J. Cryptology (2006) 19: 135–167
DOI: 10.1007/s00145-005-0419-9

Journal of
CRYPTOLOGY

On the Limitations of Universally Composable Two-Party Computation Without Set-Up Assumptions*

Ran Canetti
IBM T.J. Watson Research,
19 Skyline Drive, Hawthorne, NY 10532, U.S.A.
canetti@watson.ibm.com

Eyal Kushilevitz
Computer Science Department, Technion,
Haifa 32000, Israel
eyalk@cs.technion.ac.il

Yehuda Lindell
Department of Computer Science, Bar-Ilan University,
Ramat Gan 52900, Israel
lindell@cs.biu.ac.il

Communicated by Matthew Franklin

Received 17 May 2004 and revised 5 June 2005
Online publication 30 November 2005

Abstract. The recently proposed *universally composable* security framework for analyzing security of cryptographic protocols provides very strong security guarantees. In particular, a protocol proven secure in this framework is guaranteed to maintain its security even when run concurrently with arbitrary other protocols. It has been shown that if a majority of the parties are honest, then universally composable protocols exist for essentially any cryptographic task in the *plain model* (i.e., with no set-up assumptions beyond that of authenticated communication). When honest majority is not guaranteed, general feasibility results are known only when given a trusted set-up, such as in the common reference string model. Only little was known regarding the existence of universally composable protocols in the plain model without honest majority, and in particular regarding the important special case of two-party protocols.

We study the feasibility of universally composable two-party *function evaluation* in the plain model. Our results show that in this setting, very few functions can be securely computed in the framework of universal composability. We demonstrate this by providing broad impossibility results that apply to large classes of deterministic and

* An extended abstract of this work appeared at EUROCRYPT 2003. Part of this work was done while Eyal Kushilevitz was a visitor at the IBM T.J. Watson Research Center. Most of this work was done while Yehuda Lindell was at the IBM T.J. Watson Research Center.

probabilistic functions. For some of these classes, we also present full characterizations of what can and cannot be securely realized in the framework of universal composability. Specifically, our characterizations are for the classes of deterministic functions in which (a) both parties receive the same output, (b) only one party receives output, and (c) only one party has input.

Key words. Secure two-party computation, Universal composability, Impossibility results.

1. Introduction

Traditionally, cryptographic protocol problems were considered in a model where the only involved parties are the actual participants in the protocol, and only a single execution of the protocol takes place. This model allows for relatively concise problem statements, simplifies the design and analysis of protocols, and is a natural choice for the initial study of protocols. However, this model of "stand-alone computation" does not fully capture the security requirements from cryptographic protocols in modern computer networks. In such networks, a protocol execution may run concurrently with an unknown number of other copies of the protocol and, even worse, with unknown, arbitrary protocols. These arbitrary protocols may be executed by the same parties or other parties, they may have potentially related inputs and the scheduling of message delivery may be adversarially coordinated. Furthermore, the local outputs of a protocol execution may be used by other protocols in an unpredictable way. These concerns, or "attacks," on a protocol are not captured by the stand-alone model. Indeed, over the years definitions of security became more and more sophisticated and restrictive, in an effort to guarantee security in more complex, multi-execution environments. However, in spite of the growing complexity, none of the proposed notions guarantee security in arbitrary multi-execution and multi-protocol environments.

An alternative approach to guaranteeing security in arbitrary protocol environments is to use notions of security that are preserved under general protocol composition. This approach was adopted in [C1], where a general framework for defining security of protocols was proposed. In this framework, called the universally composable (UC) security framework, protocols are designed and analyzed as stand-alone. Yet, once a protocol is proven secure, it is guaranteed that the protocol remains secure even when composed with an unbounded number of copies of either the same protocol or other unknown protocols. This guarantee is provided by a general *composition theorem*.

UC notions of security for a given task tend to be considerably more stringent than other notions of security for the same task. Consequently, many known protocols (e.g., the general protocol of [GMW] to name one) are *not* UC secure. Thus, the feasibility of realizing cryptographic tasks requires re-investigation within the UC framework. We briefly summarize the known results.

In the case of a majority of honest parties, there exist UC secure protocols for computing any functionality ([C1], building on [BGW], [RB], and [CFGN]). Also, in the honest-but-curious case (i.e., when even corrupted parties follow the protocol specification), UC secure protocols exist for essentially any functionality [CLOS]. However, the situation is different when no honest majority exists and the adversary is malicious (in which case the corrupted parties can arbitrarily deviate from the protocol specification).

In this case, UC secure protocols have been demonstrated for a number of specific (but important) functionalities such as key exchange and secure communication [CK1], [C2]. Furthermore, in the common reference string model, UC secure protocols exist for essentially any two-party and multi-party functionality, with any number of corrupted parties [CLOS].[1] In addition, it has been shown that in the plain model (i.e., assuming authenticated channels, but without any additional set-up assumptions), there are a number of natural two-party functionalities that *cannot* be securely realized in the UC framework. These include coin-tossing, bit commitment, and zero-knowledge [C1], [CF].

These results leave open the possibility that useful relaxations of the coin-tossing, bit commitment and zero-knowledge functionalities can be securely realized. A natural open question is what are the tasks that can be securely realized in the UC framework with a malicious adversary, no honest majority, and in the the plain model?

Our results. We concentrate on the case of *two-party function evaluation*, where the parties wish to evaluate some predefined function of their local inputs. We present extensive impossibility results for both deterministic and probabilistic functions in the UC framework. For some classes of functions, we also present full characterizations of what can and cannot be securely realized in the UC framework. Below, we refer to functions $f = (f_1, f_2)$ where the designated output of party P_1 is $f_1(x_1, x_2)$ and the designated output of party P_2 is $f_2(x_1, x_2)$. We now briefly summarize some of our results (all of the results below relate to feasibility in the *plain model*):

1. *Impossibility for general deterministic functions*: Informally stated, a function is said to be completely revealing for party P_1 if party P_2 can choose an input so that the output of the function (when applied to P_1's input and the input chosen by P_2) fully reveals P_1's input. That is, a function $f = (f_1, f_2)$ is completely revealing for P_1 if there exists an input x_2 for P_2 so that, for every x_1, it is possible to derive x_1 from $f_2(x_1, x_2)$.

   We prove that a deterministic two-party function $f = (f_1, f_2)$ *cannot* be securely realized in the UC framework unless it is completely revealing for both P_1 and P_2.
2. *Characterization for single-input deterministic functions*: A function f is single-input if it depends on at most one of its two inputs (i.e., $f(x_1, x_2) = g(x_i)$ for some function g and $i \in \{1, 2\}$). A function g is efficiently invertible if there exists an efficient inverting algorithm M that successfully inverts g for any samplable distribution over the input x.

   We prove that a deterministic single-input function can be securely realized in the UC framework *if and only if* it is efficiently invertible.
3. *Characterization for same-output deterministic functions*: A function f is same-output if $f_1 = f_2$ (i.e., both parties receive the same output).

   We prove that a deterministic same-output function can be securely realized in the UC framework *if and only if* it is single-input and efficiently invertible.

[1] In the common reference string model, it is assumed that all parties have access to a common string that is chosen by a trusted party according to some specified distribution. This implies an implicit trusted set-up phase.

4. *Characterization for single-output deterministic functions over finite domains*: A function f is single-output if one of f_1 or f_2 are empty (i.e., only one party receives output).

   We prove that a deterministic single-output function over a finite (or constant-size) domain can be securely realized in the UC framework *if and only if* it is completely revealing, as defined above.
5. *Impossibility for same-output probabilistic functions*: Loosely speaking, we say that a probabilistic function f is unpredictable for P_2 if there exists an input x_1 for P_1 such that for every input x_2 and every possible output value v, there is at least a non-negligible probability that $f(x_1, x_2)$ does not equal v. (That is, $f(x_1, x_2)$ is a random variable that does not almost always accept a single value v. In such a case, $f(x_1, x_2)$ defines a non-trivial distribution, irrespective of the value x_2 that is input by P_2.) Likewise, f is unpredictable for P_1 if there exists an x_2 such that for every x_1 and every v, with at least non-negligible probability $f(x_1, x_2)$ does not equal v.

   We prove that a probabilistic same-output function that is unpredictable for both P_1 and P_2 *cannot* be securely realized in the UC framework.[2]

Interestingly, our results hold unconditionally, in spite of the fact that they rule out protocols that provide only computational security guarantees. We remark that security in the UC framework allows "early stopping," or protocols where one of the parties may abort after it receives its output and before the other party has received output (that is, fairness is not required). Hence, our impossibility results do not, and cannot, rely on an early stopping strategy by the adversary (as used in previous impossibility results like [C4]). We also note that our impossibility results hold even for the case of static adversaries (where the set of corrupted parties is fixed ahead of time).

Our results provide an alternative proof to previous impossibility results regarding UC zero-knowledge and UC coin-tossing in the plain model [C1], [CF]. In fact, our results also rule out significant relaxations of these functionalities. We stress, however, that these results do *not* rule out the possibility of securely realizing interesting functionalities like key-exchange, secure message transmission, digital signatures, and public-key encryption in the plain model. Indeed, as noted above, these functionalities can be securely realized in the plain model [C1], [CK1].

Techniques. The proofs of the impossibility results utilize the strong requirements imposed by the UC framework in an essential way. The UC definition follows the standard paradigm of comparing a real protocol execution to an ideal process involving a trusted third party.[3] It also differs in a very important way. The traditional model considered

[2] We note that in the preliminary version of this paper that appeared at EUROCRYPT 2003, it was erroneously stated that impossibility holds if f is unpredictable for P_1 **or** P_2. This is incorrect; unpredictability for both parties is needed.

[3] This well-established paradigm of defining security can be described as follows. First, an ideal execution is defined. In such an execution, the parties hand their inputs to a trusted third party, who simply computes the desired functionality, and hands each party its designated output. Security is then formulated by requiring that the adversary should not be able to do any more harm in a real execution of the protocol than in this ideal execution.

for secure computation includes the parties running the protocol, plus an adversary $\mathcal{A}$ that controls a set of corrupted parties. In the UC framework, an additional adversarial entity called the environment $\mathcal{Z}$ is introduced. This environment generates the inputs to all parties, reads all outputs, and in addition interacts with the adversary in an arbitrary way throughout the computation. A protocol securely computes a function f in this framework if, for any adversary $\mathcal{A}$ that interacts with the parties running the protocol, there exists an ideal process adversary (or "simulator") $\mathcal{S}$ that interacts with the trusted third party, such that no environment $\mathcal{Z}$ can tell whether it is interacting with $\mathcal{A}$ and the parties running the protocol, or with $\mathcal{S}$ in the ideal process.

On a high level, our results are based on the following observation. A central element of the UC definition is that the real and ideal process adversaries $\mathcal{A}$ and $\mathcal{S}$ both interact with the environment $\mathcal{Z}$ in an "on-line" manner. This implies that $\mathcal{S}$ must succeed in simulation while interacting with an external adversarial entity that it cannot "rewind." In a setting without an honest majority or set-up assumptions that can be utilized, it turns out that the simulator $\mathcal{S}$ has *no advantage* over a real participant. Thus, a corrupted party can actually run the code of the simulator.

Given the above observation, we demonstrate our results in two steps. First, in Section 3, we prove a general "technical lemma," asserting that a certain adversarial behavior (which is based on running the code of the simulator) is possible in our model. We then use this lemma in Sections 4 and 5 to prove the respective impossibility results and characterizations mentioned above. (Impossibility for probabilistic functions is proven separately in Section 6, but uses the same ideas.) Loosely speaking, the technical lemma states that a real adversary can do to an honest party "whatever" an ideal process simulator can do. For example, one thing that an ideal process simulator must be able to do is to extract the input used by the real model adversary. Therefore, the lemma states that a real model adversary can also extract the input used by an honest party. This implies that any function that can be securely realized in the UC framework must reveal the input of the participating parties. Thus, only *completely revealing* functions (as described above) can be securely realized.

Impossibility for variants of the UC definition. Different variants of the UC definition have been presented. One aspect where these variants differ is regarding the delivery of messages between the ideal functionality and the parties in the ideal model. Our results hold for all known variants. Another issue where some published variants differ is with respect to the running time of the parties. The original UC definition [C1] models polynomial-time computation as "polynomial in the security parameter." In the revised version of [C1], polynomial-time computation is defined as "polynomial in the input length" [C3]. (The reasons for these changes are discussed in detail in [C3].) Our impossibility results hold for both of these definitions (without any modification to the proofs). Due to its lack of effect on our results, we ignore this technical detail in the presentation. Finally, we note that adaptive corruption strategies are also not used in proving our results; therefore, impossibility also holds when the adversary is limited to static corruptions.

Impossibility for relaxations of the UC definition. Our impossibility results also apply to two relaxations of the UC definition. The first relaxation relates to the complexity of the environment. The UC definition models the environment as a non-uniform Turing machine. A natural relaxation to consider is one where the environment is modeled as a *uniform machine*. (This relaxation was first considered in [HMS], which also showed that the UC composition theorem holds even when the environment is uniform.) We prove impossibility results also for this relaxation (the results obtained are only slight variations of those proven for the case that the environment is non-uniform).

The second relaxation relates to the order of quantifiers between the adversary and the environment in the definition. The UC definition requires that for every adversary $\mathcal{A}$, there should exist a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish a real execution with $\mathcal{A}$ from an ideal process execution with $\mathcal{S}$ (i.e., the order of quantifiers is $\forall\mathcal{A}\exists\mathcal{S}\forall\mathcal{Z}$). Thus, a single simulator $\mathcal{S}$ must successfully simulate for all environments $\mathcal{Z}$. A relaxation of this would allow a different simulator for every environment; i.e., $\forall\mathcal{A}\forall\mathcal{Z}\exists\mathcal{S}$. This difference is only relevant when considering the UC definition of [C1], where polynomial-time computation is defined as "polynomial in the security parameter." In this model it is unknown whether or not the definitions with the original and reversed order of quantifiers are equivalent. In contrast, when modeling polynomial-time computation as "polynomial in the input length," as in the UC definition of [C3], this reversal of quantifiers has *no* effect. That is, the definitions with the original and reversed order of quantifiers are equivalent [C3]. As we have mentioned, our main results are identical for the definition of [C3] and for the original UC definition of [C1]. However, they do not imply impossibility for the relaxed variant of the definition of [C1] where the order of quantifiers is reversed. We therefore show how to extend our impossibility results (with a few mild modifications) to this relaxed variant as well.

