_{\mathcal{S}} = \mathrm{Merge}(\mathrm{io}_{\mathcal{S}}, \mathrm{io}_i)$. Compute $(v_i, \mathrm{St}') = \mathrm{Patch}(\mathrm{io}_{\mathcal{S}}, \mathrm{Conv}_{\mathcal{Z}})$. Put $\mathcal{S}$ in state St', send v_i to the environment, append v_i to $\mathrm{Conv}_{\mathcal{Z}}$ and go to Step 2.

To see that this simulation works, note first that it is obvious that $\mathrm{Conv}_{\mathcal{Z}}$ contains at all times the conversation of $\mathcal{Z}$ so far, and that $\mathrm{io}_{\mathcal{S}}$ contains at all times the inputs and outputs we have exchanged with F so far.

We can now show the following

Claim: whenever $\mathcal{S}'$ enters step 2 the state of $\mathcal{Z}, \mathcal{S}$ and F are distributed exactly as in a run of $\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}$ where $E_{\mathbf{s}}$ occurs.

We show this by induction: the claim is trivial when we enter step 2 the first time since here $\mathbf{s}$ is empty. So consider a later stage where we enter step 2, and write the current $\mathbf{s}$ as $\mathbf{s} = \mathbf{s}', i$. The previous time we entered step 2, by induction hypothesis, the state of $\mathcal{Z}, \mathcal{S}$ and F were distributed exactly as in a run of $\mathcal{Z}_{A(\mathbf{s}')} \diamond \mathcal{S} \diamond \mathsf{F}$ where $E_{\mathbf{s}'}$ occurs. During the following execution of step 2, $\mathcal{S}'$ simply ran $\mathcal{S}$, so when the i'th player is corrupted, the state of the state of $\mathcal{Z}, \mathcal{S}$ and F were distributed exactly as in a run of $\mathcal{Z}_{A(\mathbf{s}')} \diamond \mathcal{S} \diamond \mathsf{F}$ where $E_{\mathbf{s}',i}$ occurs. Now, by Lemma 3, the views (and hence state) of $\mathcal{Z}$ and the state of F are distributed as in a run of $\mathcal{Z}_{A(\mathbf{s}',i)} \diamond \mathcal{S} \diamond \mathsf{F}$ where $E_{\mathbf{s},i}$ occurs.

Then Patch was run and the claim now follows by assumption on Patch, if we show that the inputs $\mathrm{Conv}_{\mathcal{Z}}, \mathrm{io}_{\mathcal{S}}$ we use have the distribution they would have in $\mathcal{Z}_{A(\mathbf{s}',i)} \diamond \mathcal{S} \diamond \mathsf{F}$, given the current values of the view of $\mathcal{Z}$ and the state of F. This is trivially true for $\mathrm{Conv}_{\mathcal{Z}}$ as it follows deterministically from the view of $\mathcal{Z}$. For $\mathrm{io}_{\mathcal{S}}$, note that the inputs to F that occur in this string also follow deterministically from the view of $\mathcal{Z}$, we argued this in the proof of Lemma 3. But since F is input-based, the resulting outputs from F will be the same regardless of whether we run $\mathcal{Z}_{A(\mathbf{s})}$ or $\mathcal{Z}_{A(\mathbf{s}',i)}$, and so $\mathrm{io}_{\mathcal{S}}$ has the desired distribution.

We can now argue that $V_{\mathcal{Z}}(\mathcal{Z} \diamond \pi \diamond \mathcal{R}) \stackrel{\mathrm{perf}}{\equiv} V_{\mathcal{Z}}(\mathcal{Z} \diamond \mathcal{S}' \diamond \mathsf{F})$ which clearly implies the theorem.