Related work. Characterizations of the functions that are securely computable were made in a number of other models and with respect to different notions of security. For example, in the case of *honest-but-curious* parties and information-theoretic privacy, characterizations of the functions that can be computed were found for the two-party case [CK2], [K2], and for boolean functions in the multi-party case [CK2]. In [BMM] the authors consider a setting of computational security against malicious parties where the output is given to only one of the parties, and provide a characterization of the *complete* functions. (A function is complete if given a black-box for computing it, it is possible to securely compute any other function.) Some generalizations were found in [K1]. Similar completeness results for the information-theoretic, honest-but-curious setting are given in [KKMO]. Interestingly, while the characterizations mentioned above are very different from each other, there is some similarity in the type of structures considered in those works and in ours (e.g., the insecure minor of [BMM] and the embedded-OR of [KKMO]). A characterization of the complexity assumptions needed to compute functions in a computational setting (i.e., bounded adversary and arbitrary input sizes) is given in [HNRR], based on a stand-alone, indistinguishability-based definition of security for the honest-but-curious setting.

Subsequent work. Subsequent to this work, our results have formed the basis for other impossibility results. In order to describe these results, we introduce the terminology of *self* and *general* composition. Loosely speaking, a protocol is said to be secure under concurrent general composition if it remains secure when run concurrently with any other arbitrary protocol (thus UC security implies security in this setting). A more restricted type of composition, called concurrent self composition, considers the case that a secure protocol runs concurrently with itself (i.e., it alone is run many times concurrently). We can even further restrict the setting to one where only two parties exist in the network, and they alone run many copies of the protocol.

It has been shown that any definition of security that follows the standard simulatability paradigm and implies security under concurrent general composition implies security under a variant of universal composability where the order of quantifiers is reversed [L1]. Therefore, our impossibility results for this relaxed variant of the UC definition apply to *any* such definition that implies security under concurrent general composition. Next, it was shown that security under concurrent self composition is actually equivalent to security under concurrent general composition [L2]. Therefore, our impossibility results apply even to the setting of concurrent self composition.

Motivated by our broad impossibility results, and their application to concurrent general and self composition, other subsequent works have considered alternative models and definitions of security for which secure protocols can be constructed. For example, see [PS] and [KLP], and the discussion in [C3].

Open questions. Although the impossibility results in this work are quite broad (and even provide full characterizations in some cases), many open questions still remain. First, we do not deal with the case of reactive functionalities at all. Second, we do not deal with functionalities which obtain inputs directly from the adversary and provide outputs directly to the adversary. (Indeed, the ability to "directly communicate with the adversary" was already used to provide meaningful relaxations of functionalities. See for instance the "non-information oracles" of [CK1].) Third, our impossibility results for general deterministic functionalities and for probabilistic functionalities are far from full characterizations. Thus, we still do not have a full and exact understanding of what functions can and cannot be securely realized under the UC definition in the plain model. Given the wide applicability of impossibility results in the UC framework (as described in the previous paragraph), it is important to resolve these questions fully.

2. Review of UC Security

We present a very brief overview of how security is defined in the UC framework. See [C1] for further details.

As in other general definitions (e.g., [GL], [MR], and [B]), the security requirements of a given task (i.e., the functionality expected from a protocol that carries out the task) are captured via a set of instructions for a "trusted party" that obtains the inputs of the participants and provides them with the desired outputs. Informally, a protocol securely carries out a given task if running the protocol with a real adversary amounts to "emulating" an ideal process in which the parties hand their inputs to a trusted party who computes the appropriate functionality and hands their outputs back, without any other interaction. We call the algorithm run by the trusted party the ideal functionality, and describe the interaction in the ideal model to be between the parties and the ideal functionality (with the understanding that what we really mean is the trusted party running this functionality).

In order to prove the universal composition theorem, the notion of emulation in this framework is considerably stronger than in previous ones. Traditionally, the model of computation includes the parties running the protocol and an adversary $\mathcal{A}$ that controls the communication channels and potentially corrupts parties. "Emulating an ideal process" means that for every adversary $\mathcal{A}$ there should exist an "ideal process adversary," or simulator, $\mathcal{S}$ such that the distribution over all parties' inputs and outputs is essentially the same in the ideal and real processes. In the UC framework, an additional entity, called the environment $\mathcal{Z}$, is introduced. The environment generates the inputs to all parties, reads all outputs, and in addition interacts with the adversary in an arbitrary way throughout the computation. A protocol is said to securely realize a given ideal functionality $\mathcal{F}$ if for any "real-life" adversary $\mathcal{A}$ that interacts with the protocol and the environment there exists an "ideal-process adversary" $\mathcal{S}$, such that *no environment* $\mathcal{Z}$ can tell whether it is interacting with $\mathcal{A}$ and parties running the protocol, or with $\mathcal{S}$ and parties that interact with $\mathcal{F}$ in the ideal process. In a sense, here $\mathcal{Z}$ serves as an "interactive distinguisher" between a run of the protocol and the ideal process with access to $\mathcal{F}$. A bit more precisely, let $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ be the ensemble describing the output of environment $\mathcal{Z}$ after interacting with parties running protocol π and with adversary $\mathcal{A}$. Similarly, let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ be the ensemble describing the output of environment $\mathcal{Z}$ after interacting in the ideal process with adversary $\mathcal{S}$ and parties that have access to the ideal functionality $\mathcal{F}$. We note that all entities run in time that is polynomial in the security parameter, denoted by k. In addition, the environment receives an initial input z, and security is required to hold for all such inputs (this makes the environment a non-uniform machine). Security in the UC framework is formalized in the following definition.

Definition 2.1. Let $\mathcal{F}$ be an ideal functionality and let π be a two-party protocol. We say that π securely realizes $\mathcal{F}$ if for every adversary $\mathcal{A}$ there exists an ideal-process adversary $\mathcal{S}$ such that for every environment $\mathcal{Z}$, the ensembles $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ are indistinguishable.

Variants of the UC definition. As we have discussed, our results hold for all known variants of the UC definition. As will become evident from the proofs, the impossibility results are very robust and are not dependent on minor changes to the specific UC formalization.

The plain model. As we have mentioned, our impossibility results here are for the plain model, where no trusted preprocessing phase is assumed. This model is defined as the basic UC model (as described above), together with *authenticated channels*. (Formally, authenticated channels are modeled by considering the $\mathcal{F}_{\text{AUTH}}$-hybrid model, as described in [C1].)

Non-trivial protocols and the requirement to generate output. As we have mentioned above, in the variant of UC that we consider here, the ideal-process adversary can choose when (if ever) to deliver messages that are sent between the parties and the ideal functionality. Consequently, the definition provides no guarantee that a protocol will ever generate output or "return" to the calling protocol. Rather, the definition concentrates on the security requirements *in the case that the protocol generates output*.

A corollary of the above fact is that a protocol that "hangs," never sends any messages and never generates output, securely realizes any ideal functionality. However, such a protocol is clearly not interesting. We therefore use the notion of a non-trivial protocol [CLOS]. Such a protocol has the property that if the real-life adversary delivers all messages and does not corrupt any parties, then the ideal-process adversary also delivers all messages (and does not corrupt any parties). Thus, non-trivial protocols have the *minimal* property that when all participants are honest (and the adversary does not prevent any messages from being delivered), then all parties receive output. Our impossibility results are for non-trivial protocols only.

The UC composition theorem. As mentioned, a UC protocol remains secure under a very general composition operation. In particular, it maintains its security even when run concurrently with other arbitrary protocols that are being run by arbitrary sets of possibly different sets of parties, with possibly related inputs. Thus, UC protocols can be used in modern networks, and security is guaranteed. It is therefore of great importance to understand what functions can and cannot be securely realized under this definition. See [C1] for more details.

3. The Main Technical Lemma for Deterministic Functions

This section contains the main technical lemma that is used for proving our impossibility results. Loosely speaking, the lemma describes an "attack" that is possible against any UC protocol that securely realizes a deterministic function f. This lemma itself does not claim impossibility of securely realizing any functionality. However, in Section 4, we use it for proving all our impossibility results.

Notation. We consider deterministic, polynomial-time computable functions $f\colon X \times X \to \{0,1\}^* \times \{0,1\}^*$, where $X \subseteq \{0,1\}^*$ is an arbitrary, possibly infinite, domain (for simplicity of notation, we assume that both parties' inputs are from the same domain; changing this makes no difference to our results). We note that functions that depend on the security parameter can be derived by defining $X = \mathsf{N} \times \{0,1\}^*$. These functions have two outputs, one for each party. We denote $f = (f_1, f_2)$ where f_1 denotes the first party's output and f_2 denotes the second party's output.

Motivation. To motivate the lemma, recall the way an ideal-model simulator typically works. Such a simulator interacts with an ideal functionality by sending it an input (in the name of the corrupted party) and receiving back an output. Since the simulated view of the corrupted party is required to be indistinguishable from its view in a real execution, it must hold that the input sent by the simulator to the ideal functionality corresponds to the input that the corrupted party (implicitly) uses. Furthermore, the corrupted party's output from the protocol simulation must correspond to the output received by the simulator from the ideal functionality. That is, such a simulator must be able to *"extract"* the input used by the corrupted party, in addition to causing the corrupted party to *output* a value that corresponds to the output received by the simulator from the ideal functionality.

We show that, essentially, a malicious P_2 can do "whatever" the simulator can do. That is, consider the simulator that exists when P_1 is corrupted. This simulator can extract P_1's input and can also cause its output to be consistent with the output from the ideal functionality. Therefore, P_2 (when interacting with an *honest* P_1) can also extract P_1's input and cause its output to be consistent with an ideally generated output. Indeed, P_2 succeeds in doing this by internally running the ideal-process simulator for P_1. In other models of secure computation, this cannot be done because a simulator typically has some additional "power" that a malicious party does not. (This power is usually the ability to rewind a party or to hold its description or code.) Thus, we actually show that in the *plain model* and *without an honest majority*, the simulator for the UC setting has no power beyond what a real (adversarial) party can do in a real execution. This enables a malicious P_2 to run the simulator as required. We now describe the above-mentioned strategy of P_2.

Strategy description for P_2. The malicious P_2 that we construct internally runs two separate machines (or entities): P_2^a and P_2^b. Entity P_2^a interacts with (the honest) P_1 and runs the simulator that is guaranteed to exist for P_1, as described above. In contrast, entity P_2^b emulates the ideal functionality for the simulator that is run by P_2^a. Loosely speaking, P_2^a first "extracts" the input used by P_1. Entity P_2^a then hands this input to P_2^b, who computes the function output and hands it back to P_2^a. Entity P_2^a then continues with the emulation, and causes P_1 to output a value that is consistent with the input that is chosen by P_2^b. We now formally define this strategy of P_2. We begin by defining the structure of this adversarial attack, which we call a "split adversarial strategy," and then proceed to define what it means for such a strategy to be "successful."

Definition 3.1 (Split Adversarial Strategy). Let $f\colon X \times X \to \{0,1\}^* \times \{0,1\}^*$ be a polynomial-time function where f_1 and f_2 denote the first and second outputs of f, respectively, and let Π_f be a protocol. Let $X_2 \subseteq X$ be a polynomial-size subset of inputs (i.e., $|X_2| = \text{poly}(k)$, where k is the security parameter), and let $x_2 \in X_2$. Then a corrupted party P_2 is said to run a split adversarial strategy if it consists of machines P_2^a and P_2^b such that:

1. Upon input (X_2, x_2), party P_2 internally gives the machine P_2^b the input pair (X_2, x_2).
2. An execution between (an honest) P_1 running Π_f and $P_2 = (P_2^a, P_2^b)$ works as follows:
   (a) P_2^a interacts with P_1 according to some specified strategy.
   (b) At some stage of the execution P_2^a hands P_2^b a value x_1'.
   (c) When P_2^b receives x_1' from P_2^a, it computes $y_1' = f_1(x_1', x_2')$ for some $x_2' \in X_2$ of its choice.[4]
   (d) P_2^b hands P_2^a the value y_1', and P_2^a continues interacting with P_1.

[4] The choice of x_2' can depend on the values of both x_1' and x_2 and can be chosen by P_2^b according to any efficient strategy. The fact that x_2' must come from the polynomial-size subset of inputs X_2 is needed for proving the existence of a *successful* split adversarial strategy, as defined below.

Informally speaking, a split adversarial strategy is said to be *successful* if the value x_1' procured by P_2^a is "equivalent to" (the honest) P_1's input x_1 with respect to f_2. That is, the output of P_2, when computed according to f_2, is the same whether x_1 or x_1' is used. (Note that x_1' may differ from x_1 with respect to P_1's output, but we consider the effect on P_2's output only.) Furthermore, P_2^a should succeed in causing P_1 to output the value $y_1 = f_1(x_1, x_2')$. That is, the output of P_1 should be consistent with the value x_2' chosen by P_2^b.

Definition 3.2 (Successful Strategies). Let f be a polynomial-time function and let Π_f be a protocol, as in Definition 3.1. Furthermore, let k be the security parameter and let $\mathcal{Z}$ be an environment who hands an input $x_1 \in X$ to P_1 and a pair (X_2, x_2) to P_2, where $X_2 \subseteq X$, $|X_2| = \text{poly}(k)$, and $x_2 \in_R X_2$.[5] Then a split adversarial strategy for a malicious P_2 is said to be successful if for every $\mathcal{Z}$ as above and every input z to $\mathcal{Z}$, the following two conditions hold in a real execution of P_2 with $\mathcal{Z}$ and an honest P_1:

1. The value x_1' output by P_2^a in step 2(b) of Definition 3.1 is such that for *every* $x_2 \in X_2$, $f_2(x_1', x_2) = f_2(x_1, x_2)$.
2. P_1 outputs $f_1(x_1, x_2')$, where x_2' is the value chosen by P_2^b in step 2(c) of Definition 3.1.

Loosely speaking, the lemma below states that a successful split adversarial strategy exists for any protocol that securely realizes a two-party function in the plain model. In Section 4 we show that the existence of successful split adversarial strategies rules out the possibility of securely realizing large classes of functions. We are now ready to state the lemma:

Lemma 3.3 (Main Technical Lemma). *Let f be a polynomial-time two-party function, and let $\mathcal{F}_f$ be the two-party ideal functionality that receives x_1 from P_1 and x_2 from P_2, and hands them back their respective outputs $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$. If $\mathcal{F}_f$ can be securely realized in the plain model by a non-trivial protocol Π_f,[6] then there exists a machine P_2^a such that for every machine P_2^b of the form described in Definition* 3.1, *the split adversarial strategy for $P_2 = (P_2^a, P_2^b)$ is* successful, *except with negligible probability.*

Proof. The intuition behind the proof is as follows. If $\mathcal{F}_f$ can be securely realized by a protocol $\Pi_{\mathcal{F}}$, then this implies that for any real-life adversary $\mathcal{A}$ (and environment $\mathcal{Z}$), there exists an ideal-process adversary (or "simulator") $\mathcal{S}$. As we have mentioned, the simulator $\mathcal{S}$ interacts with the ideal process and must hand it the input that is (implicitly) used by the adversary while controlling the corrupted party. In other words, $\mathcal{S}$ must be able to *extract* the input used by $\mathcal{A}$. The key point in the proof is that $\mathcal{S}$ must essentially accomplish this extraction while running a "straight-line black-box" simulation. (This

[5] Formally, $\mathcal{Z}$ can only write a single input x_2 on the input tape of P_2. However, in this case P_2 is corrupted and so $\mathcal{Z}$ can pass it the set X_2 using the communication between the adversary and the environment.

[6] Recall that a non-trivial protocol is such that if the real model adversary corrupts no party and delivers all messages, then so does the ideal model adversary. This rules out the trivial protocol that does not generate output. See Section 2 for details.

just means that $\mathcal{S}$ cannot rewind $\mathcal{A}$ and also has no access to its code. Stated differently, $\mathcal{S}$ interacts with $\mathcal{A}$ just like real parties interact in a protocol execution.) This is shown as follows. Consider the case that $\mathcal{Z}$ and $\mathcal{A}$ cooperate so that $\mathcal{Z}$ runs the adversarial strategy and $\mathcal{A}$ does nothing but forward messages between $\mathcal{Z}$ and the honest party. In this case, $\mathcal{S}$ must extract the input that is implicitly used by $\mathcal{Z}$ (since $\mathcal{A}$ actually does nothing). However, $\mathcal{S}$ interacts with $\mathcal{Z}$ like in a real interaction (i.e., in a "straight-line black-box" manner). Therefore, $\mathcal{S}$ must successfully extract even in such a scenario. To complete the intuition for success for item (1) of Definition 3.2, consider the case that $\mathcal{Z}$'s adversarial strategy is just to follow the protocol instructions of Π_f for the honest P_1. Then we have that $\mathcal{S}$ can extract the honest P_1's output in a real protocol interaction (because this interaction with P_1 is the same as with $\mathcal{Z}$, where successful extraction is guaranteed). Thus, machine P_2^a just consists of running $\mathcal{S}$ with P_1, in order to obtain P_1's input. The above intuition relates to success under item (1) of Definition 3.2 (i.e., input extraction). Similar arguments are also used to prove item (2). We proceed to the formal proof.

Assume that $\mathcal{F}_f$ can be securely realized by a protocol Π_f. Then, for every real-life adversary $\mathcal{A}$, there exists an ideal-process adversary/simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish between an execution of the ideal process with $\mathcal{S}$ and $\mathcal{F}_f$ and an execution of the real protocol Π_f with $\mathcal{A}$. We now define a specific adversary $\mathcal{A}$ and environment $\mathcal{Z}$. The adversary $\mathcal{A}$ controls party P_1 and is a "dummy adversary" who does nothing except delivers all messages that it receives from P_2 to $\mathcal{Z}$, and delivers all messages that it receives from $\mathcal{Z}$ to P_2. That is, $\mathcal{A}$ merely acts as a bridge that passes messages between $\mathcal{Z}$ and P_2 (the first box in Fig. 1 corresponds to this setting). Next we define $\mathcal{Z}$. Let X_2 be some polynomial-size set of inputs (chosen by $\mathcal{Z}$), and let (x_1, x_2) be P_1 and P_2's respective inputs as decided by $\mathcal{Z}$, where $x_2 \in_R X_2$. Then $\mathcal{Z}$ writes x_2 on P_2's input tape and plays the role of the honest P_1 on input x_1. That is, $\mathcal{Z}$ runs P_1's protocol instructions in Π_f on input x_1 and the incoming messages that it receives from $\mathcal{A}$ (which are in turn received from P_2). The messages that $\mathcal{Z}$ passes to $\mathcal{A}$ are exactly the messages as computed by an honest P_1 according to Π_f. At the conclusion of the execution of Π_f, the environment $\mathcal{Z}$ obtains some output, as defined by the protocol specification for P_1 that $\mathcal{Z}$ runs internally; we call this $\mathcal{Z}$'s local P_1-output. $\mathcal{Z}$ then reads P_2's output tape and outputs 1 if and only if $\mathcal{Z}$'s local P_1-output equals $f_1(x_1, x_2)$ *and*, in addition, P_2's output equals $f_2(x_1, x_2)$; see the first box in Fig. 1. Observe that in the real-life model $\mathcal{Z}$ outputs 1 with probability negligibly close to 1. This is because such an execution of Π_f, with the above $\mathcal{Z}$ and $\mathcal{A}$, looks exactly like an execution between two honest parties P_1 and P_2 upon inputs x_1 and x_2, respectively. Furthermore, all messages between these (honest) parties are delivered by the adversary. Intuitively, in such an execution, both parties must receive output, and this output is exactly $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$, respectively. Formally, this holds by the assumption that Π_f is a *non-trivial* protocol. Thus, in the ideal process, the parties must receive $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$, respectively. By the indistinguishability between the real and ideal processes, the same must also hold in a real execution (except with negligible probability). We conclude that $\mathcal{Z}$ outputs 1 in the real-life model, except with negligible probability.

By the assumption that Π_f securely realizes $\mathcal{F}_f$, there exists an ideal process simulator $\mathcal{S}$ for the *specific* $\mathcal{A}$ and $\mathcal{Z}$ described above (the second box in Fig. 1 corresponds to the setting in which $\mathcal{S}$ operates). We use $\mathcal{S}$ to obtain the required strategy for P_2^a (the

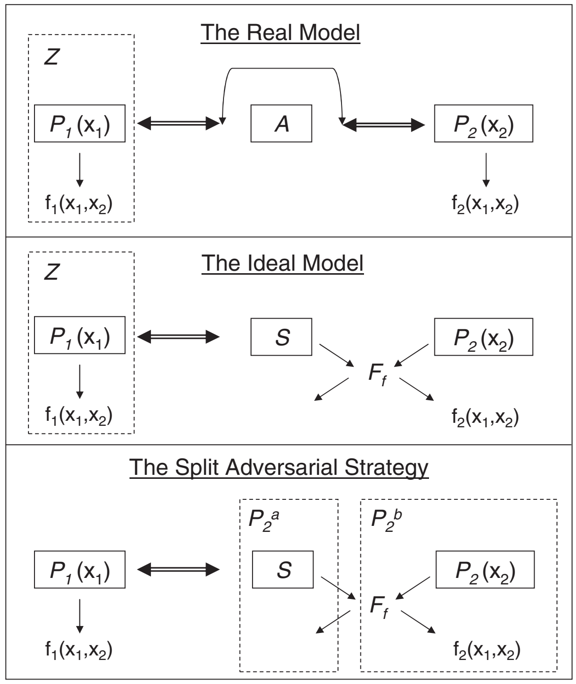

Fig. 1. Three stages in the proof of Lemma 3.3.

third box in Fig. 1 shows how this strategy is derived). Machine P_2^a invokes simulator $\mathcal{S}$ and *emulates* an ideal process execution of $\mathcal{S}$ with $\mathcal{F}_f$ and the above $\mathcal{Z}$. That is, every message that P_2^a receives from the honest P_1 in the real execution of Π_f, it forwards to $\mathcal{S}$ as if $\mathcal{S}$ received it from $\mathcal{Z}$. Likewise, every message that $\mathcal{S}$ sends to $\mathcal{Z}$ in the emulation, P_2^a forwards to P_1 in the real execution. When $\mathcal{S}$ outputs a value x_1' that it intends to send to $\mathcal{F}_f$, entity P_2^a hands it to P_2^b. Then, when P_2^a receives a value y_1' back from P_2^b, it passes this to $\mathcal{S}$ as if it was sent from $\mathcal{F}_f$, and continues with the emulation. (Recall that this value y_1' is computed by P_2^b and equals $f_1(x_1', x_2')$, for some $x_2' \in X_2$ of P_2^b's choice.)

We now prove that, except with negligible probability, the above P_2^a is such that for every P_2^b of the form described in Definition 3.1, party $P_2 = (P_2^a, P_2^b)$ is a successful split adversarial strategy. That is, we prove that items (1) and (2) from Definition 3.2 hold with respect to this P_2. We begin by proving that, except with negligible probability, the value x_1' output by P_2^a is such that for every $x_2 \in X_2$, $f_2(x_1', x_2) = f_2(x_1, x_2)$. In order to demonstrate this, we first claim that up until the point that P_2^a outputs x_1', the real execution between P_1 (with input x_1) and $P_2 = (P_2^a, P_2^b)$ is *identical* to the ideal process with the above environment $\mathcal{Z}$ (with input x_1), the ideal functionality $\mathcal{F}_f$ and the simulator $\mathcal{S}$. This follows from the fact that in the ideal process, $\mathcal{Z}$ plays the honest P_1 strategy upon input x_1. Therefore, the messages that $\mathcal{S}$ receives from $\mathcal{Z}$ are distributed exactly like the messages that P_2^a forwards to $\mathcal{S}$ from the honest P_1 in the real execution between P_1 and $P_2 = (P_2^a, P_2^b)$. Furthermore, $\mathcal{S}$ is given the same initial state in both scenarios. (Note that, in general, in the ideal process $\mathcal{S}$ is given the strategy and input of $\mathcal{A}$. However, the adversary $\mathcal{A}$ here has no input and follows a straightforward deterministic strategy. Thus, $\mathcal{S}$ is exactly the same in the ideal process and in the emulation by $P_2 = (P_2^a, P_2^b)$.) We conclude that the distribution over the messages sent in both processes is identical until the point that $\mathcal{S}$ outputs x_1'. (See Fig. 1 that highlights the equivalence between the ideal process and a real execution with the split adversary strategy.) We now show that in the ideal process with $\mathcal{F}_f$ and $\mathcal{Z}$, simulator $\mathcal{S}$ must obtain and send $\mathcal{F}_f$ an input x_1' such that for every $x_2 \in X_2$, $f_2(x_1', x_2) = f_2(x_1, x_2)$, except with negligible probability. This suffices for proving item (1) because if $\mathcal{S}$ obtains such an x_1' in the ideal process, then it also obtains it in the emulation with $P_2 = (P_2^a, P_2^b)$ where the distribution over all messages sent is identical.) This can be seen as follows. Assume, by contradiction, that with non-negligible probability x_1' is such that for some $\tilde{x}_2 \in X_2$, $f_2(x_1', \tilde{x}_2) \neq f_2(x_1, \tilde{x}_2)$. Now, if in an ideal execution, P_2 has input $\tilde{x}_2$ and $\mathcal{S}$ sends x_1' to $\mathcal{F}_f$, then P_2 outputs $f_2(x_1', \tilde{x}_2) \neq f_2(x_1, \tilde{x}_2)$. By the specification of $\mathcal{Z}$, when this occurs $\mathcal{Z}$ outputs 0. Now, recall that X_2 is of polynomial size and that P_2's input is uniformly chosen from the set X_2. Furthermore, the probability that $\mathcal{S}$ sends x_1' is independent of the choice of x_2 for P_2 (because $\mathcal{S}$ has no information on x_2 when it sends x_1'). Therefore, the probability that $\mathcal{Z}$ outputs 0 is at least $1/|X_2|$ times the probability that x_1' as output by $\mathcal{S}$ is such that for some $\tilde{x}_2$, $f_2(x_1', \tilde{x}_2) \neq f_2(x_1, \tilde{x}_2)$. Thus, $\mathcal{Z}$ outputs 0 in the ideal process with non-negligible probability. However, we have already argued above that in a real protocol execution, $\mathcal{Z}$ outputs 0 with at most negligible probability. Thus, $\mathcal{Z}$ distinguishes the real and ideal executions, contradicting the security of the protocol. We conclude that except with negligible probability, item (1) of Definition 3.2 holds.

We proceed to prove item (2) of Definition 3.2. Assume by contradiction, that in the emulation with $P_2 = (P_2^a, P_2^b)$, party P_1 outputs $\tilde{y}_1 \neq f_1(x_1, x_2')$ with non-negligible probability. First, consider the following thought experiment: Modify P_2^b so that instead of choosing x_2' as some function of x_1' and x_2, it chooses $\tilde{x}_2 \in_R X_2$ instead; denote this modified party $\tilde{P}_2^b$. It follows that with probability $1/|X_2|$, the value chosen by the modified $\tilde{P}_2^b$ *equals* the value chosen by the unmodified P_2^b. Therefore, the probability that P_1 outputs $\tilde{y}_1 \neq f_1(x_1, \tilde{x}_2)$ in an emulation with the modified $\tilde{P}_2 = (P_2^a, \tilde{P}_2^b)$ is at least $1/|X_2|$ times the (non-negligible) probability that this occurred with the unmodified P_2^b. Since X_2 is of polynomial size, we conclude that P_1 outputs $\tilde{y}_1 \neq f_1(x_1, \tilde{x}_2)$ with non-negligible probability in an emulation with the modified $\tilde{P}_2 = (P_2^a, \tilde{P}_2^b)$. Next, we claim that the view of $\mathcal{S}$ in the ideal process with $\mathcal{Z}$ and $\mathcal{F}_f$ is identical to its view in the modified emulation by $\tilde{P}_2 = (P_2^a, \tilde{P}_2^b)$. The fact that this holds until $\mathcal{S}$ outputs x_1' was shown above in the proof of item (1). The fact that it holds from that point on follows from the observation that in the emulation by $\tilde{P}_2 = (P_2^a, \tilde{P}_2^b)$, simulator $\mathcal{S}$ receives $f_1(x_1', \tilde{x}_2)$ where $\tilde{x}_2 \in_R X_2$. However, this is exactly the same as it receives in an ideal execution (where $\mathcal{Z}$ chooses $x_2 \in_R X_2$ and gives it to the honest P_2). It follows that the distribution of messages received by P_1 in a real execution with $\tilde{P}_2 = (P_2^a, \tilde{P}_2^b)$ is exactly the same as the distribution of messages received by $\mathcal{Z}$ from $\mathcal{S}$ in the ideal process. Thus, $\mathcal{Z}$'s local P_1-output in the ideal process is identically distributed to P_1's output in the emulation with $\tilde{P}_2 = (P_2^a, \tilde{P}_2^b)$. Since with non-negligible probability, in this emulation P_1 outputs $\tilde{y}_1 \neq f_1(x_1, \tilde{x}_2)$, we have that with non-negligible probability, $\mathcal{Z}$'s local P_1-output in the ideal process is also not equal to $y_1 = f_1(x_1, x_2)$, where x_1 and x_2 are the inputs chosen by $\mathcal{Z}$ (notice that x_2 and $\tilde{x}_2$ are identically distributed). Therefore, $\mathcal{Z}$ outputs 0 in the ideal process with non-negligible probability (since $\mathcal{Z}$ outputs 1 only if its local P_1-output equals $f_1(x_1, x_2)$ where (x_1, x_2) are the inputs that it chose). Thus, $\mathcal{Z}$ distinguishes the real and ideal processes. This completes the proof. □

Split adversarial strategies for P_1. Definitions 3.1 and 3.2 and Lemma 3.3 can all be formulated for P_1 instead of P_2, and the proof is the same (modulo switching the roles of P_1 and P_2).