We will consider the executions of step 2 one by one. In the first execution, by the above claim, the state of $\mathcal{Z}, \mathcal{S}$ and F are distributed exactly as in a run of $\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}$ where $E_{\mathbf{s}}$ occurs. But here $\mathbf{s}$ is the empty sequence so $E_{\mathbf{s}}$ always occurs. In step 2, we simply run $\mathcal{S}$, so by Lemma 1, we obtain a perfect simulation of $\mathcal{Z}$'s view until the point where is halts or corrupts the first player. In particular, in the latter case, that player is chosen with the distribution we would also see in a real execution of the protocol. When we have executed step 3, again by the claim, the state of $\mathcal{Z}, \mathcal{S}$ and F are distributed exactly as in a run of $\mathcal{Z}_{A(\mathbf{s})} \diamond \mathcal{S} \diamond \mathsf{F}$ where $E_{\mathbf{s}}$ occurs, and where $\mathbf{s}$ now contains one player. Again by Lemma 1, while executing step 2 we obtain a perfect simulation of $\mathcal{Z}$'s view from the point where it corrupts the first player until the point where is halts or corrupts the second player.

Repeating this argument at most n times (since $\mathcal{Z}$ can corrupt only so many players), we see that we get a perfect simulation of the entire view of $\mathcal{Z}$.

Using Theorem 1 In order to use Theorem 1 on a concrete protocol, one has to first construct a static simulator $\mathcal{S}$, verify that is is conversation-based and that the target functionality F is input-based. This is usually quite easy. Then one has to construct an efficient procedure Patch. This may seem harder because the formal definition is quite technical and involves two static environments constructed from an arbitrary adaptive environment.

We therefore give an explanation in more "human" language of what Patch must be able to do. Recall that when Patch is called, players in the sequence $\mathbf{s}$ were corrupted earlier and a new player P_i has just been corrupted. We know $\mathrm{io}_{\mathcal{S}}$, i.e., all the inputs and outputs that players in $\mathbf{s}, i$ have exchanged with the functionality F, and we know $\mathrm{Conv}_{\mathcal{Z}}$, that is, the protocol execution as seen by $\mathcal{Z}$ until now. Patch now has two tasks that must be solved efficiently:

The first one is to construct a complete view of P_i playing honestly in the protocol until now, and this must be consistent with $\mathrm{io}_{\mathcal{S}}$ and $\mathrm{Conv}_{\mathcal{Z}}$. In particular, $\mathrm{io}_{\mathcal{S}}$ contains the inputs and outputs P_i has exchanged with F and $\mathrm{Conv}_{\mathcal{Z}}$ contains the messages that P_i has sent to players who were corrupted earlier. The reason why this can be feasible for, e.g., protocols based on secret sharing is that the corrupted players (actually, $\mathcal{Z}$) have seen less than t shares of the secrets of P_i. This leaves the secrets undetermined, so when we now learn the actual secret values of P_i (from $\mathrm{io}_{\mathcal{S}}$), we are able to create a full set of shares that is consistent with the secrets and the shares of the corrupt players.

The second task is to create a new state for the simulator $\mathcal{S}$. This must be the state as it would have looked if we had run $\mathcal{S}$ with all players in $\mathbf{s}, i$ being corrupt from the start, but where P_i plays honestly until the current point in time.

We point out that UC proofs in the existing literature often use a strategy for building a static simulator $\mathcal{S}$ that actually makes both tasks easier: Initially, $\mathcal{S}$ sets up a internally copies of the honest players in the protocol, and gives them dummy inputs. It now simulates by letting these "virtual players" execute the protocol with the corrupt players (controlled by $\mathcal{Z}$). The state of $\mathcal{S}$ is simply the state of the virtual players. Now, when P_i is corrupted, Patch will compute how the view of the virtual copy of P_i should change, now that we know its actual inputs, and will then update the state of the other virtual players to make everything consistent, including $\text{io}_{\mathcal{S}}$ and $\text{Conv}_{\mathcal{Z}}$. This creates the required view of P_i and the new state of $\mathcal{S}$ is the state of the updated virtual players, except that of P_i. It is not hard to see that if one can show that Patch generates correctly distributed states for the internal players, given $\text{io}_{\mathcal{S}}$ and $\text{Conv}_{\mathcal{Z}}$, then Theorem 1 applies and we get adaptive security.