Relaxations of UC. As we have mentioned, we also prove our impossibility results for two relaxations of the UC definition. Consider first the relaxation obtained by reversing the order of quantifiers between $\mathcal{S}$ and $\mathcal{Z}$. It follows that Lemma 3.3 holds without any modification for this relaxation; this is the case because in the proof, $\mathcal{Z}$ is a *fixed* strategy. Therefore, $\mathcal{S}$ only needs to simulate successfully for the specific environment $\mathcal{Z}$ described in the proof. Next, consider the relaxation of the definition obtained by modeling the environment $\mathcal{Z}$ as a uniform machine. In this case a slight variant of Lemma 3.3 is obtained; the only difference being that the set X_2 and the inputs x_1 and x_2 chosen by $\mathcal{Z}$ must be uniformly generated.

4. Impossibility Results for Deterministic Functions

In this section we use Lemma 3.3 in order to prove a number of impossibility results. The results apply to functions with different combinatorial or other properties. Before continuing, we define the following terminology. A function $f = (f_1, f_2)$ is called same-output if $f_1 = f_2$ (i.e., both parties receive the same output). Similarly, we say that a function is single-output if either f_1 or f_2 is the constant function outputting the empty string λ (i.e., only one party receives output). Finally, we say that a function is single-input if it depends on the input of only one party. Our impossibility results in this section relate to functions of the above three classes, as well as to general functions (where both parties receive possibly different outputs that possibly depend on both inputs). Note that we actually obtain *full characterizations* of feasibility for the above three classes of functions (with the limitation that the characterization for single-output functions is only for the case that the domain of the function is finite); these are presented in Section 5. In contrast, we do not obtain full characterizations for general functions; the impossibility results that apply for this general case appear in Section 4.4.

4.1. *Single-Input Functions which Are Not Efficiently Invertible*

This section considers functions that depend on only one party's input. We show that if such a function is not efficiently invertible, then it cannot be securely realized in the UC framework. Intuitively, a function is efficiently invertible if there exists a machine that can find preimages of $f(x)$, when x is chosen according to any efficiently samplable distribution.

Definition 4.1. A polynomial-time function $f : X \to \{0, 1\}^*$ is efficiently invertible if there exists a probabilistic polynomial-time inverting machine M such that for every distribution $\hat{X} = \{\hat{X}_k\}$ over X that is samplable in probabilistic polynomial-time by a non-uniform Turing machine, every polynomial $p(\cdot)$, and all sufficiently large k's,

$$\Pr_{x \leftarrow \hat{X}_k}[M(1^k, f(x)) \in f^{-1}(f(x))] > 1 - \frac{1}{p(k)}.$$

Discussion. A few remarks regarding Definition 4.1: First, note that every function f over a finite domain X is efficiently invertible. Second, note that a function that is *not* efficiently invertible is not necessarily even weakly one-way. This is because the definition of invertibility requires the existence of an inverter that works for *all* distributions, rather than only for the uniform distribution (as in the case of one-way functions). In fact, a function that is not efficiently invertible can be constructed from any NP-language L that is not in $\mathcal{BPP}$, as follows. Let R_L be an NP-relation for L, i.e., $x \in L$ if and only if there exists a w such that $R_L(x, w) = 1$. Then define $f_L(x, w) = (x, R_L(x, w))$. It then follows that f_L is not efficiently invertible unless $L \in \mathcal{BPP}$. (This argument holds only when the distributions $\hat{X}$ are allowed to be efficiently samplable by a non-uniform Turing machine, but not necessarily efficiently samplable by a uniform one. In this case, for every k the distribution can just output a fixed input (x, w), $|x| = k$. Then, since the inverting machine M must be able to invert for all such distributions, we have that it must be able to invert all inputs, except with negligible probability. It therefore follows that $L \in \mathcal{BPP}$.) Finally, note that the function f_L as defined above corresponds in fact to the ideal zero-knowledge functionality for the language L. That is, the ideal functionality $\mathcal{F}_{f_L}$ as defined above is exactly the ideal zero-knowledge functionality $\mathcal{F}_{\text{ZK}}^{R_L}$ for relation R_L, as defined in [C1] and [CLOS]. Consequently, the impossibility theorem below (Theorem 4.2) provides, as a special case, an alternative proof that $\mathcal{F}_{\text{ZK}}^{R_L}$ cannot be realized unless $L \in \mathcal{BPP}$ [C1].

We now prove impossibility for functions that are not efficiently invertible.

Theorem 4.2. *Let $f\colon X \to \{0, 1\}^*$ be a polynomial-time function and let $\mathcal{F}_f$ be a functionality that receives x from P_1 and sends $f(x)$ to P_2. If f is not efficiently invertible, then $\mathcal{F}_f$ cannot be securely realized in the plain model by a non-trivial protocol.*

Proof. The idea behind the proof is that according to Lemma 3.3, a real adversary can always extract the input of the other party by running a split adversarial strategy. Therefore, an ideal adversary can also extract this input. Since this ideal adversary extracts the input based only on the output (because it works in the ideal process), we are able to use it to construct an inverting machine M for the function f, as described in Definition 4.1. We therefore conclude that if $\mathcal{F}_f$ can be securely realized, then f is efficiently invertible.

Let $f\colon X \to \{0, 1\}^*$ be a polynomial-time function and assume that there exists a protocol Π_f that securely realizes f. Then consider a real execution of Π_f with an honest P_1 and an adversary $\mathcal{A}$ who corrupts P_2. The environment $\mathcal{Z}$ for this execution, with security parameter k, samples a value x from some distribution $\hat{X}_k$ and hands it to P_1. Then $\mathcal{Z}$ outputs 1 if and only if, at sometime during the execution, it receives a value x' from $\mathcal{A}$ where $f(x') = f(x)$. This concludes the description of $\mathcal{Z}$. We now describe the real-life adversary $\mathcal{A}$. Adversary $\mathcal{A}$ runs a successful split adversarial strategy for P_2

(the input chosen by P_2^b in this case is the empty string because f is single-input). By Lemma 3.3, such a successful strategy exists. Now, at some stage of the execution, P_2^a hands P_2^b a value x'. When $\mathcal{A}$ obtains this value x' from P_2^a, it hands it to $\mathcal{Z}$ and halts. This concludes the description of $\mathcal{A}$.

We now show that in the real-life model with this $\mathcal{A}$, the environment $\mathcal{Z}$ outputs 1 except with negligible probability. This follows from item (1) of Definition 3.2 which implies that the x' obtained by $\mathcal{A}$ is such that $f(x') = f(x)$ except with negligible probability. We note that the description of $\mathcal{Z}$ refers to "some" distribution $\hat{X}$ over the inputs. We do not specify this further; rather, a different environment $\mathcal{Z}$ is considered for every different distribution $\hat{X}$.

Next, consider an ideal execution with the same $\mathcal{Z}$ and with an ideal-process simulator $\mathcal{S}$ for the above $\mathcal{A}$. Clearly, in such an ideal execution $\mathcal{S}$ receives $f(x)$ only (because it has no input and just receives the output of the corrupted party P_2). Nevertheless, $\mathcal{S}$ succeeds in handing $\mathcal{Z}$ a value x' such that $f(x') = f(x)$ except with negligible probability; otherwise, $\mathcal{Z}$ would distinguish a real execution from an ideal one.

We now use $\mathcal{S}$ to construct an inverting machine M for f. Given $y = f(x)$, M runs $\mathcal{S}$, gives it y as if it was sent by $\mathcal{F}_f$, and outputs whatever value $\mathcal{S}$ hands to $\mathcal{Z}$. The fact that M is a valid inverting machine follows from the above argument. That is, for every environment $\mathcal{Z}$, the simulator $\mathcal{S}$ causes $\mathcal{Z}$ to output 1 except with negligible probability. Therefore, for every efficiently samplable distribution $\hat{X}$, the machine M succeeds in outputting x' such that $f(x') = f(x)$, except with negligible probability. Thus, f is efficiently invertible, concluding the proof. □

Relaxations of UC. If the environment $\mathcal{Z}$ is uniform, then the distributions $\hat{X}$ over X must also be uniform. There is no other difference to Theorem 4.2 for this relaxation. However, there are significant differences for the relaxation of UC obtained by reversing the order of quantifiers between $\mathcal{S}$ and $\mathcal{Z}$. This is because the above proof assumes that the same simulator $\mathcal{S}$ must work for all environments. That is, $\mathcal{S}$ succeeds in obtaining x' when interacting with *every* environment $\mathcal{Z}$ that uses some distribution $\hat{X}$ to choose inputs. Therefore, M can invert for *every* distribution $\hat{X}$ demonstrating that f is efficiently invertible. However, if a different simulator $\mathcal{S}$ could be provided for every $\mathcal{Z}$, then we would only obtain that for every distribution $\hat{X}$ there exists a machine M which can invert inputs from $\hat{X}$. This does not imply efficient inversion as formulated in Definition 4.1, where a single machine must work for all distributions.

Nevertheless, we do obtain the following impossibility result: If f is such that there exists a *single* polynomial-time samplable distribution $\hat{X}$ (called a "hard distribution") for which it is hard for *all* efficient machines M to invert $f(\hat{X})$, then f cannot be securely realized, even according to the relaxed order of quantifiers for UC.[7] This follows because $\mathcal{Z}$ can choose x according to this hard distribution. Then no simulator $\mathcal{S}$ can invert the inputs chosen by $\mathcal{Z}$. We remark that an example of such a function f is a (weak) one-

[7] More formally, let f be a polynomial-time single-input function and let $\hat{X} = \{\hat{X}_k\}_{k\in\mathrm{N}}$ be a family of distributions that are efficiently samplable by non-uniform Turing machines, so that for every machine M there exists a polynomial p_M such that for all sufficiently large k's $\Pr[M(1^k, f(\hat{X}_k)) \in f^{-1}(f(\hat{X}_k))] < 1 - 1/p_M(k)$. Then f cannot be securely realized even when the definition of UC is relaxed by reversing the quantifiers between $\mathcal{S}$ and $\mathcal{Z}$.

Table 1. An insecure minor (here $b \neq c$).

	α_2	α_2'
α_1	a	b
α_1'	a	c

way function (note that the ideal zero-knowledge functionality over hard-on-the-average languages is weakly one-way).

4.2. *Same-Output Functions with Insecure Minors*

This section contains an impossibility result for *same-output* functions with a special combinatorial property, namely those functions containing an insecure minor. Insecure minors have been used in the past to show non-realizability results in a different context of information-theoretic security [BMM]. Since same-output functions are considered, we drop the $f = (f_1, f_2)$ notation and consider $f\colon X \times X \to \{0, 1\}^*$.

A same-output function $f : X \times X \to \{0, 1\}^*$ is said to contain an insecure minor if there exist inputs $\alpha_1, \alpha_1', \alpha_2$, and α_2' such that $f_2(\alpha_1, \alpha_2) = f_2(\alpha_1', \alpha_2)$ and $f_2(\alpha_1, \alpha_2') \neq f_2(\alpha_1', \alpha_2')$; see Table 1.

In the case of boolean functions, the notion of an insecure minor boils down to the so-called "embedded-OR"; see, e.g., [KKMO]. Such a function has the property that when P_2 has input α_2, then party P_1's input is "hidden" (i.e., given $y_2 = f_2(x_1, \alpha_2)$, it is impossible for P_2 to know whether P_1's input, x_1, was α_1 or α_1'). Furthermore, α_1 and α_1' are not "equivalent," in that when P_2 has α_2' for input, then the function value when P_1 has $x_1 = \alpha_1$ differs from its value when P_1 has $x_1 = \alpha_1'$ (because $f_2(\alpha_1, \alpha_2') \neq f_2(\alpha_1', \alpha_2')$). We stress that there is no requirement that $f_2(\alpha_1, \alpha_2) \neq f_2(\alpha_1, \alpha_2')$ or $f_2(\alpha_1', \alpha_2) \neq f_2(\alpha_1', \alpha_2')$ (i.e., in Table 1, a may equal b or c, but clearly not both).

We now show that if a same-output function f contains an insecure minor, then f cannot be securely realized in the plain model.

Theorem 4.3. *Let f be a polynomial-time same-output two-party function containing an insecure minor, and let $\mathcal{F}_f$ be the two-party ideal functionality that receives x_1 and x_2 from P_1 and P_2 respectively, and hands both parties $f(x_1, x_2)$. Then $\mathcal{F}_f$ cannot be securely realized in the plain model by a non-trivial protocol.*

Proof. The idea behind the proof is as follows. Consider a function f with an insecure minor as in Table 1. Then, in the case that P_2's input equals α_2, party P_2 cannot know if P_1's input was α_1 or α_1' (because in both cases the output of the function is a). However, by Lemma 3.3, if $\mathcal{F}_f$ can be securely realized, then a successful split adversarial strategy can be used by P_2 to (almost) always obtain P_1's input. Thus, we conclude that $\mathcal{F}_f$ cannot be securely realized.[8]

[8] We note that a function with an insecure minor may be completely revealing (as in Definition 4.5 of Section 4.4). This is because P_2 can always choose to just input α_2', and it will then always be able to distinguish the case that P_1's input was α_1 from the case that its input was α_1'. In our proof here, we utilize the fact that f is same-output in order to show that P_2 *cannot* arbitrarily choose its own input. Specifically, since P_1 receives output as well, the environment is able to check whether or not P_2 used its prescribed input (α_2 or α_2'), by looking at P_1's output. This forces P_2 to use its prescribed input. Then, in the case that this input is α_2, party P_2 is unable to know whether P_1's input was α_1 or α_1'.