Extension to Statistical Security It is not clear that Theorem 1 is true for statistical security in general. But it not hard to see that it holds in an important special case: suppose we can define an "error event" E such that E occurs with negligible probability, and we can make a static simulator $\mathcal{S}$ that simulates perfectly if E does not occur. Then we can redo the proof of Theorem 1 while conditioning throughout on E not occurring. We leave the details to the reader.

4 Adaptive UC Security of the BGW protocol

The protocol. For simplicity we will only consider security of the passively secure version of BGW. The analysis extend to the active case using known fairly standard arguments from secret-sharing. We assume the reader is familiar with the protocol but as a reminder, each secret value a in the computation is secret shared using a polynomial f_a of degree at most t of finite field $\mathbb{F}$, where the protocol is secure against corruption of t of the n players, and where $t < n/2$. Each player P_i then holds $f_a(i)$, and $f_a(0) = a$.

To add secret shared values a, b, each player P_i locally computes $f_a(i)+f_b(i)$ which effectively means we now have secret shared $a + b$ using polynomial $f_a + f_b$.

To multiply secret shared values a, b, each player P_i locally computes $f_a(i)f_b(i) = (f_af_b)(i)$, and secret shares this value using a random polynomial g_i, i.e., he sends $g_i(j)$ to each player P_j. Let $r_1, ..., r_n$ be the Lagrange coefficients with the property that $\sum_{i=1}^{n} r_i h(i) = h(0)$ for any h of degree less than n. Then each player P_j computes $c_j = \sum_{i=1}^{n} r_i g_i(j)$. If we define $f_c = \sum_{i=1}^{n} r_i g_i$, then since the degree of f_af_b is less than n, it is not hard to see that $f_c(0) = f_a(0)f_b(0) = ab$ and that $f_c(j) = c_j$ so that we have effectively secret shared the product $c = ab$[6].

Note that since each honest player contributes a random polynomial g_i, this protocol ensures that the polynomial used to secret share the product is a random polynomial of degree at most t with the only constraint that it determines ab as the secret.

To compute a function securely, players secret share their inputs, work their way through an arithmetic circuit computing there desired function and finally open the results, by broadcasting their shares. For simplicity we will assume that each player P_i has a single input x_i and we want to compute a single public output y.

[6] This is actually not quite the original BGW multiplication protocol, but it is simpler to consider this variant for our purposes. As discussed in the introduction, it is in fact unclear where the protocol can be proven adaptively secure without this modification.

The functionality. The natural functionality one would expect this protocol to implement is one that gets input from all players, computes the desired function and outputs the results to everyone. Such a functionality is clearly input based: as long as the inputs it gets are the same, the result will be the same, regardless of who is corrupted.

The simulator. A static simulator for the protocol is very easy to describe: if the player set A is corrupted from the start, then the simulator sets up internally dummy players $\tilde{\mathsf{P}}_i$ for each $\mathsf{P}_i \notin A$ and gives them dummy inputs. The dummy players will execute the code prescribed by the protocol.

The simulator now lets the dummy players execute the protocol with the players in A, who are controlled by the environment. When a player in A secret shares his input, the simulator reconstructs the input from the shares that are sent to the honest (dummy) players, and passes these inputs to the functionality. This gives a perfect simulation of all steps up to the phase where outputs are opened: the environment never sees more than t shares of any value held by honest players, and hence in its view, there is no difference between dummy and real players.

When an output y is about to be opened, the simulator gets the correct output y from the functionality. The players hold a polynomial $f_{y'}$ that represents the output, but of course we cannot expect that $y' = y$ since y' was computed from dummy inputs. The simulator therefore computes a polynomial g of degree at most t with the property that $g(0) = y - y'$ and $g(i) = 0$ for all $\mathsf{P}_i \in A$. Note that if less than t players are corrupted these constraints do not determine g, so a random polynomial satisfying the constraints is chosen.

The simulator now pretends that in fact the polynomial $f_y = f_{y'} + g$ is held by the players, which is possible as only the state of dummy players need to be changed. Now the opening will indeed determine the correct y.

It is very easy to see that this is a perfect static simulator and that it is conversation based.

An assumption on the circuit. Before we continue we describe an assumption we will make on the structure of the circuit: we will assume that each output comes directly out of a multiplication gate. This will make the proof below much easier and is perhaps even essential. This can be assumed essentially without loss of generality: we can just add multiplication by dummy value 1 if needed. The effect of this assumption is that the polynomial that is opened is random of degree at most t with the only constraint that it determines the correct output. In particular it is independent of all random choices made by players earlier.

Patching the views. We now show how to construct the procedure Patch that is required before we can use our main theorem to conclude adaptive security. So we assume that a player P_k may be corrupted in any round during the protocol. If this happens, the simulator learns the true value of the input x_k and then it has to show the internal state of P_k to the environment. For this, we patch the state of $\tilde{\mathsf{P}}_k$ so that it is consistent with x_k and the rest of the values shown to the environment during the execution. Also the state of the remaining dummy players must be patched to be consistent with the state we created for P_k. We can then continue the static simulation from this point.

We describe here the case where P_k gets corrupted after the protocol is completed. Then, handling corruption at an earlier stage will simply consist of only doing a smaller part of the patching steps (namely up to the point where the player gets corrupted). Basically, the way to patch the state is to recompute every honest dummy player's share which is affected by the input of P_k, while not changing any of the shares that corrupted players have seen. This is done as follows:

- *Input Sharing.* Based on x_k we compute a new random secret sharing of x_k which is consistent with the (strictly less than t) shares shown to the environment and $f_{x_k}(0) = x_k$ and updates the shares of the dummy players to be consistent with f_{x_k}.
- *Addition or Multiplication by a constant.* Here the players only do local computations so we can simply recompute all the shares of dummy players which were affected by f_{x_k}.
- *Multiplication.* The first round in the processing of a multiplication gate only has local computations and hence, we recompute local value as above. Then in the second round, new shares are distributed of the product of two polynomials $f_a f_b$. If f_{x_k} is involved in one of these polynomials, we compute a new random secret sharing of $f_a(0)f_b(0)$ as it did for x_k in input sharing. In the third and last round the simulator is again able to recompute the shares of the dummy parties by local computations. Recall that the recombination vector is independent of $(f_a f_b)(\mathtt{X})$; it is the same for all polynomials of degree at most n, so it is indeed possible to redo this step with the new product.
- *Output Reconstruction.* We are given the correct output value y. Note that the environment has already seen (points on) a polynomial f_y that was created earlier. Therefore we must patch the state such that P_k ends up holding $f_y(k)$ as his share. Recall that in the real protocol, f_y is produced by the multiplication gate protocol. Our patching procedure applied to this gate produces a polynomial f_i for each player, and then the local recombination step determines a polynomial $f = \sum_{i \in S} r_i f_i$. Since the patching never changes the shares of players in C, we have $f_{y_j}(i) = f(i)$ for each $\mathsf{P}_i \in C$. But we also need that $f_{y_j}(k) = f(k)$, and this is not guaranteed.
  To correct for this, note that there must exist an honest player P_{i_0} such that $r_{i_0} \neq 0$, otherwise the corrupt players could reconstruct the product on their own. We now choose a random polynomial g of degree at most t, subject to

$$g(0) = 0, \quad g(i) = 0 \text{ for all } \mathsf{P}_i \in C, \text{ and } g(k) = r_{i_0}^{-1}(f_{y_j}(k) - f(k)).$$