Formally, let f be a polynomial-time same-output two-party function, and let α_1, α_1', α_2, α_2' form an insecure minor in f. Assume by contradiction that $\mathcal{F}_f$ can be securely realized by a non-trivial protocol Π_f. Then consider a real execution of Π_f with an honest P_1 and an adversary $\mathcal{A}$ who corrupts P_2. The environment $\mathcal{Z}$ for this execution chooses a pair of inputs (x_1, x_2) where $x_1 \in_R \{\alpha_1, \alpha_1'\}$ and $x_2 \in_R \{\alpha_2, \alpha_2'\}$. (Since $\mathcal{Z}$ receives auxiliary input, we can assume that it knows the insecure minor $\alpha_1, \alpha_1', \alpha_2, \alpha_2'$.) $\mathcal{Z}$ then writes x_1 and x_2 on P_1 and P_2's respective input tapes. Furthermore, $\mathcal{Z}$ passes $\mathcal{A}$ the set $X_2 = \{\alpha_2, \alpha_2'\}$. Finally, $\mathcal{Z}$ outputs 1 if and only if the output of P_1 equals $f(x_1, x_2)$ and, in addition, $\mathcal{Z}$ receives x_1 from $\mathcal{A}$ at the conclusion of the execution. This concludes the description of $\mathcal{Z}$. We now describe the real-life adversary $\mathcal{A}$. Adversary $\mathcal{A}$ runs a split adversarial strategy for P_2. The entity P_2^b in this case simply chooses $x_2' = x_2$ (i.e., it does not change the input that it received). By Lemma 3.3, a successful strategy for this P_2^b exists. Now, since P_2 is successful, P_2^a must hand P_2^b a value x_1' such that for every $x_2 \in X_2$, $f_2(x_1', x_2) = f_2(x_1, x_2)$. $\mathcal{A}$ runs the entire successful strategy of P_2. In addition, when $\mathcal{A}$ sees the value x_1' output by P_1^a, it computes $y = f_2(x_1', \alpha_2')$. Then, if $y = f_2(\alpha_1, \alpha_2')$, it concludes that $x_1 = \alpha_1$ and sends α_1 to $\mathcal{Z}$. However, if $y = f_2(\alpha_1', \alpha_2')$, it concludes that $x_1 = \alpha_1'$ and sends α_1' to $\mathcal{Z}$. This completes the description of $\mathcal{A}$.

We now show that in the real-life model with this $\mathcal{A}$, the environment $\mathcal{Z}$ outputs 1 except with negligible probability. First, by the definition of successful split strategies, P_1 must output $f_1(x_1, x_2')$, except with negligible probability. However, here P_2^b chooses $x_2' = x_2$ and so we have that except with negligible probability P_1 outputs $f_1(x_1, x_2)$. Next, notice that $\alpha_2' \in X_2$ and $f_2(\alpha_1, \alpha_2') \neq f_2(\alpha_1', \alpha_2')$. Therefore, the value x_1' output by P_2^a must match *exactly one* of α_1 and α_1'. Thus, it must be that $\mathcal{A}$ hands $\mathcal{Z}$ the correct input x_1, except with negligible probability. We therefore have that in a real execution, P_1 outputs $f(x_1, x_2)$ and $\mathcal{A}$ hands $\mathcal{Z}$ the correct value x_1 (except with negligible probability). Thus, by the definition of $\mathcal{Z}$, it outputs 1, except with negligible probability.

In order to derive a contradiction, it suffices to show that for every simulator $\mathcal{S}$ for the ideal process, $\mathcal{Z}$ outputs 0 with non-negligible probability. In the ideal process, the simulator $\mathcal{S}$ receives an input $x_2 \in \{\alpha_2, \alpha_2'\}$, sends an input $\tilde{x}_2$ of its choice to $\mathcal{F}_f$, and receives back $f(x_1, \tilde{x}_2)$. Now, consider the case that $x_2 = \alpha_2$ (this occurs with probability $1/2$). $\mathcal{S}$ has two possible strategies for choosing $\tilde{x}_2$:

1. *The value $\tilde{x}_2$ sent by $\mathcal{S}$ to $\mathcal{F}_f$ is such that $f(\alpha_1, \tilde{x}_2) \neq f(\alpha_1', \tilde{x}_2)$*: Let a denote the value $f(\alpha_1, \alpha_2)$, which also equals $f(\alpha_1', \alpha_2)$. Now, $x_2 = \alpha_2$. Therefore, $f(x_1, x_2) = a$ for $x_1 = \alpha_1$ and $x_1 = \alpha_1'$. However, at least one of $f(\alpha_1, \tilde{x}_2)$ and $f(\alpha_1', \tilde{x}_2)$ does *not* equal a (because in this case, $f(\alpha_1, \tilde{x}_2) \neq f(\alpha_1', \tilde{x}_2)$). Therefore, with probability $1/2$, the output $f(x_1, \tilde{x}_2)$ received by P_1 does not equal $f(x_1, x_2)$. (In order to see that this is the correct probability, recall that $\mathcal{S}$ receives no information on P_1's input x_1 before it sends $\tilde{x}_2$. Therefore, we can view an ideal execution as one where $\mathcal{S}$ first sends $\tilde{x}_2$ and then $x_1 \in_R \{\alpha_1, \alpha_1'\}$ is chosen. Thus, with probability $1/2$, the output of P_1 will not equal $f(x_1, x_2)$.) We conclude that $\mathcal{Z}$ outputs 0 with probability at least $1/2$.
2. *The value $\tilde{x}_2$ sent by $\mathcal{S}$ to $\mathcal{F}_f$ is such that $f(\alpha_1, \tilde{x}_2) = f(\alpha_1', \tilde{x}_2)$*: In this case the output received by $\mathcal{S}$ reveals nothing about P_1's input (α_1 or α_1'). Therefore, $\mathcal{S}$ can succeed in sending $\mathcal{Z}$ the correct x_1 with probability at most $1/2$.

By the above, we have that when P_2's input equals α_2, the environment $\mathcal{Z}$ outputs 0 with probability at least $1/2$. Since P_2's input is chosen uniformly from $\{\alpha_2, \alpha_2'\}$, we have that this "bad case" also happens with probability $1/2$. Combining this together, we conclude that for every possible $\mathcal{S}$, the environment $\mathcal{Z}$ outputs 0 in an ideal execution with probability at least $1/4$. In contrast, as we have seen, $\mathcal{Z}$ outputs 0 with at most negligible probability in the real model. Therefore, $\mathcal{Z}$ distinguishes the ideal and real executions with non-negligible probability, in contradiction to the security of Π_f. □

Relaxing the requirements regarding same-output. Let $f = (f_1, f_2)$. Then Theorem 4.3 is stated for the special case of same-output functions where $f_1 = f_2$. However, the proof of the theorem only uses the fact that both f_1 and f_2 have an insecure minor in the same place. Thus, the impossibility result is actually more general than stated.

Relaxations of UC. Theorem 4.3 remains unchanged for the relaxation of UC where the order of quantifiers is reversed. However, when a uniform $\mathcal{Z}$ is considered, we cannot assume that it always knows an insecure minor in f. Therefore, we require that f has an insecure minor that can be efficiently found by a uniform Turing machine. This holds, for example, in the case that the domain of f is finite.

4.3. *Same-Output Functions with Embedded XORs*

This section contains an impossibility result for same-output functions with another combinatorial property, namely those functions containing an embedded-XOR. A function f is said to contain an embedded-XOR if there exist inputs α_1, α_1', α_2, and α_2' such that the two sets $A_0 \stackrel{\text{def}}{=} \{f(\alpha_1, \alpha_2), f(\alpha_1', \alpha_2')\}$ and $A_1 \stackrel{\text{def}}{=} \{f(\alpha_1, \alpha_2'), f(\alpha_1', \alpha_2)\}$ are disjoint; see Table 2.

(In other words, the table describes an embedded-XOR if no two elements in a single row or column are equal. The name "embedded-XOR" originates from the case of boolean functions f, where one can pick $A_0 = \{0\}$ and $A_1 = \{1\}$.) The intuitive idea is that none of the parties, based on its input (among those in the embedded-XOR subdomain), should be able to bias the output towards one of the sets A_0, A_1 of its choice. In our impossibility proof, we will in fact show a strategy for P_2 to bias the output. We now show that no function containing an embedded-XOR can be securely computed in the plain model.

Theorem 4.4. *Let f be a polynomial-time same-output function containing an embedded-XOR, and let $\mathcal{F}_f$ be the two-party ideal functionality that receives x_1 and x_2 from P_1 and P_2, respectively, and hands both parties $f(x_1, x_2)$. Then $\mathcal{F}_f$ cannot be securely realized in the plain model by a non-trivial protocol.*

Proof. Again, we prove this lemma using Lemma 3.3. However, the use here is different. That is, instead of relying on the extraction property (step 2(b) of Definition 3.1

Table 2. An embedded-XOR —if $\{a, d\} \cap \{b, c\} = \emptyset$.

	α_2	α_2'
α_1	a	b
α_1'	c	d

and item (1) of Definition 3.2), we rely on the fact that P_2 can influence the output by choosing its input as a function of P_1's input (step 2(c)), and then cause P_1 to output the value y that corresponds to these inputs (step 2(d) and item (2) of Definition 3.2). That is, P_2 is able to bias the output, something which it should not be able to do when a function has an embedded-XOR.

Formally, let f be a polynomial-time same-output two-party function and let $\alpha_1, \alpha'_1, \alpha_2, \alpha'_2$ form an embedded-XOR in f with corresponding sets A_0, A_1 (as described above). Furthermore, assume that f does not have an insecure minor (otherwise, the theorem already holds by applying Theorem 4.3). Now, assume by contradiction that $\mathcal{F}_f$ can be securely realized by a protocol Π_f. Then consider a real execution of Π_f with an honest P_1 and an adversary who corrupts P_2. The environment $\mathcal{Z}$ for this execution chooses a pair of inputs (x_1, x_2) where $x_1 \in_R \{\alpha_1, \alpha'_1\}$ and $x_2 \in_R \{\alpha_2, \alpha'_2\}$. $\mathcal{Z}$ then writes x_1 and x_2 on P_1 and P_2's respective input tapes. Furthermore, $\mathcal{Z}$ passes $\mathcal{A}$ the set $X_2 = \{\alpha_2, \alpha'_2\}$. Finally, $\mathcal{Z}$ outputs 1 if and only if the output of P_1 is in the set A_0. This concludes the description of $\mathcal{Z}$. We now describe the real-life adversary $\mathcal{A}$. Adversary $\mathcal{A}$ runs a split adversarial strategy for P_2, as follows. When P_2^b receives a value x'_1 from P_2^a, it computes $v = f(x'_1, \alpha_2)$ and $w = f(x'_1, \alpha'_2)$. If both $v, w \in A_0$ or both $v, w \notin A_0$, then P_2^b sets $x'_2 = \alpha_2$. (This case is just for completeness; as we will see below, it occurs with at most negligible probability.) Otherwise, if $v \in A_0$ and $w \notin A_0$, then P_2^b sets $x'_2 = \alpha_2$ (in order that $f(x'_1, x'_2) \in A_0$). Finally, if $v \notin A_0$ and $w \in A_0$, then P_2^b sets $x'_2 = \alpha'_2$ (again, in order that $f(x'_1, x'_2) \in A_0$). Adversary $\mathcal{A}$ runs the entire successful strategy and then halts.

We now show that in the real-life model with this $\mathcal{A}$, the environment $\mathcal{Z}$ outputs 1, except with negligible probability. In order to see this, first recall that by Lemma 3.3, a successful strategy for the above P_2^b exists. Now, since P_2 is successful, we have that except with negligible probability, P_2^a must hand P_2^b a value x'_1 such that for every $x_2 \in X_2$, $f(x'_1, x_2) = f(x_1, x_2)$. Therefore, except with negligible probability, it must be that one of v and w above is in A_0 and the other is in A_1 (otherwise, this condition on x'_1 is not fulfilled). Furthermore, by item (2) of Definition 3.2, party P_1 outputs $f(x_1, x'_2)$ except with negligible probability. Now, x'_2 is chosen so that $f(x'_1, x'_2) \in A_0$. Since $x'_2 \in X_2$, we also know that $f(x'_1, x'_2) = f(x_1, x'_2)$. Therefore, the value $f(x_1, x'_2)$ that is output by P_1 is guaranteed to be in A_0, except with negligible probability. We conclude that $\mathcal{Z}$ outputs 1 in the real-life model (again, except with negligible probability).

In order to derive a contradiction, it suffices to show that for every $\mathcal{S}$, the environment $\mathcal{Z}$ outputs 1 in an ideal execution with probability at most $1/2$. This is demonstrated by proving that in an ideal execution, $\mathcal{S}$ can cause P_1's output to be in A_0 with probability at most $1/2$. In an ideal execution, the simulator $\mathcal{S}$ receives an input $x_2 \in \{\alpha_2, \alpha'_2\}$, sends an input $\tilde{x}_2$ of its choice to $\mathcal{F}_f$, and receives back $f(x_1, \tilde{x}_2)$. The important point here is that P_1's output is defined as soon as $\mathcal{S}$ sends $\tilde{x}_2$ to $\mathcal{F}_f$. Furthermore, $\mathcal{S}$ sends $\tilde{x}_2$ to $\mathcal{F}_f$ without any information whatsoever on P_1's input x_1. Now, since f has no insecure minor and $f(\alpha_1, \alpha_2) \neq f(\alpha'_1, \alpha_2)$, it follows that for *every* $\tilde{x}_2$, $f(\alpha_1, \tilde{x}_2) \neq f(\alpha'_1, \tilde{x}_2)$ (otherwise, $\alpha_1, \alpha'_1, \alpha_2, \tilde{x}_2$ would constitute an insecure minor in f). Therefore, for $x_1 \in \{\alpha_1, \alpha'_1\}$ and for every $\tilde{x}_2$, at most one of $f(x_1, \tilde{x}_2)$ is in A_0. Since $\mathcal{Z}$ chooses x_1 uniformly from $\{\alpha_1, \alpha'_1\}$, we have that no matter what value $\tilde{x}_2$ that $\mathcal{S}$ sends to $\mathcal{F}_f$, the value $f(x_1, \tilde{x}_2)$ that is output by P_1 is in A_0 with probability at most $1/2$.

We conclude that for every possible $\mathcal{S}$, the environment $\mathcal{Z}$ outputs 0 in an ideal execution with probability at least $1/2$. In contrast, $\mathcal{Z}$ outputs 0 with at most negligible probability in the real model. Therefore, $\mathcal{Z}$ distinguishes the ideal and real executions with non-negligible probability, in contradiction to the security of Π_f. □

Relaxations of UC. As above, Theorem 4.4 remains unchanged for the relaxation of UC where the order of quantifiers is reversed. However, when a uniform $\mathcal{Z}$ is considered, we require that f has an embedded-XOR that can be efficiently found by a uniform Turing machine.

4.4. *Not Completely Revealing Functions*

In this section we consider functions that are not completely revealing. This notion does *not* refer to *protocols* and information that is "revealed" by them. Rather, it refers to the question of whether or not a party's input is completely revealed by the *function output itself*. Note that in this section we do not limit ourselves to functions that have only one input or output. Rather, we consider the general case where f_1 and f_2 may be different functions and may depend on both parties' inputs.

Loosely speaking, a function is completely revealing for party P_1, if party P_2 can choose an input so that the output of the function fully reveals P_1's input (for *all* possible choices of P_1's input). That is, a function is completely revealing for P_1 if there exists an input x_2 for P_2 so that for every x_1, it is possible to derive x_1 from $f_2(x_1, x_2)$. For example, let us take the maximum function for a given range, say $\{0, \ldots, n\}$. Then party P_2 can input $x_2 = 0$ and the result is that it will always learn P_1's exact input. In contrast, the less-than function (i.e., $f(x, y) = 1$ iff $x < y$) is *not* completely revealing because for any input used by P_2, there will always be uncertainty about P_1's input (unless P_1's input is the smallest or largest in the range). In fact, any function where the range is smaller than the domain, like for the less-than function, cannot be completely revealing. (This holds if there are no equivalent inputs; see below.)

Functions over finite domains. We first define what it means for a function to be completely revealing for the special case of functions over finite domains (i.e., domains that are of constant-size, and not dependent on the security parameter). The definition in this case is simpler and more intuitive.

We begin by defining what it means for two inputs to be "equivalent": Let $f\colon X \times X \to \{0,1\}^* \times \{0,1\}^*$ be a two-party function and denote $f = (f_1, f_2)$. Let $x_1, x_1' \in X$. We say that x_1 and x_1' are equivalent with respect to f_2 if for every $x_2 \in X$ it holds that $f_2(x_1, x_2) = f_2(x_1', x_2)$. The rationale for this definition is that if x_1 and x_1' are equivalent with respect to f_2, then x_1 can always be used instead of x_1' without affecting P_2's output. We now define completely revealing functions:

Definition 4.5 (Completely Revealing Functions over Finite Domains). Let $f\colon X \times X \to \{0,1\}^* \times \{0,1\}^*$ be a polynomial-time two-party function such that the domain X is finite, and denote $f = (f_1, f_2)$. We say that function f is completely revealing for P_1 if there exists an input $x_2 \in X$ for P_2 such that for every two distinct inputs x_1 and x_1' for P_1 that are not equivalent with respect to f_2, it holds that $f_2(x_1, x_2) \neq f_2(x_1', x_2)$.

Completely revealing for P_2 is defined analogously. We say that a function is completely revealing if it is completely revealing for both P_1 and P_2.

If a function is completely revealing for P_1, then party P_2 can set its own input to be the "special value" x_2 from the definition, and then P_2 will always obtain the exact input used by P_1. Specifically, given $y = f_2(x_1, x_2)$, party P_2 can traverse over all X and find the unique x_1 for which it holds that $f_2(x_1, x_2) = y$ (where uniqueness here is modulo equivalent inputs x_1 and x_1'). It then follows that x_1 must be P_1's input (or at least is equivalent to it). Thus we see that P_1's input is completely revealed by f_2. In contrast, if a function f is *not* completely revealing for P_1, then there does not exist such an input for P_2 that enables it to determine P_1's input completely. This is because for every x_2 that is input by P_2, there exist two non-equivalent inputs x_1 and x_1' such that $f_2(x_1, x_2) = f_2(x_1', x_2)$. Therefore, if P_1's input happens to be x_1 or x_1', it follows that P_2 is unable to determine which of these inputs were used by P_1. Notice that if a function is not completely revealing, P_2 may still learn much of P_1's input (or even the exact input "most of the time"). However, there is a *possibility* that P_2 will not fully obtain P_1's input. As we will see, the existence of this "possibility" suffices for proving impossibility.

Note that we require that x_1 and x_1' be non-equivalent because otherwise, for all intents and purposes, x_1 and x_1' are the same input. That is, x_1 and x_1' could be used interchangeably by P_1 with no affect whatsoever on the output. Notice that if we were to remove the requirement of non-equivalence between x_1 and x_1', then it may be that a function is not completely revealing simply because for equivalent inputs it holds that $f_2(x_1, x_2) = f_2(x_1', x_2)$ for every x_2. Now, recall that if a function is completely revealing, then P_2 should be able to learn the exact input of P_1 by inputting the "special value" x_2 from the definition. In the case of equivalent x_1 and x_1', it is indeed true that P_2 cannot differentiate between the case that P_1 has input x_1 and the case that P_1 has input x_1'. However, as far as we are concerned x_1 and x_1' are the same input, and so there is nothing to differentiate between them. Thus, the requirement of x_1 and x_1' being not equivalent is natural and follows the intuition of what it means for a function to be completely revealing.

As we have mentioned above, the "less than" function (otherwise known as Yao's millionaires' problem) is not completely revealing, as long as the range of inputs is larger than 2. This can be easily demonstrated.

Functions over infinite domains. In the case of functions that may be over an infinite domain, the definition of completely revealing is slightly more complex.[9] Recall that in the definition of completely revealing for functions over finite domains, we require the existence of a *single* input x_2 that can reveal P_1's input, for *all* possible inputs $x_1 \in X$ where X is the entire domain. However, when the domain is infinite, we will only require that for every *polynomial-size* set $X_1 \subseteq X$ there exists a single input x_2, such that x_2 can

[9] We note that in the case of infinite domains, the convention is that for a given security parameter k, only a finite subset of the domain is considered. However, this finite subset may be of size exponential in k (in particular, it may contain all of the strings of length k or $\text{poly}(k)$). Thus, a polynomial-time machine does not have time to traverse the entire domain, in contrast to the above case where it is finite and so of constant size.

reveal P_1's input, for any input $x_1 \in X_1$. Thus, a different x_2 can be used for every subset X_1 of inputs for P_1. Notice that any function that is completely revealing when a single x_2 can be used to reveal all inputs x_1, is also completely revealing when every polynomial-size set X_1 can have a different x_2. However, the reverse is not true (and it is not difficult to construct a concrete example). This modification of the definition therefore makes the set of completely revealing functions larger. Since we prove impossibility for any function that is *not* completely revealing, this actually weakens our impossibility result. Nevertheless, it is needed for our proof. We now present the formal definition:

Definition 4.6 (Completely Revealing Functions). Let $f\colon X \times X \to \{0,1\}^* \times \{0,1\}^*$ be a two-party function, denoted $f = (f_1, f_2)$, and let k be the security parameter. We say that function f is completely revealing for P_1 if for every polynomial $p(\cdot)$, all sufficiently large k's, and every set of inputs $X_1 \subseteq X$ for P_1 of size at most $p(k)$, there exists an input $x_2 \in X$ for P_2, such that for every pair of distinct inputs $x_1, x_1' \in X_1$ that are not equivalent with respect to f_2, it holds that $f_2(x_1, x_2) \neq f_2(x_1', x_2)$. Completely revealing for P_2 is defined analogously. We say that a function is completely revealing if it is completely revealing for both P_1 and P_2.

We stress, once again, that "completely revealing" or "not completely revealing" is a property of *functions* and not of protocols. We now show that a function that is *not* completely revealing cannot be securely realized in the plain model by any non-trivial protocol.

Theorem 4.7. *Let $f = (f_1, f_2)$ be a polynomial-time two-party function that is not completely revealing, and let $\mathcal{F}_f$ be the two-party ideal functionality that receives x_1 from P_1 and x_2 from P_2, and hands $f_1(x_1, x_2)$ to P_1 and $f_2(x_1, x_2)$ to P_2. Then $\mathcal{F}_f$ cannot be securely realized in the plain model by a non-trivial protocol.*

Proof. The idea behind the proof here is very similar to that of Theorem 4.2. Specifically, according to Lemma 3.3, a real adversary can always extract the input of the other party (or an input equivalent to it) by running a split adversarial strategy. Therefore, an ideal adversary can also extract this input. However, if f is not completely revealing, then it is impossible to extract the exact input (or an input equivalent to it) with high enough probability. We therefore conclude that if $\mathcal{F}_f$ can be securely realized, then f must be completely revealing.

Let $f = (f_1, f_2)$ be a polynomial-time two-party function, and assume that there exists a protocol Π_f that securely realizes $\mathcal{F}_f$. We now prove that this implies that f is completely revealing. We actually prove that f is completely revealing for P_1; the fact that f is also completely revealing for P_2 is proven analogously. Assume by contradiction that f is *not* completely revealing for P_1. Then this implies that there exists a polynomial $p(\cdot)$ such that for infinitely many k's, there exists a set X_1 such that $|X_1| \leq p(k)$ and the following holds: For every x_2 there exist at least two inputs $x_1, x_1' \in X_1$ such that $f_2(x_1, x_2) = f_2(x_1', x_2)$. Let X_1 be a minimal such set (for a given $p(\cdot)$ and k). (By minimality here, we mean that no input can be removed from X_1 while preserving the requirements in the definition.) We construct a polynomial-size set of inputs X_2 for P_2 as follows. First, we claim that for every pair of inputs $x_1, x_1' \in X_1$ there exists an input x_2 such that $f_2(x_1, x_2) \neq f_2(x_1', x_2)$. This follows from the fact X_1 is minimal and therefore does not contain any two inputs that are equivalent with respect to f_2. Next, we construct the set X_2 by adding a single value x_2 as above for every $x_1, x_1' \in X_1$ (i.e., we add an x_2 for which $f_2(x_1, x_2) \neq f_2(x_1', x_2)$). Note that since the size of X_1 is polynomial in k, the same holds for X_2.

We are now ready to define an environment $\mathcal{Z}$ and a real-life adversary $\mathcal{A}$ for protocol Π_f. The environment $\mathcal{Z}$ chooses $x_1 \in_R X_1$ and $x_2 \in_R X_2$ and writes x_1 and x_2 on P_1 and P_2's respective input tapes. Furthermore, $\mathcal{Z}$ passes $\mathcal{A}$ the set X_2. Finally, $\mathcal{Z}$ outputs 1 if and only if at some stage $\mathcal{A}$ sends $\mathcal{Z}$ the correct input value x_1 used by P_1. We now describe the adversary $\mathcal{A}$. Adversary $\mathcal{A}$ controls party P_2 and runs a successful split adversarial strategy (the exact strategy used by P_2^b to choose x_2' is immaterial here because we only need the value x_1' obtained by P_2^a in the first part of the attack). At some stage of the attack, $\mathcal{A}$ obtains the value x_1' that P_2^a passes to P_2^b. Given this value, $\mathcal{A}$ finds an input $\tilde{x}_1 \in X_1$ such that for every $\tilde{x}_2 \in X_2$ it holds that $f_2(\tilde{x}_1, \tilde{x}_2) = f_2(x_1', \tilde{x}_2)$. $\mathcal{A}$ then hands this $\tilde{x}_1$ to $\mathcal{Z}$ (if such a value does not exist, then $\mathcal{A}$ outputs fail).

We now claim that in the real-life model, $\mathcal{Z}$ outputs 1 except with negligible probability. This follows from the fact that by Lemma 3.3 a successful split strategy exists for P_2. Therefore, except with negligible probability, the value x_1' obtained by P_2^a is such that for every $x_2 \in X_2$, $f_2(x_1', x_2) = f_2(x_1, x_2)$, where $x_1 \in X_1$ is the input that $\mathcal{Z}$ writes on P_1's input tape and X_2 is the polynomial-size set of inputs given to $\mathcal{A}$ by $\mathcal{Z}$. This means that there exists at least one value $\tilde{x}_1 \in X_1$ such that for every $\tilde{x}_2 \in X_2$ it holds that $f_2(\tilde{x}_1, \tilde{x}_2) = f_2(x_1', \tilde{x}_2)$; specifically, this value is P_1's correct input x_1. It remains to show that there is at most one such value, and therefore $\mathcal{A}$ sends $\mathcal{Z}$ the correct input x_1. This follows from the construction of X_2. Specifically, for every $x_1, x_1' \in X_1$ there exists an input $x_2 \in X_2$ such that $f_2(x_1, x_2) \neq f_2(x_1', x_2)$. Now, let x_1' be the value that $\mathcal{A}$ obtains from P_2^a. Then there cannot be two values $\tilde{x}_1$ and $\hat{x}_1$ such that for every $\tilde{x}_2 \in X_2$, $f_2(\tilde{x}_1, \tilde{x}_2) = f_2(x_1', \tilde{x}_2)$ *and* $f_2(\hat{x}_1, \tilde{x}_2) = f_2(x_1', \tilde{x}_2)$, because this would imply that $\tilde{x}_1, \hat{x}_1 \in X_1$ result in the same output for all $\tilde{x}_2 \in X_2$. This is in contradiction to the construction of X_2. We conclude that there is only one value that passes the test carried out by $\mathcal{A}$, and this is P_1's correct input x_1. That is, $\mathcal{Z}$ obtains P_1's correct input from $\mathcal{A}$, and so outputs 1, except with negligible probability.

The proof is concluded by showing that in the ideal process there does not exist a simulator $\mathcal{S}$ that can cause $\mathcal{Z}$ to output 1 with probability that is negligibly close to 1. This can be seen as follows. The simulator $\mathcal{S}$ sends some input $\tilde{x}_2$ to $\mathcal{F}_f$ and receives back $f_2(x_1, \tilde{x}_2)$. Furthermore, $\mathcal{S}$ sends $\tilde{x}_2$ before receiving any information about x_1. Therefore, we can view the ideal process as one where $\mathcal{S}$ first sends $\tilde{x}_2$ to $\mathcal{F}_f$ and then $\mathcal{Z}$ chooses P_1's input x_1 uniformly from X_1. Now, by our contradicting assumption, since f is not completely revealing for P_1, for every $\tilde{x}_2$ there exist two distinct inputs $\tilde{x}_1, \tilde{x}_1' \in X_1$ such that $f_2(\tilde{x}_1, \tilde{x}_2) = f_2(\tilde{x}_1', \tilde{x}_2)$. Therefore, with probability $2/|X_1|$, we have that $x_1 \in \{\tilde{x}_1, \tilde{x}_1'\}$. In this case, information theoretically, $\mathcal{S}$ can send $\mathcal{Z}$ the correct x_1 with probability at most $1/2$. We conclude that in the ideal process, $\mathcal{Z}$ outputs 0 with probability at least $1/|X_1|$, for every ideal process simulator $\mathcal{S}$. Since the size of X_1 is polynomial in k, we have that $\mathcal{Z}$ distinguishes the real and ideal processes with non-negligible probability, in contradiction to the assumed security of Π_f. □

Impossibility of oblivious transfer. We remark that Theorem 4.7 can be used to rule out the possibility of securely realizing the 2-out-of-1 oblivious transfer functionality [R], [EGL]. This is because this functionality is clearly not completely revealing for the "sender."