  This is possible since at most $t-1$ players could be corrupted before P_k, so we fix the value of g in at most $t+1$ points. We now adjust the state of dummy player $\tilde{\mathsf{P}}_{i_0}$, such that we replace its polynomial f_{i_0} by $f_{i_0} + g$. This keeps the state of P_{i_0} internally consistent because $f_{i_0}(0) = (f_{i_0} + g)(0)$, and the shares of players in C are unchanged. However, we have now replaced f by $f + r_{i_0} g$, and clearly $(f + r_{i_0} g)(k) = f_{y_j}(k)$ as desired.

It is not hard to see that Patch as described above indeed produces polynomials for P_k and the remaining dummy players that are random, subject to the constraint that they are consistent with x_k, and the adversary's view so far. We therefore conclude that the protocol is adaptively secure.

Is the assumption on the circuit necessary? The intuitive reason why the above proof technique needs the assumption that every output value comes from a multiplication gate is as follows: When Patch is started, the environment has already been shown P_k's share of the output y, and this share is a result of a random choice that was made by the static simulator earlier. But now Patch produces a view for P_k starting from the input x_k and working its way forward through the protocol, making several random choices underway. This also leads to a share of y, but there is no reason to expect that it will agree with the one the environment has seen, as it should. However, because the polynomial used for y is produced from independent random choices from all honest players, we can adjust the random choice of a dummy player so that P_k's view will indeed lead to the right share of y.

One way to see this is on a higher level is as follows: what Patch needs to do is find random choices for P_k and the dummy players that are solutions to a set of equations that describe

what the choices must satisfy (namely the resulting views are consistent with x_k and the view of the environment). The multiplication gate assumption implies that the equations are linear and easy to solve, and so certainly simplifies the proof.

If we do not make this assumption, the resulting equations do still have a solution, because static security implies adaptive security in the MR definition, and this essentially means that there is a patching procedure, which however is not guaranteed to be efficient. At the time of writing, it is open whether an efficient solution exists. Note that since the local computations of players involve multiplications, it is not clear that the equations one needs to solve are linear.

References

1. *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, Chicago, Illinois, 2–4 May 1988.
2. G. Asharov and Y. Lindell. A full proof of the bgw protocol for perfectly-secure multiparty computation. *IACR Cryptology ePrint Archive*, 2011:136, 2011.
3. D. Beaver. Foundations of secure interactive computing. In J. Feigenbaum, editor, *Advances in Cryptology - Crypto '91*, pages 377–391, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.
4. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In ACM [1], pages 1–10.
5. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.
6. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, 14–17 Oct. 2001. IEEE.
7. R. Canetti, I. Damgård, S. Dziembowski, Y. Ishai, and T. Malkin. Adaptive versus non-adaptive security of multi-party protocols. *J. Cryptology*, 17(3):153–207, 2004.
8. R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 639–648, Philadelphia, Pennsylvania, 22–24 May 1996.
9. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 494–503, Montreal, Quebec, Canada, 2002.
10. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In ACM [1], pages 11–19.
11. R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient multiparty computations secure against an adaptive adversary. In J. Stern, editor, *Advances in Cryptology - EuroCrypt '99*, pages 311–326, Berlin, 1999. Springer-Verlag. Lecture Notes in Computer Science Volume 1592.
12. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [18], pages 643–662.
13. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, 25–27 May 1987.
14. U. Maurer. Constructive cryptography - a new paradigm for security definitions and proofs. In S. Mödersheim and C. Palamidessi, editors, *TOSCA*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.
15. S. Micali and P. Rogaway. Secure computation. In J. Feigenbaum, editor, *Advances in Cryptology - Crypto '91*, pages 392–404, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.
16. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [18], pages 681–700.
17. B. Pfitzmann, M. Schunter, and M. Waidner. Secure reactive systems. Technical Report RZ 3206, IBM Research, Zürich, May 2000.
18. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.