Completely revealing functions. We remark that some completely revealing functions *can* be securely realized in the plain model, and some *cannot*. The fact that some completely revealing functions can be securely realized is demonstrated below in Section 5.3. In order to see that some completely revealing functions cannot be securely realized, notice that the XOR function $f(x_1, x_2) = (x_1 \oplus x_2, x_1 \oplus x_2)$ is completely revealing (e.g., take $x_2 = 0^k$). However, by Theorem 4.4, it cannot be securely realized in the plain model.

Relaxations of UC. Notice that in the proof of Theorem 4.7, the environment $\mathcal{Z}$ is fixed. Therefore, the theorem and proof remain the same for the relaxed definition where the order of quantifiers between the environment and simulator is reversed. In contrast, if the environment is assumed to be a uniform machine, then success is only defined with respect to uniform environments (in particular, this means that X_1 and X_2 must be uniformly generated). This is still very general and is equivalent for functions that have finite domains.

5. Characterizations for Deterministic Functions

This section is organized as follows: We first present a characterization for the case of *single-input* functions (i.e., functions that depend on only one of the two inputs). Next, we show that functions that do not have an insecure minor or an embedded XOR actually depend on only one input. This is combined to provide a full characterization of *same-output* functions. Following this, we provide a characterization of *single-output* functions for the special case that the function domain is *finite*.

Secure channels. In order to provide full characterizations of feasibility regarding the above-described classes of functionalities, we combine impossibility results proven in Section 4, as well as present protocols where impossibility does not hold. We note that all of the protocols that we present assume the existence of UC-secure channels (which provide both authenticity as well as privacy). We do not define this notion here; details on the definitions and constructions can be found in [CK1]. We do, however, remark that in the plain model as defined here, UC-secure channels can be securely realized under standard complexity assumptions.

5.1. *Characterization for Single-Input Functions*

In this section we show that the notion of "efficient invertibility" in Definition 4.1 actually fully characterizes the single-input functions that can and cannot be securely realized in the framework of universal composability. Recall that we have already proven that it is impossible to securely realize functions that are not efficiently invertible in Theorem 4.2.

Theorem 5.1. *Let $f\colon X \to \{0,1\}^*$ be a polynomial-time function and let $\mathcal{F}_f$ be a functionality that receives x from P_1 and sends $f(x)$ to P_2. Then, assuming UC-secure channels, $\mathcal{F}_f$ can be securely realized in the plain model by a non-trivial protocol if and only if f is efficiently invertible.* (*The above also holds also when P_1 and P_2 are reversed.*)

Proof. As we have mentioned, the fact that $\mathcal{F}_f$ cannot be securely realized if f is not efficiently invertible has already been shown in Theorem 4.2. It therefore remains to prove that if f *is* efficiently invertible, then $\mathcal{F}_f$ can be securely realized.

This is achieved by the following simple protocol: Upon input x and security parameter k, party P_1 computes $y = f(x)$ and runs the inverting machine M on $(1^k, y)$. Then P_1 sends P_2 the value x' output by M. (In order to guarantee security against an external adversary that does not corrupt any party and so should learn nothing by eavesdropping on the protocol messages, the value x' is sent over the UC-secure channel.) Simulation of this protocol is demonstrated by constructing a simulator who receives $y = f(x)$, and simulates P_1 sending P_2 the output of $M(1^k, y)$. The proof is straightforward and so details are omitted. □

Relaxations of UC. Following the proof of Theorem 4.2, we discussed the existence of analogous impossibility results for the relaxations of UC where the environment $\mathcal{Z}$ is uniform and for the relaxation where the order of quantifiers is reversed. We note that the characterization described here holds also for the relaxation of UC in which the environment $\mathcal{Z}$ is uniform. However, in the case that the order of quantifiers is reversed, the analogous impossibility result obtained is only for "hard-on-the-average" distributions. Therefore, we do not obtain a full characterization. Specifically, we have shown that invertible efficiently functions can be securely realized, and yet functions with hard-on-the-average distributions cannot. However, there may exist functions that are not efficiently invertible and do not have such hard-on-the-average distributions. We do not know whether or not such a function can be securely realized under this relaxation of UC.

5.2. *Characterization for Same-Output Functions*

This section provides a full characterization of the deterministic two-party *same-output* functionalities that can be securely realized in the plain model. Let $f\colon X \times X \to \{0,1\}^*$ be a deterministic two-party function. Each of Theorems 4.3 and 4.4 provides a necessary condition for $\mathcal{F}_f$ to be securely realizable (namely, f should not contain an insecure minor or an embedded-XOR). In addition, Theorem 5.1 gives a characterization of those functionalities $\mathcal{F}_f$ that can be securely realized, assuming that f depends on the input of one party only. In this section we show that the combination of these three theorems is actually quite powerful. Indeed, we show that this provides a full characterization of the two-party, same-output deterministic functions that can be securely realized. In fact, this characterization turns out to be very simple.

Theorem 5.2. *Let f be a polynomial-time same-output two-party function and let $\mathcal{F}_f$ be a functionality that receives x_1 and x_2 from P_1 and P_2 respectively, and hands both parties $f(x_1, x_2)$. Then, assuming UC-secure channels, $\mathcal{F}_f$ can be securely realized in the plain model by a non-trivial protocol if and only if f is an efficiently invertible function depending on (at most) one of the inputs (x_1 or x_2).*

Proof. First we prove the theorem for the case that f contains an insecure-minor or an embedded-XOR (with respect to either P_1 or P_2). By Theorems 4.3 and 4.4, in this case $\mathcal{F}_f$ cannot be securely realized. Indeed, such functions f do *not* solely depend on the input of a single party; that is, for each party there is some input for which the output depends on the other party's input.

Next we prove the theorem for the case that f does not contain an insecure-minor (with respect to either P_1 or P_2) or an embedded-XOR. We prove that in this case f depends on the input of (at most) one party and hence, by Theorem 5.1, the present theorem follows. Pick any $x \in X$ and let $a = f(x, x)$. Let $B_1 = \{x_1 \mid f(x_1, x) = a\}$ and $B_2 = \{x_2 \mid f(x, x_2) = a\}$. Since $f(x, x) = a$, both sets are non-empty. Next we claim that at least one of $\bar{B}_1$ and $\bar{B}_2$ is empty; otherwise, if there exist $\alpha_1 \in \bar{B}_1$ and $\alpha_2 \in \bar{B}_2$, then setting $\alpha_1' = \alpha_2' = x$ gives us a minor which is either an insecure minor or an embedded-XOR. To see this, denote $b = f(\alpha_1, x)$ and $c = f(x, \alpha_2)$; by the definition of $\bar{B}_1, \bar{B}_2$ both b and c are different than a. Consider the possible values for $d = f(\alpha_1, \alpha_2)$. If $d = b$ or $d = c$, we get an insecure minor; if $d = a$ or $d \notin \{a, b, c\}$, we get an embedded-XOR. Thus, we showed that at least one of $\bar{B}_1, \bar{B}_2$ is empty; assume, without loss of generality, that it is $\bar{B}_2$.

We now show that for every x_1, the function $g(\cdot) = f(x_1, \cdot)$ is constant. Fix x_1. Then we show that for every x_2, x_3 it must hold that if $f(x_1, x_2) = b$ and $f(x_1, x_3) = c$, then $b = c$ (thus demonstrating that $f(x_1, \cdot)$ is constant). In order to see why this holds, notice that if $x_1 = x$, then $b = c = a$ because by our assumption $\bar{B}_2$ is empty. On the other hand, if $x_1 \neq x$ and this is not true, then we obtain an insecure minor in f at (x, x_1, x_2, x_3), because $f(x, x_2) = f(x, x_3) = a$ and $f(x_1, x_2) \neq f(x_1, x_3)$. This contradicts the assumption that f does not contain an insecure minor.

We conclude that either $f(x_1, \cdot)$ is a constant function for every x_1, or $f(\cdot, x_2)$ is a constant function for every x_2. That is, f depends on the input of at most one party, as needed. □

5.3. *Characterization for Single-Output Functions over Finite Domains*

This section contains a full characterization of the *single-output* two-party functions over finite domains that can be securely realized in the plain model. Recall that a function $f = (f_1, f_2)$ is single-output if $f_1 = \lambda$ or $f_2 = \lambda$.

Theorem 5.3. *Let $f\colon X \times X \to \{0, 1\}^*$ be a polynomial-time single-output function where X is a finite set, and let $\mathcal{F}_f$ be a functionality that receives x_1 from P_1 and x_2 from P_2, and sends $f(x_1, x_2)$ to P_2. Then, assuming UC-secure channels, $\mathcal{F}_f$ can be securely realized in the plain model by a non-trivial protocol if and only if f is completely revealing for P_1. (The above also holds when P_1 and P_2 are reversed.)*

Proof. We have already proven in Theorem 4.7 that if f is not completely revealing for P_1, then $\mathcal{F}_f$ cannot be securely realized. It therefore remains to show the converse. That is, assume that f is completely revealing for P_1. Then, by Definition 4.5, there exists an input x_2 such that for every $x_1, x_1' \in X$ that are not equivalent with respect to f, it holds that $f(x_1, x_2) \neq f(x_1', x_2)$. This therefore yields the following protocol: Let x_1 be P_1's input and let X_1 be the set of inputs that are equivalent to x_1 with respect to f_2. Then P_1 sends P_2 the value x_1' that is lexicographically the smallest in X_1. (Note that X_1 can be efficiently computed because f has a finite domain.) As in the protocol described in the proof of Theorem 5.1, the value x_1' is sent over a UC-secure channel in order to guarantee security against an external adversary that does not corrupt any party.

First, note that the protocol is non-trivial. Next, in order to see why it securely realizes $\mathcal{F}_f$, consider the following ideal-process simulator $\mathcal{S}$ (for the case that P_2 is corrupt). Simulator $\mathcal{S}$ sends x_2 to $\mathcal{F}_f$, where x_2 is the above-mentioned input that is guaranteed to exist for f. $\mathcal{S}$ then receives back an output y. Since f has a finite domain, it is possible to compute efficiently the set of values X_1 such that for every $x_1 \in X_1$, $f(x_1, x_2) = y$. By the assumption that f is completely revealing, this set X_1 is exactly the set of all values that are equivalent to x_1. $\mathcal{S}$ therefore hands the adversary $\mathcal{A}$ the lexicographically smallest value from X_1, as it expects to see in a real execution. This completes the proof. □

We note that the above proof cannot be extended to functions that are not single-output. For example, the XOR function over domain $\{0, 1\}$ is clearly completely revealing. However, if both parties obtain output, then this function cannot be securely realized, as shown in Theorem 4.4.

6. Probabilistic Same-Output Functionalities

In this section we concentrate on *probabilistic* two-party functionalities where both parties obtain the same output. We show that probabilistic functionalities that generate non-trivial distributions cannot be securely realized in the plain model in the UC framework. This rules out the possibility of realizing any "coin-tossing style" functionality, or any functionality whose outcome is "unpredictable" whatever the choice of inputs by the parties. It is stressed, however, that our result does not rule out the possibility of securely realizing other useful probabilistic functionalities, such as functionalities where, when both parties remain uncorrupted, they can obtain a random value that is unknown to the adversary. An important example of such a functionality that can indeed be securely realized is key-exchange.

Let $f = \{f_k\}$ be a family of polynomial-time functions where, for each value of the security parameter k, $f_k\colon X \times X \to \{0, 1\}^*$ is a probabilistic function. We begin by defining the notion of safe values. Informally, such values have the property that they induce non-trivial distributions over the output. More precisely, let $p(\cdot)$ be a polynomial and let $k \in \mathsf{N}$. We say that $x_1 \in X$ is a (p, k)-safe value for P_1 if for every $x_2 \in X$ and all possible output values $v \in \{0, 1\}^*$ it holds that $\Pr[f_k(x_1, x_2) \neq v] > 1/p(k)$. (Indeed, when the security parameter equals k and P_1 inputs the safe value x_1, the output of the function is chosen from a non-trivial distribution, irrespective of the input x_2.) We define (p, k)-safe values for P_2 in an analogous way. A probabilistic function family $f = \{f_k\}$ as above is said to be unpredictable if there exists a polynomial $p(\cdot)$ such that for infinitely many k's, there exist (p, k)-safe values for both P_1 and P_2. (Note that there must be infinitely many k's for which there exist (p, k)-safe values for both P_1 and P_2; for example, it does not suffice if (p, k)-safe values for P_1 only exist for different k's than the (p, k)-safe values for P_2.)

Below, we prove that unpredictable function families cannot be securely realized in the plain model. This is quite a broad impossibility result. In order to see this, consider the structure of a function that is not unpredictable, because, say, P_1 does not have safe values. Such a function has the property that for all but a finite number of k's, for *every* $x_1 \in X$ there exists a value $x_2 \in X$ such that almost all of the support of the distribution $f_k(x_1, x_2)$ falls on one point v. Now, if P_2 uses input x_2, then the output of the function will have a trivial distribution (i.e., it will almost always equal a single value v). Thus, it will essentially behave like a deterministic function.[10]

Theorem 6.1. *Let $f = \{f_k\}$ be a family of unpredictable probabilistic polynomial-time same-output functions and let $\mathcal{F}_f$ be a functionality that, given a security parameter k, receives x_1 from P_1 and x_2 from P_2, samples a value v from the distribution of $f_k(x_1, x_2)$, and hands v to both P_1 and P_2. Then $\mathcal{F}_f$ cannot be securely realized in the plain model by any non-trivial protocol.*

Proof. The proof uses the same ideas as in the proof of Lemma 3.3 and the proofs of impossibility that use Lemma 3.3. Let f be a family of unpredictable polynomial-time probabilistic functions. Assume, by contradiction, that $\mathcal{F}_f$ can be securely realized by a protocol Π_f. We prove that in this case there exists an adversary (called $\tilde{\mathcal{A}}$ below) that can essentially fix the output. This therefore contradicts the assumption that f is unpredictable.

By the assumption in the theorem, f is a family of unpredictable functions. Therefore, there exists a polynomial $p(\cdot)$ such that for infinitely many k's, there exist (p, k)-safe values for P_1 and P_2. Let k be one these infinitely many k's, and let x_1 and x_2 be the (p, k)-safe values for P_1 and P_2, respectively. We are now ready to define a real-life adversary $\mathcal{A}$ and an environment $\mathcal{Z}$. Adversary $\mathcal{A}$ controls party P_1 and is a dummy adversary who does nothing except deliver messages between $\mathcal{Z}$ and P_2 (exactly like $\mathcal{A}$ in Lemma 3.3). Next we define $\mathcal{Z}$ who uses the above safe values x_1 and x_2. Environment $\mathcal{Z}$ writes x_2 on P_2's input tape and locally runs P_1's protocol instructions in Π_f on input x_1 and the incoming messages that it receives from $\mathcal{A}$ (that are in turn received from P_2). At the conclusion of the execution of Π_f, the environment $\mathcal{Z}$ obtains an output, called $\mathcal{Z}$'s local P_1-output. Then $\mathcal{Z}$ outputs 1 if and only if its local P_1-output equals P_2's output.

We first claim that in the real-life process, $\mathcal{Z}$ outputs 1 except with negligible probability. This follows from the fact that the real-life process looks exactly like an honest

[10] We note that if the definition of unpredictability is relaxed to require only that there exist infinitely many values of k for which there is *either* a (p, k)-safe value for P_1 *or* a (p, k)-safe value for P_2, then the theorem no longer holds. Indeed, it is possible to construct such functions that are realizable in the plain model. We remark that in the preliminary version of this paper that appeared at EUROCRYPT 2003, it was erroneously stated that impossibility holds even for this relaxed notion of unpredictability.

execution between P_1 and P_2. Furthermore, all messages between these honest parties are delivered by the adversary. Therefore, since Π_f is non-trivial, both parties must receive output and this output is correct. (The formal argument for this is identical to the argument in Lemma 3.3.)

By the assumption that Π_f securely realizes $\mathcal{F}_f$, there exists an ideal-process adversary (simulator) $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish between a real execution of Π_f with $\mathcal{A}$ and an ideal process execution with $\mathcal{S}$ and $\mathcal{F}_f$. In particular, the local P_1-output received by $\mathcal{Z}$ in the ideal process must equal the output that P_2 receives from $\mathcal{F}_f$. Recall that in an ideal execution, $\mathcal{S}$ sends some input x_1' to $\mathcal{F}_f$. Functionality $\mathcal{F}_f$ then samples a value v from $f_k(x_1', x_2)$ and hands this value to both P_1 (who is controlled by $\mathcal{S}$) and P_2. It follows that except with negligible probability, $\mathcal{Z}$'s local P_1-output must be the *exact value* v that $\mathcal{S}$ receives from $\mathcal{F}_f$.

Next, we switch context to a new real-life adversary $\tilde{\mathcal{A}}$ that controls P_2 and a new environment $\tilde{\mathcal{Z}}$. Let x_1 and x_2 be the same values as above (i.e., (p, k)-safe values for P_1 and P_2, respectively). The environment $\tilde{\mathcal{Z}}$ writes the input x_1 on P_1's input tape. In addition, $\tilde{\mathcal{Z}}$ waits to obtain a value x_1' from the adversary. When it receives this value, $\tilde{\mathcal{Z}}$ samples a value v from the distribution of $f_k(x_1', x_2)$ and hands this value to the adversary. Finally, $\tilde{\mathcal{Z}}$ outputs 1 if and only if P_1 outputs v. We now define $\tilde{\mathcal{A}}$ who controls P_2. Adversary $\tilde{\mathcal{A}}$ works by internally running the simulator $\mathcal{S}$ from above, and emulating an ideal execution of $\mathcal{S}$ with $\mathcal{Z}$ and $\mathcal{F}_f$. Each message that $\tilde{\mathcal{A}}$ receives from P_1, it internally passes to $\mathcal{S}$ as if it was sent by the environment $\mathcal{Z}$. Likewise, any message that $\mathcal{S}$ attempts to send to its environment $\mathcal{Z}$, the adversary $\tilde{\mathcal{A}}$ externally sends to party P_1. When the internal $\mathcal{S}$ outputs a value x_1' to be sent to $\mathcal{F}_f$, adversary $\tilde{\mathcal{A}}$ hands x_1' to the environment $\tilde{\mathcal{Z}}$, obtains the value v back from $\tilde{\mathcal{Z}}$, and hands v to $\mathcal{S}$.

We claim that in a real-life execution of Π_f with $\tilde{\mathcal{Z}}$ and $\tilde{\mathcal{A}}$, party P_1 outputs the exact value v generated by $\tilde{\mathcal{Z}}$, except with negligible probability. Consequently, by the definition of $\tilde{\mathcal{Z}}$, it follows that $\tilde{\mathcal{Z}}$ outputs 1 in a real execution, except with negligible probability. To see that this holds, notice that $\mathcal{S}$'s view in an ideal process with $\mathcal{Z}$ and $\mathcal{F}_f$ is identical to its view when it is internally invoked by $\tilde{\mathcal{A}}$. Furthermore, the real P_1 that interacts with $\tilde{\mathcal{A}}$ has exactly the same view as $\mathcal{Z}$ in the ideal process with $\mathcal{S}$. Therefore, the output of P_1 has exactly the same distribution as the local P_1-output of $\mathcal{Z}$ in the ideal process with $\mathcal{S}$. We have already shown that $\mathcal{Z}$'s local P_1-output in this case equals v, except with negligible probability. Therefore, the real P_1's output also equals v, except with negligible probability. Notice that what we have essentially proven here is analogous to proving that $\tilde{\mathcal{A}}$ succeeds in item (2) of a successful split adversarial strategy (as in Lemma 3.3).[11]

The remainder of the proof shows how the existence of such an adversary $\tilde{\mathcal{A}}$ (that succeeds in having P_1 output v) contradicts the security of the protocol for unpredictable functions (this part of the proof is analogous to the proofs of Section 4). Intuitively, since f is unpredictable, a corrupted party should not be able to force the output of an execution to be any one value. However, as we have seen, $\tilde{\mathcal{A}}$ succeeds in doing just this. Formally, we complete the proof by considering an interaction of $\tilde{\mathcal{Z}}$ in an ideal process with $\mathcal{F}_f$ and some ideal-process adversary $\tilde{\mathcal{S}}$ that is guaranteed to exist by the assumption

[11] Of course, this analogy is not exact, and the definition of successful split adversarial strategies must be modified for the case of probabilistic functionalities. We provide a direct proof here (rather than first proving an analogue of Lemma 3.3) since we found that it is simpler and clearer in this case.

that Π_f securely realizes $\mathcal{F}_f$. In this ideal execution, $\tilde{\mathcal{Z}}$ first writes x_1 and x_2 on P_1 and P_2's respective input tapes. Then $\tilde{\mathcal{Z}}$ interacts with $\tilde{\mathcal{S}}$ in the same way that it interacted with $\tilde{\mathcal{A}}$ above. That is, $\tilde{\mathcal{Z}}$ waits to receive some value x_1'' from $\tilde{\mathcal{S}}$. Then it samples a value v from $f_k(x_1'', x_2)$ and hands v back to $\tilde{\mathcal{S}}$. Finally, $\tilde{\mathcal{Z}}$ outputs 1 if and only if P_1's output equals v. The above describes the interaction between $\tilde{\mathcal{Z}}$ and $\tilde{\mathcal{S}}$. In the rest of the ideal execution, the honest P_1 sends its input x_1 to $\mathcal{F}_f$, and $\tilde{\mathcal{S}}$ sends $\mathcal{F}_f$ some $\tilde{x}_2$ of its choice. The functionality $\mathcal{F}_f$ then samples a value w from $f_k(x_1, \tilde{x}_2)$ and hands it to both P_1 and P_2. Now, $\tilde{\mathcal{Z}}$ outputs 1 if and only if $w = v$ (i.e., P_1's output equals v). Furthermore, by the assumed security of Π_f, environment $\tilde{\mathcal{Z}}$ outputs 1 except with negligible probability (because this is the case in a real execution with $\tilde{\mathcal{A}}$). We conclude that $w = v$, except with negligible probability. However, we claim that since x_1 and x_2 are (p, k)-safe values for P_1 and P_2, respectively, it must be the case that with non-negligible probability w does *not* equal v. In order to see this, we distinguish between two cases:

1. *$\tilde{\mathcal{S}}$ sends x_1'' to $\tilde{\mathcal{Z}}$ before sending $\tilde{x}_2$ to $\mathcal{F}_f$*: In this case, $\tilde{\mathcal{Z}}$ samples the value v before $\mathcal{F}_f$ samples the value $w = f_k(x_1, \tilde{x}_2)$. Since x_1 is a (p, k)-safe value for P_1, we have that for every $\tilde{x}_2$ and v, the probability that $f_k(x_1, \tilde{x}_2) \neq v$ is greater than $1/p(k)$.
2. *$\tilde{\mathcal{S}}$ sends $\tilde{x}_2$ to $\mathcal{F}_f$ before sending x_1'' to $\tilde{\mathcal{Z}}$*: In this case we use the reverse argument to above. That is, w is sampled and fixed before $\tilde{\mathcal{Z}}$ samples $v = f_k(x_1'', x_2)$. Therefore, since x_2 is a (p, k)-safe value for P_2, the probability that $f_k(x_1'', x_2) \neq w$ is greater than $1/p(k)$.

We therefore have that for infinitely many k's, environment $\tilde{\mathcal{Z}}$ outputs 0 in the ideal process (because $w \neq v$) with probability greater than $1/p(k)$. In contrast, as we have shown, $\tilde{\mathcal{Z}}$ outputs 1 in the real process except with negligible probability. This contradicts the assumption that Π_f securely realizes $\mathcal{F}_f$. We therefore conclude that an unpredictable function f cannot be securely realized in the plain model. □

Relaxations of UC. The proof of Theorem 6.1 remains unchanged for the relaxation of UC obtained by reversing the order of quantifiers between $\mathcal{Z}$ and $\mathcal{S}$. When considering the relaxation based on limiting $\mathcal{Z}$ to be a uniform machine, we must assume that for infinitely many k's, safe values can be efficiently found by a uniform Turing machine.

Universally composable key exchange. As we have mentioned, the probabilistic functionality of key exchange *can* be securely realized [CK1]. This does not contradict our results here because in the case that one of the participants in a key exchange protocol is corrupted, it is explicitly given the power to single-handedly determine the value of the output key. Therefore, the key exchange functionality is *not* unpredictable (there are no safe values). Note that this makes sense for key exchange because it is only meant to protect two honest parties from an eavesdropping adversary.

References

[B] D. Beaver. Secure Multi-Party Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. *Journal of Cryptology*, 4(2):75–122, 1991.

[BGW] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proc*. 20*th STOC*, pages 1–10, 1988.

[BMM] A. Beimel, T. Malkin and S. Micali. The All-or-Nothing Nature of Two-Party Secure Computation. In *CRYPTO* '99 (LNCS 1666), Springer-Verlag, Berlin, pages 80–97, 1999.

[C1] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proc*. 42*nd FOCS*, pages 136–145, 2001. Preliminary full version available at `http://eprint.iacr.org/2000/067`, 2000.

[C2] R. Canetti. Universally Composable Signatures, Certification and Authentication. In *Proc*. 17*th IEEE Computer Security Foundations Workshop*, pages 219–235, 2004.

[C3] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols (full version). Manuscript, 2004.

[C4] R. Cleve. Limits on the Security of Coin-Flips when Half the Processors are Faulty. In *Proc*. 18*th STOC*, pages 364–369, 1986.

[CF] R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO* 2001 (LNCS 2139), Springer-Verlag, Berlin, pages 19–40, 2001.

[CFGN] R. Canetti, U. Feige, O. Goldreich and M. Naor. Adaptively Secure Multi-Party Computation. In *Proc*. 28*th STOC*, pages 639–648, 1996.

[CK1] R. Canetti and H. Krawczyk. Universally Composable Key Exchange and Secure Channels. In *Eurocrypt* 2002 (LNCS 2332), Springer-Verlag, Berlin, pages 337–351, 2002.

[CK2] B. Chor, and E. Kushilevitz. A Zero–One Law for Boolean Privacy. In *Proc*. 21*st STOC*, pages 62–72, 1989.

[CLOS] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and Multi-Party Computation. In *Proc*. 34*th STOC*, pages 494–503, 2002.

[EGL] S. Even, O. Goldreich and A. Lempel. A Randomized Protocol for Signing Contracts. In *Communications of the ACM*, 28(6):637–647, 1985.

[GL] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO*'90 (LNCS 537), Springer-Verlag, Berlin, pages 77–93, 1990.

[GMW] O. Goldreich, S. Micali and A. Wigderson. How to Play Any Mental Game—A Completeness Theorem for Protocols with Honest Majority. In *Proc*. 19*th STOC*, pages 218–229, 1987.

[HMS] D. Hofheinz, J. Müller-Quade and R. Steinwandt. On Modeling IND-CCA Security in Cryptographic Protocols. Cryptology ePrint Archive, Report 2003/024, `http://eprint.iacr.org/`, 2003.

[HNRR] D. Harnik, M. Naor, O. Reingold and A. Rosen. Completeness in Two-Party Secure Computation—A Computational View. In *Proc*. 36*th STOC*, pages 252–261, 2004.

[K1] J. Kilian. More General Completeness Theorems for Secure Two-Party Computation. In *Proc*. 32*nd STOC*, pages 316–324, 2000.

[K2] E. Kushilevitz. Privacy and Communication Complexity. In *Proc*. 30*th FOCS*, pages 416–421, 1989.

[KKMO] J. Kilian, E. Kushilevitz, S. Micali and R. Ostrovsky. Reducibility and Completeness in Private Computations. *SIAM Journal on Computing*, 29(4):1189–1208, 2000.

[KLP] Y. Kalai, Y. Lindell and M. Prabhakaran. Concurrent General Composition of Secure Protocols in the Timing Model. In *Proc*. 37*th STOC*, pages 644–653, 2005.

[L1] Y. Lindell. General Composition and Universal Composability in Secure Multi-Party Computation. In *Proc*. 44*th FOCS*, pages 394–403, 2003.

[L2] Y. Lindell. Lower Bounds for Concurrent Self Composition. In *Proc*. 1*st Theory of Cryptography Conference* (*TCC*) (LNCS 2951), Springer-Verlag, Berlin, pages 203–222, 2004.

[MR] S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO* '91 (LNCS 576), Springer-Verlag, Berlin, pages 392–404, 1991.

[PS] M. Prabhakaran and A. Sahai. New Notions of Security: Universal Composability without Trusted Setup. In *Proc*. 36*th STOC*, pages 242–251, 2004.

[R] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Technical Memo TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[RB] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In *Proc*. 21*st STOC*, pages 73–85, 1989.